SPL-1 Project Report

Institute of Information Technology
University of Dhaka

Escape The Cop

A network based Game Project

Submitted by

Turya Biswas

BSSE Roll: BSSE-1507

Exam Roll: 231131

BSSE Session: 2022-2023

Submitted to

Dr. Md. Nurul Ahad Tawhid

Associate Professor

Institute of Information Technology, University of Dhaka

Project name

Escape The Cop - A network based Game

Supervised By

Dr. Md. Nurul Ahad Tawhid
Associate Professor
Institute of Information Technology
University of Dhaka

Supervisor Signature : _____

Submitted By

Turya Biswas
Roll: BSSE - 1507
Session : 2022 - 23

Signature : _____

Table Of Contents

List of Figures

1. Introduction

1.1. Project Overview

The SPL-1 project is a comprehensive software application developed to provide an engaging and interactive gaming experience. The project is divided into two main components: the Client and the Server. The Client component handles the game's user interface, levels, and assets, while the Server component manages the backend functionalities, including handling client-server communication and game logic.

1.2. Objectives

The primary objectives of the SPL-1 project are as follows:

1. To develop a user-friendly and visually appealing game interface.
2. To implement a robust server-client architecture for seamless communication.
3. To create multiple pre-built game levels to enhance the gaming experience.
4. To provide tools for analyzing and maintaining the codebase.

These objectives aim to create a fully functional and enjoyable game that can be easily maintained and extended in the future.

1.2.1 Required Libraries and their usage:

1. **Built in library:**
   - `<iostream>`: For system input output task
   - `<string>`: To manipulate c++ string type data
   - `<vector>`: To make dynamic sized array.
   - `<algorithm>`: For data sorting algorithm
   - `<fstream>`: To take input from file-stream or to write into file-stream.
   - `<sstream>`: StringStream to easily set any type data into string variables.
   - `<cmath>`: To use math related functions that comes from gcc.

2. Graphics Library:

- `<SFML/Graphics.h>`**: For graphics interface.**
- `<SFML/Audio.h>`**: For sound effects.**
- `<SFML/Network.h>`**: For network Connection.**

3. OOP: Object-Oriented Programming (OOP) in C++ is a programming paradigm that uses objects and classes to design and develop applications. Here are the key concepts:

- **Class**: A blueprint for creating objects. It defines a datatype by bundling data and methods that work on the data. It can be compared with struct/union in C
- **Object**: An instance of a class. It represents a real-world object. Like in the project, I have created Block object, thief object, police object, fruit object, etc.
- **Encapsulation**: Object oriented programming has encapsulation feature that can be used to restrict use of some specific data within the Class interface but not allow to use these data outside class components.
- **Polymorphism**: Polymorphism allows methods to do different things based on the object it is acting upon, even though they share the same name.

4. Dynamic Array with std::vector: In C++, we can use the std::vector class from the Standard Template Library (STL) to create a 2D dynamic array. This approach is more flexible and easier to manage compared to traditional C-style arrays.

```
std::vector<std::vector<any_instance>> matrix(rows,
std::vector<int>(cols));
```

Figure: Syntax of initializing <vector> array

Here we can replace any_instance with any valid **data-type** in **C++** that is int, char, double, std::string, Any kind of Object like **Block**, etc.

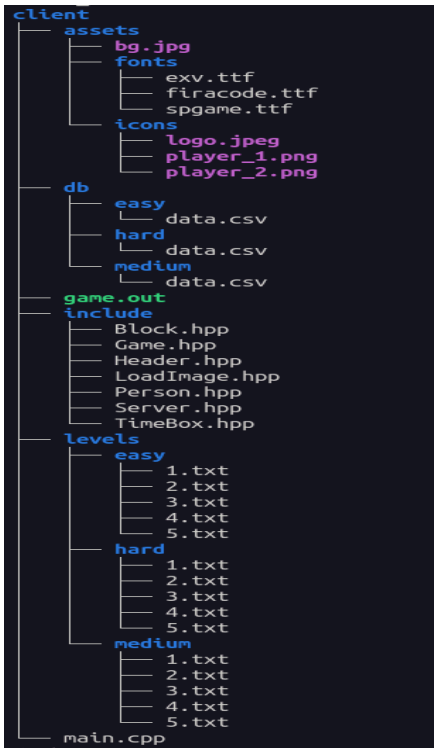
2. Repository Structure

Repository Link: https://github.com/turya07/SPL-1

The SPL-1 repository is organized into several key folders and files, each serving a specific purpose in the development and functionality of the project. Below is and overview of the main folders and files in the repository:

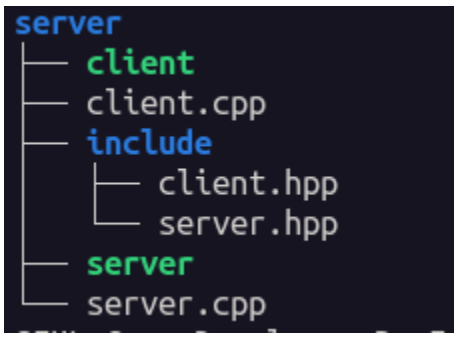
2.1. Overview of Folders and Files

Client Folder:



1. Assets: This folder has some images and fonts used in the game UI.
2. Db: This folder contains a database of each level's score list made by the players previously played in different levels.
3. Include: This folder contains all the important header files for the game.
4. Levels: This folder contains levels of different types of UI in text file format.
5. main.cpp(f): This the main file that is compiled for an executable file to play the game.

Figure: Client Folder Structure



Server Folder:

1. Server.hpp: In include folder, this file is used for creating/initializing a server on host device.
2. Client.hpp: In include folder, this file is used to connect to the host-device server.

NB:- These files are only necessary for on-site gameplay.

Figure: Server Folder Structure

Summary

The repository is structured to separate the client and server components, making it easier to manage and maintain the project. The `Tools` folder provides additional support for code analysis, ensuring the quality and maintainability of the codebase.

3. Client Folder

3.1. Assets Folder

- ➔ **Images: In assets/icon folder, there are three images.**
  - **Logo.jpeg**
  - **player_1.png**
  - **player_2.png**
- ➔ **Fonts: In assets/fonts folder, there are three fonts**
  - **Exv.ttf**
  - **Firacode.ttf**
  - **Spgame.ttf**

Here, exv.ttf is mostly used in the project and the rest of the fonts are barely used in some cases.

3.2 Levels Folder

There are 3 major levels in this game project. These are Easy, Medium & Hard. Each Level contains 5 different types of levels. One Example is shown in Figure 3.2.1. Look at the image. Here the # characters are indicating Blocks of the Game. * characters are indicating the fruit/rewards of a player. 1 & 2 are indicating the initial position of the CPU player.

Figure: Example of Level Structure

3.3 Header Include Folder

The include folder contains various header files essential for the game's functionality:

- *client/include/Header.hpp:*
  - This file includes necessary libraries and constants used throughout the client-side code. It serves as a central point for common includes and definitions.
- *client/include/LoadImage.hpp:*
  - Manages image loading and rendering. This file contains functions and classes responsible for loading images from disk and rendering them on the screen.

```
class LoadImage
{
private:
   sf::Sprite sprite;
   sf::Texture texture;
   int playerId;

public:
   LoadImage();
   LoadImage(const std::string &imagePath, int playerId);
   void draw(sf::RenderWindow &);
   void move(sf::Vector2f);
   ~LoadImage();
};
```

Figure: LoadImage Class

- *client/include/Person.hpp:*
    - Defines the Person class, which represents the player character and other characters in the game. It includes attributes such as position, movement, and interactions.

```
class Person
{

public:
   Person();
   void assignPerson(std::string, unsigned int, std::string,
std::string, std::string, sf::Font);
   void updateScore(unsigned int score);
   int getScore() { return score; }
   void updateLevel(std::string, std::string);
   void deleteIt();
   void draw(sf::RenderWindow &window)
   {
       // std::cout << "Drawing Player: " <<
playerInfo.getString().toAnsiString() << std::endl;
       sprite.draw(window);
       // window.draw(playerInfo);
   }

   void move(sf::Vector2f pos)
   {
       sprite.move(pos);
   }

   // getter
   std::string getPlayerName()
   {
       return playerName;
   }
   std::pair<std::string, int> getLevel()
   {
       return level.getAll();
   }
```

```
    ~Person();

private:
    std::string playerName;
    sf::Text playerInfo;
    unsigned int playerId;
    unsigned int score;
    Level level;
    LoadImage sprite;
};
```

Figure: Person Class

- *client/include/Game.hpp:*
    - Manages game logic and UI. This file contains the main game loop, event handling, and UI updates. It coordinates the various components of the game to ensure smooth gameplay. It has various methods is used to run the game in a better User Experience (UX).

```
class Game
{
public:
    sf::RenderWindow window;

    Game();
    void initGame();
    std::string getName();
    std::string showMenu();
    void playGame(std::string, std::string);
    bool chooseOption();
    void server_client_UI();
    void showPauseMenu(bool);
    ~Game()
    {
    }
```

```
private:
    unsigned int score = 0; // SCORE Variable
    int64_t time = 0;
    int64_t lastTime = 0;
    int totalFruits = 0;
    bool isMenuOpen = false;
    bool isPlayerOne = true; // that denotes server if true else client
    bool isSoloMode = false;
    sf::Font font;
    sf::Text title = sf::Text("Escape The Cop", font, 30);
    sf::Text levelInfo;
    sf::Text otherInfo;
    sf::Text scoreBoard;
    sf::Clock clock;
    sf::Music music;

    std::vector<std::vector<Block>> blocks; // 2d matrix for WALL & GAPS
    std::vector<std::vector<Block>> fruits; // fruits as SCORE

    bool isShowScoreBoard = false;

    // Network objects
    Server *server = nullptr;
    Client *client = nullptr;

    // Person Info
    Person player1, player2;
    TimeBox timebox;

    std::string get_current_ip();
};
```

Figure: Game class

- *client/include/TimeBox.hpp:*
    - Displays elapsed time. This file includes classes and functions to manage and display the game timer, providing feedback to the player on their progress.

```
class TimeBox
{
public:
   TimeBox()
   {
   }
   TimeBox(sf::Font &font, sf::Color col, unsigned int dimension, int
y_axis)
   {
       timerBox.setFont(font);
       timerBox.setCharacterSize(16);
       timerBox.setFillColor(col);
       timerBox.setPosition(dimension - 16 * 16, y_axis);
   }
   void draw(sf::RenderWindow &window, int time)
   {
       timerBox.setString("Time Elapsed: " + std::to_string(time) +
"s");
       window.draw(timerBox);
   }
   ~TimeBox() {};

private:
   sf::Text timerBox;
};
```

Figure: TimeBox class

- *client/include/Block.hpp:*
    - Manages game blocks and movement. This file defines the Block class, which represents obstacles and other static elements in the game. It includes methods for handling block interactions and movements. It has two major roles in the Game.

        - It is used to build wall-blocks and fruits in the game.
        - It is used to make players and cops block.

```
typedef struct ID
{
   int idx;
   int idy;
} ID;

class Block
{
public:
   Block(int width, float x, float y);
   Block(int width, float x, float y, ID id);
   void draw(sf::RenderWindow &window);
   sf::Vector2f getPosition();
   sf::Vector2f getSize();
   void deleteIt();
   bool isColliding();
   void move(sf::Event ev, sf::Vector2f startBoundary, sf::Vector2f
endBoundary, std::vector<std::vector<Block>> blocks);
   void setPosition(sf::Vector2f position);
   void setColor(sf::Color color);
   void setOutlineColor(sf::Color color);
   void setOutlineThickness(float thickness);
private:
   int width;
   int height;
   int speed = 10;
   ID id = {0, 0};
   sf::Vector2f position;
   sf::RectangleShape block;
   sf::Color myColor;
};
```

Figure: Block Class

4. Server Folder

4.1. server/include/server.hpp

The server/include/server.hpp file defines the Server class, which is responsible for handling server-side networking and communication with clients. This file includes the following key components:

- Server Class: Manages client connections, data transmission, and game state synchronization.
- Networking Functions: Functions to initialize the server, accept client connections, and handle incoming and outgoing data.
- Game State Management: Methods to update and broadcast the game state to all connected clients, ensuring a consistent experience across all players.

```
class Server {
public:
    Server(unsigned short port) {
        if (listener.listen(port) != sf::Socket::Done) {
            std::cerr << "Could not listen on port " << port << std::endl;
            return;
        }
        std::cout << "Server is listening on port " << port << std::endl;
    }

    void run() {
        if (listener.accept(client) != sf::Socket::Done) {
            std::cerr << "Could not accept client" << std::endl;
            return;
        }
        std::cout << "Client connected" << std::endl;

        std::thread receiveThread(&Server::receiveData, this);
        receiveThread.detach();
    }

    void sendData(const std::string& data) {
        if (client.send(data.c_str(), data.size() + 1) != sf::Socket::Done) {
            std::cerr << "Failed to send data to client" << std::endl;
        }
    }
```

```
    std::string getReceivedData() {
        return receivedData;
    }

private:
    void receiveData() {
        char buffer[1024];
        std::size_t received;
        while (true) {
            if (client.receive(buffer, sizeof(buffer), received) == sf::Socket::Done) {
                receivedData = std::string(buffer, received);
                std::cout << "Received: " << receivedData << std::endl;
            }
        }
    }

    sf::TcpListener listener;
    sf::TcpSocket client;
    std::string receivedData;
};
```

Figure: Server.hpp code

4.2. server/include/client.hpp

The server/include/client.hpp file defines the Client class, which manages client-side networking and communication with the server. This file includes the following key components:

- Client Class: Handles the connection to the server, sending player actions, and receiving game state updates.
- Networking Functions: Functions to connect to the server, send data, and process incoming data from the server.
- Player Actions: Methods to send player actions (e.g., movement, interactions) to the server for processing and synchronization.

```
class Client {
public:
    Client() {}

    bool connect(const std::string& ip, unsigned short port) {
        std::cout << ip << " " << port << std::endl;
        if (socket.connect(ip, port) != sf::Socket::Done) {
```

```
            std::cerr << "Could not connect to server" << std::endl;
            return false;
        }
        std::cout << "Connected to server" << std::endl;

        std::thread receiveThread(&Client::receiveData, this);
        receiveThread.detach();

        return true;
    }

    void sendData(const std::string& data) {
        if (socket.send(data.c_str(), data.size() + 1) != sf::Socket::Done) {
            std::cerr << "Failed to send data to server" << std::endl;
        }
    }

    std::string getReceivedData() {
        return receivedData;
    }

private:
    void receiveData() {
        char buffer[1024];
        std::size_t received;
        while (true) {
            if (socket.receive(buffer, sizeof(buffer), received) == sf::Socket::Done) {
                receivedData = std::string(buffer, received);
                std::cout << "Received: " << receivedData << std::endl;
            }
        }
    }

    sf::TcpSocket socket;
    std::string receivedData;
};
```

Figure: Client.hpp code

4.3 Overview and Purpose

The server folder contains the core networking code that enables multiplayer functionality in the game. The server.hpp and client.hpp files work together to establish a connection between the server and clients, allowing for real-time communication and synchronization

of game states. This ensures that all players have a consistent and synchronized gameplay experience, regardless of their individual actions.

The server is responsible for maintaining the overall game state, processing player actions, and broadcasting updates to all connected clients. The clients, on the other hand, handle player inputs and send them to the server for processing. This division of responsibilities ensures efficient and scalable multiplayer gameplay.

5. Tools Folder

The tools folder contains utility scripts and tools that assist in the development, testing, and maintenance of the project. These tools help automate repetitive tasks, analyze code quality, and manage project dependencies.

5.1. `calcScore.cpp` File

1. Structure for ScoreBlock:

```
#define INT int64_t
#define ZERO (INT)0
using namespace std;
typedef struct ScoreBlock
{
   string name;
   INT fruites;
   INT time;
   INT score = 0;
} ScoreBlock;
```

Figure: Single Score's data block

2. Score Calculating from fruits and time:

```
INT calculateScore(INT eatenFruits, INT elapsedTime)
{
   INT baseScore = eatenFruits * 10000;
   INT quickBonus = max(ZERO, 50000 - elapsedTime);   // Bonus decreases
as time increases
   INT timePenalty = max(ZERO, elapsedTime - 5000); // Penalty starts
after 5 seconds
   double multiplier = 1.0;
   if (elapsedTime < 500) // If all fruits are eaten within 5 seconds
   {
       multiplier = 1.5;
   }
   else if (elapsedTime < 10000) // If all fruits are eaten within 10
seconds
   {
       multiplier = 1.2;
   }
   // Final score calculation
   INT finalScore = static_cast<INT>((baseScore + quickBonus -
timePenalty) * multiplier);

   return max(ZERO, finalScore);
}
```

Figure: Score calculating function

3. Score File reading function:

```
vector<ScoreBlock> readFile(const string &filename)
{
   vector<ScoreBlock> scores;
   ifstream file(filename);
   if (!file.is_open())
   {
       cout << "Error opening file" << endl;
       return scores;
   }
   string line;
   while (getline(file, line))
   {
       ScoreBlock block;
       block.name = line.substr(0, line.find(";"));
       line.erase(0, line.find(";") + 1);
       block.fruites = stoi(line.substr(0, line.find(";")));
       line.erase(0, line.find(";") + 1);
       block.time = stoi(line);
       scores.push_back(block);
   }
   file.close();
   return scores;
}
```

Figure: File reading Function

4. Format of CSV file:

FORMAT: **Name;fruit_number;elapsed_time**

As they are separated with [semicolon], I have subtracted the first part into block.name, then erased the first part. Similarly subtracted the second part into block.fruits and third part into block.time.

5. Main Function:

```
int main(const int argc, const char *const argv[])
{
    if (argc != 2)
    {
        cout << "Usage: " << argv[0] << " <filename>" << endl;
        return 1;
    }
    vector<ScoreBlock> scores = readFile(argv[1]);
    int laxLenghtName = 0;
    // Calculate scores and store in the scores vector
    for (auto &score : scores)
    {
        score.score = calculateScore(score.fruites, score.time);
        laxLenghtName = max(laxLenghtName, (int)score.name.length());
    }
    // Sort scores in descending order
    sort(scores.begin(), scores.end(), greaterScore);

    // Print sorted scores
    for (auto score : scores)
    {
        cout << score.name << setw(laxLenghtName - score.name.length() +
5) << " : " << score.score << "(" << score.fruites << ")" << "(" <<
score.time/1000 << ")" << endl;
    }
    cout << endl;

    return 0;
}
```

Figure: Main function for score calculation

5.2. count_lines.cpp file:

This file counts the number of lines of code (**loc**) in a specified source file provided in argument command. Note that, this file is not related to project, but the only pre-built tool

I found in linux apt is **cloc** which even gives me a number of lines including white-spaces and also any kinds of text files. So, for better understanding, I have created my own **loc-counter** code. And for easier visualization, I have also written a **SHELL** file to get the total number of lines in the Project.

Code:

1. countLines():

```
int countLines(const std::string &filename)
{
    std::ifstream file(filename);
    if (!file.is_open())
    {
            std::cerr << "Could not open the file: " << filename << std::endl;
        return -1;
    }

    int lines = 0;
    std::string line;
    while (std::getline(file, line))
    {
        if (line.length() > 0)
            ++lines;
    }
    file.close();
    return lines;
}
```

Figure: Read lines of C/C++ code file

2. optimizeFileName():

```
std::string optimizeFileName(std::string filename)
{
   std::string optimized = filename;
   int pos = filename.find_last_of("/");
   if (pos != std::string::npos)
   {
       optimized = filename.substr(pos + 1);
   }
   return optimized;
}
```

Figure: File Name optimizer

Output:

```
============================O============================
Counting Clients lines [DIR: ./clients]:
        1. Total lines in LoadImage.hpp          :  39
        2. Total lines in Person.hpp             :  107
        3. Total lines in Game.hpp               :  862
        4. Total lines in TimeBox.hpp            :  22
        5. Total lines in Block.hpp              :  163
        6. Total lines in main.cpp               :  26
============================================================
Total Lines of C/C++ code                        :  1219
============================X============================
============================O============================
Counting Server  lines [DIR: ./server ]:
        1. Total lines in server.hpp             :  44
        2. Total lines in client.hpp             :  39
============================================================
Total Lines of C/C++ code                        :  83
============================X============================
============================O============================
Counting Tools   lines [DIR: ./tools  ]:
        1. Total lines in count_lines.cpp        :  71
        2. Total lines in calcScore.cpp          :  85
        3. Total lines in list_files.cpp         :  25
============================================================
Total Lines of C/C++ code                        :  181
============================X============================
```

Figure: Output of count_lines.cpp program

6. User Interface

6.1. Graphical Interface based on SFML Library:

This game is made with "**Simple and Fast Multimedia Library**" known as SFML tool that can be used in both windows and linux platforms. I have made this game playable for both devices. But to play this game in windows, there might be some hassle to configure the library files of SFML with "Visual Studio" IDE.

Here I am providing a gameplay walkthrough explanation with some figures of what is going on throughout the game.

6.1.1. User Name

At first, the game will show an interface that will ask player to provide a name. User is allowed to enter `[A-Z]` and `[0-9]` characters.

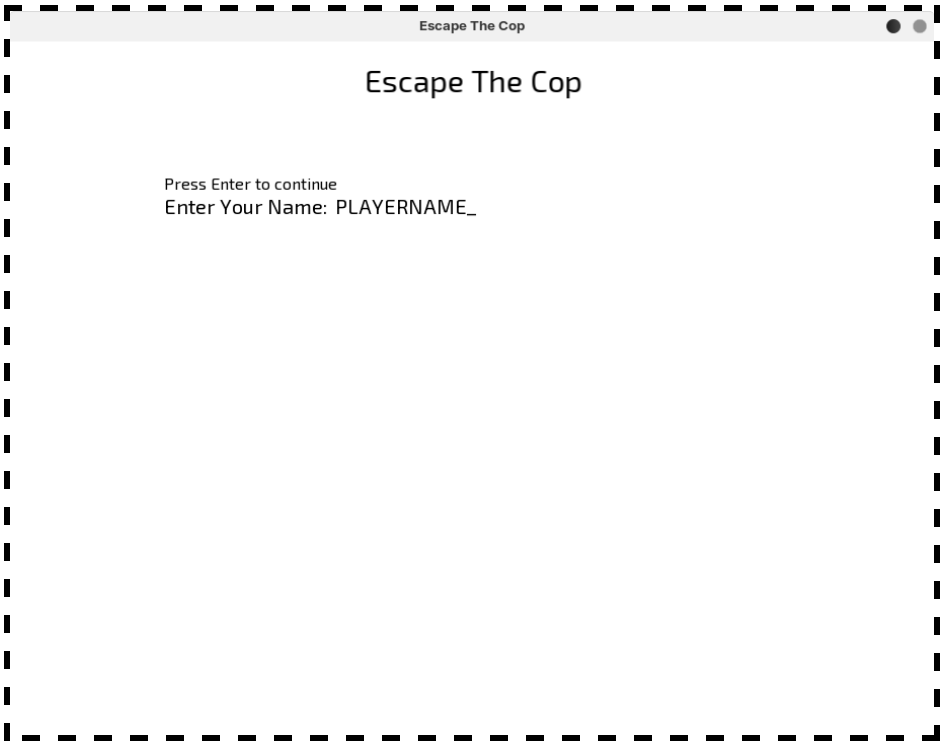


Figure: User Input Interface of Game

6.1.2. Choose a option

Here Player has to choose 1 of 3 different options.

- Create Server
- Join Server
- Solo Mode

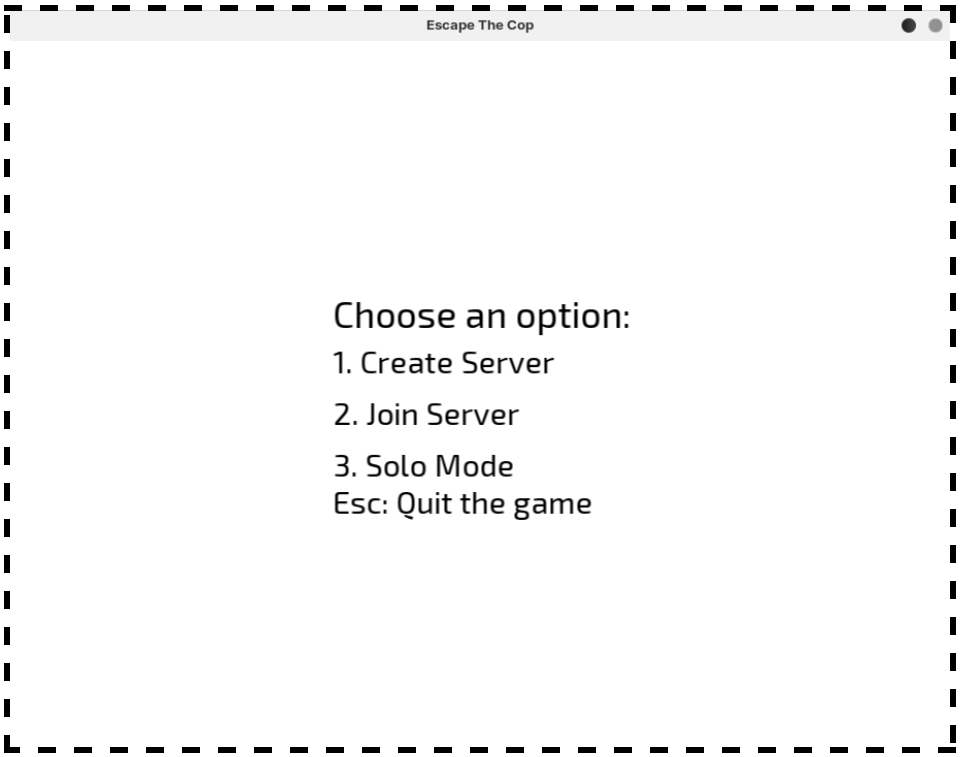


Figure: Option choosing Interface

Create Server: If a player chooses this option, he will play as a **SERVER** host as well as **player-1**. In this case, he will be shown a waiting page until the **CLIENT** connects with the current server.

Join Server: If a player chooses this option, he will play as a **CLIENT** as well as **player-2**. In this case, he will need to input **HOST** IPv4 address and **PORT** number eg. **198.168.0.12** and **8080.** Then press `ENTER` to continue. In this phase, the HOST player can also see if the client has been connected or not.

Solo Mode: If a player chooses this option, he will play as **player-1** but as a solo player, no server connection required.

6.1.3. Level Selection

Player needs to choose a level `EASY`, `MEDIUM` or `HARD`. Then press any `<number-key>` from 1 to 5 to select a specific level number.

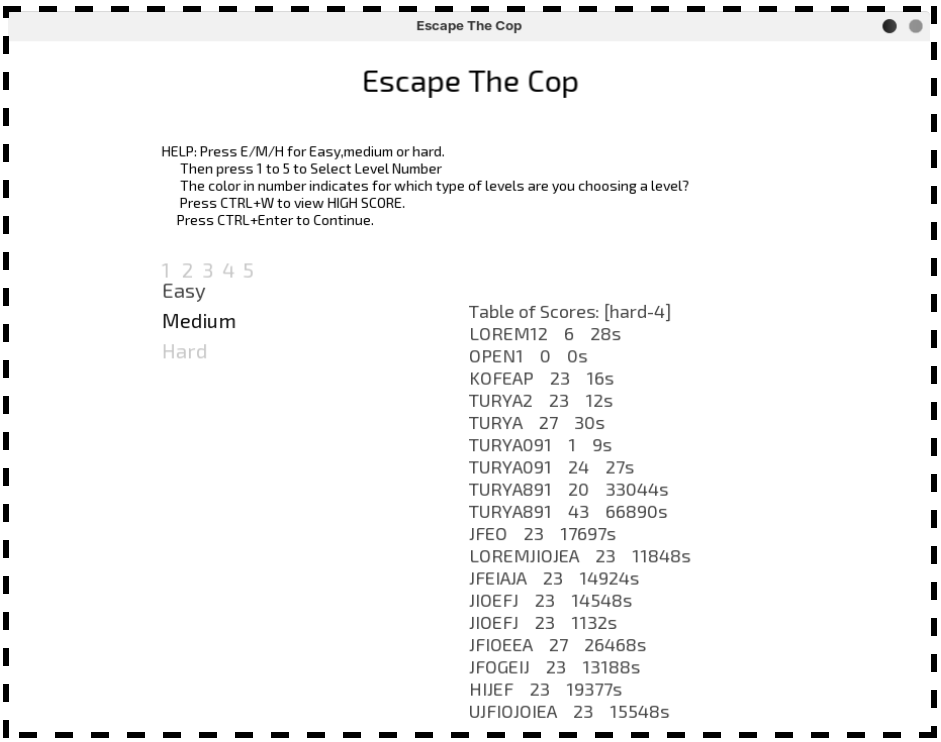


Figure: Level Choosing Interface

6.1.4. Gameplay Stage

In this stage, the game will be started. Player has to use `W, A, S, D` key-map to control player movements.

- **W**: Move Upward ⬆️
- **A**: Move Left Side ⬅️
- **S**: Move Downward ⬇️
- **D**: Move Right Side ➡️

Figure: Gameplay Interface

Here, **White** box indicates **player-1**. **Red** and **Yellow** box are 2 cops that will chase the player.

6.1.5. Pause State

If a user presses the ESC button, the game will temporarily stop/pause. Also the game timer will be stuck on the current time state.

Figure: Pausing State Interface

6.1.6. Finish State

If a player successfully eats all the fruits in the window, he wins. Then a **CONGRATULATIONS** page will be shown.

Figure: Finish State of Game

6.2. Terminal Interface

6.2.1. Calculate the score

Run `./tools/calcScore <path-to-level-csv-file>` file to view score list in **TERMINAL**.

SCORE TABLE of Hard Level:

TURYA891 : 368110(43)(66)
JIOEFJ : 334641(23)(1)
JFIOEEA : 272064(27)(26)
LOREMJIOJEA : 261304(23)(11)
JIOEFJ : 255904(23)(14)
JFEIAJA : 255152(23)(14)

7. Conclusion

In conclusion, The game involves a thief trying to avoid being caught by three cops. The player controls the thief and must navigate through different levels of difficulty, each with various obstacles and cops with different speeds. The game features:

1. **Levels**: There are three levels of difficulty - Easy, Medium, and Hard. Each level has different layouts and obstacles.
2. **Gameplay**: The player must control the thief to avoid the cops. The cops know the direction of the thief but not the exact position.
3. **Scoring**: The game has a timer that acts as the score. Scores are stored in a file, and the top five scores are displayed in the game UI.
4. **Multiplayer**: The game supports both solo and multiplayer modes. In multiplayer mode, one player acts as the server, and the other as the client.
5. **Tools**: The project includes tools for counting lines of code, calculating scores, and listing files.

The project uses **C/C++** and the **SFML** library for graphics and networking. The codebase includes various scripts and source files to handle game logic, networking, and utility functions.

This project helps me learn:
1. How OOP works, how can I handle PUBLIC and Private data. how to use methods.
2. How STRING data can be manipulated.
3. How to assign a 2D array dynamically.
4. When and why to free memory of vector arrays.
5. How to use a loop effectively for continuous tasks.
6. How does pointer works in C/C++
7. I can also learn how to create any logic using mathematical calculations.
8. How to use command line arguments to input data.
9. How to read and write in a file.
10. How does a network work and how to create a server side host and client side interface to send and receive data.

References

1. GeeksforGeeks. "Socket Programming in C/C++." GeeksforGeeks.
Socket with C++
    - Comprehensive tutorial on socket programming fundamentals

2. Communicating with sockets (SFML - 2.6)
SFML Socket Documentation
    - Step-by-step guide to implementing socket communication with SFML

3. *Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022).* Introduction to Algorithms (4th ed.). MIT Press.
    - Reference for the efficient algorithms used in movement logic.

4. SFML Documentation.
SFML Documentation
    - Visual UI implementation guideline taken from SFML documentation.

5. SFML Installation Guideline (Youtube)
SFML on LINUX
    - Guideline on how to install SFML in linux

6. SFML Necessary Tools Guideline (Youtube)
SFML Tool Tutorial
    - Learn how different tools can be used from SFML in code.

7. *Raimondas Pupius.* SFML-Game-Developme-By-Example (1st ed. 2015). Packt-Publishing Ltd. Production reference: *1181215*
    - Book reference for advanced tools information.

Appendix

Required Tools to install the game

[LINUX/UNIX]

1. gcc and g++ compiler: `sudo apt install -y gcc g++`
2. Sfml library: `sudo apt install -y libsfml-dev`

[Mac OS]

1. gcc and g++ compiler: `brew install gcc` & `brew install g++`
2. Sfml library: `brew install sfml@3.0.0`

Compilation Instructions

To compile the Game with Network files:
`sh install.sh`
Code inside install.sh file:

[LINUX/UNIX]

```
cur_path=$pwd
/usr/bin/g++ -o ./client/game.out ./client/main.cpp -lsfml-graphics
-lsfml-window -lsfml-system -lsfml-network -lsfml-audio
/usr/bin/make ./server/server
/usr/bin/make ./server/client

echo "Successfully compiled the game!"
cd ./client
./game.out
cd $cur_path
```

Figure: Compiling code for Linux/Unix

[Mac OS]

```
cur_path=$pwd
/usr/local/bin/g++ -o ./client/game.out ./client/main.cpp
-lsfml-graphics -lsfml-window -lsfml-system -lsfml-network -lsfml-audio
/usr/bin/make ./server/server
/usr/bin/make ./server/client

echo "Successfully compiled the game!"
cd ./client
./game.out
cd $cur_path
```

Figure: Compiling code for MacOS

 # Running Instructions
To run the game anytime,

```
cd ./client
./game.out
```

 # Follow the on-screen instructions to the game.
System Requirements :
- C++ compiler with C++11 support or higher
- Operating system (Linux/Unix/MacOS)
- Local Network connectivity for client-server communication