Project Report

Department & Institution

Computer Science & Engineering,
Khulna University of Engineering & Technology
(***CSE-KUET***)

Course

CSE 1206 (***Object Oriented Programming***)

Project

Terminal-Based 2D Game (***Tunneler***)

by

Md. Shifat Hasan
(2107067)

Introduction to OOP with C++

In the dynamic field of computer science, Object-Oriented Programming (OOP) stands as a cornerstone methodology essential for robust, scalable, and maintainable software design.

C++, a language renowned for its efficiency and expressive power, serves at the forefront of this paradigm, contributing significantly to the creation of intricate software systems. Object-oriented programming, organizing code into reusable objects, promotes clear separation of concerns, enabling developers to design and implement complex systems comprehensibly.

The academic exploration focuses on the principles of Object-Oriented Programming with C++, covering foundational concepts such as encapsulation, inheritance, and polymorphism.

Additionally, advanced topics including templates, exception handling, and the Standard Template Library (STL) are explored, enhancing the expressive capacity of C++ for the development of scalable and maintainable software systems.

This academic journey aims to equip readers with the knowledge and skills to effectively leverage C++ for object-oriented design, fostering a nuanced understanding of the symbiotic relationship between Object-Oriented Programming and the C++ language.

Citation:

OpenAI. (2023). *ChatGPT* (September 25 Version) [Large language model]. https://chat.openai.com

OOP Features in C++

Classes and Objects
User-defined data types with encapsulated data and methods.

Encapsulation
Hides internal details and restricts data access.

Inheritance
Subclasses inherit properties and behaviors from base classes.

Polymorphism
Allows treating different classes as a common base class.
Includes compile-time and runtime polymorphism.

Abstraction
Simplifies complex systems through class modeling.
Abstract classes and pure virtual functions support abstraction.

Operator Overloading
Defines custom behaviors for operators with user-defined types.

Function Overloading and Overriding
Defines multiple functions with the same name but different parameter lists.

Templates
Supports generic programming for any data type.

Multiple Inheritance
Allows a class to inherit from more than one base class.

Dynamic Memory Allocation
new and ***delete*** for efficient memory management.

Constructors and Destructors
Initialize and clean up object resources.

Friend Functions
Accesses private and protected class members.

Static Members
Belong to the class rather than instances.

Namespaces
Organizes code and prevents naming conflicts.

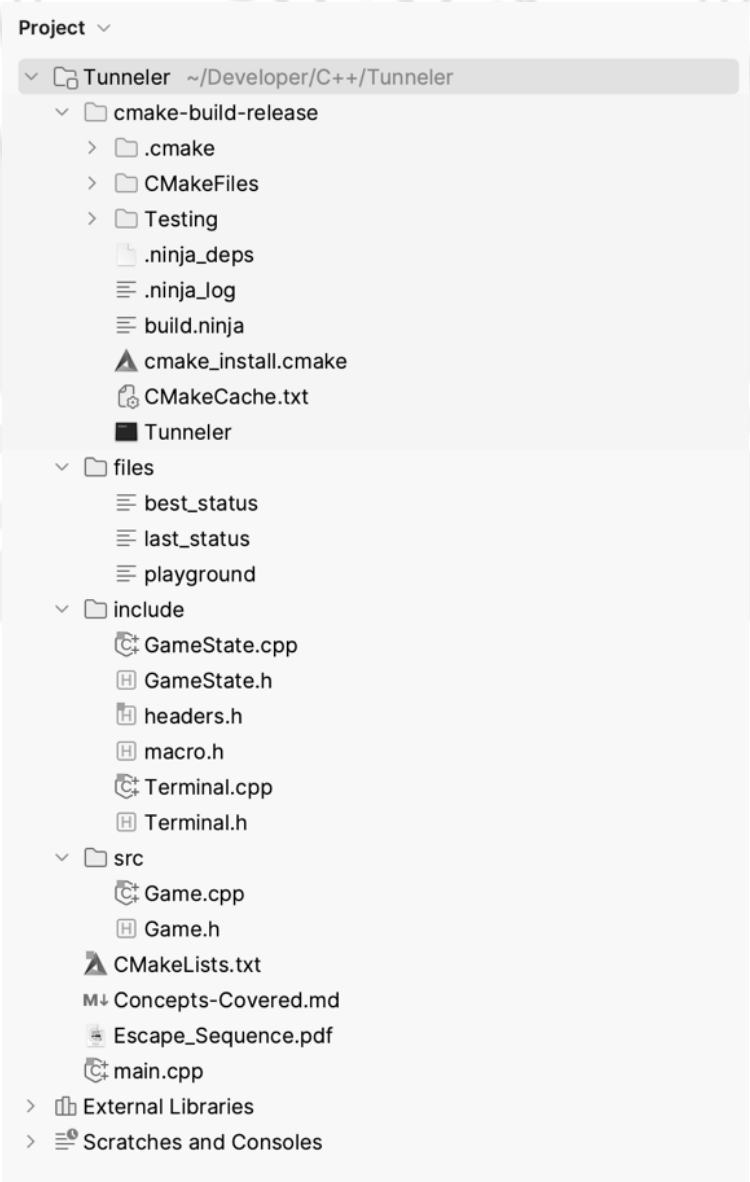
Exception Handling
try, ***catch,*** and ***throw*** for robust error handling.

Standard Template Library (STL)
Template classes and functions for common data structures and algorithms.

The Features I've Used in My Project

- </> **Classes and Objects**
- </> **Encapsulation**
- </> **Inheritance**
- </> **Polymorphism**
- </> **Abstraction | Pure Virtual Function**
- </> **Function Overriding**
- </> **Constructors and Destructors**
- </> **Static Class-Members**
- </> **File Management**

Project's File Structure

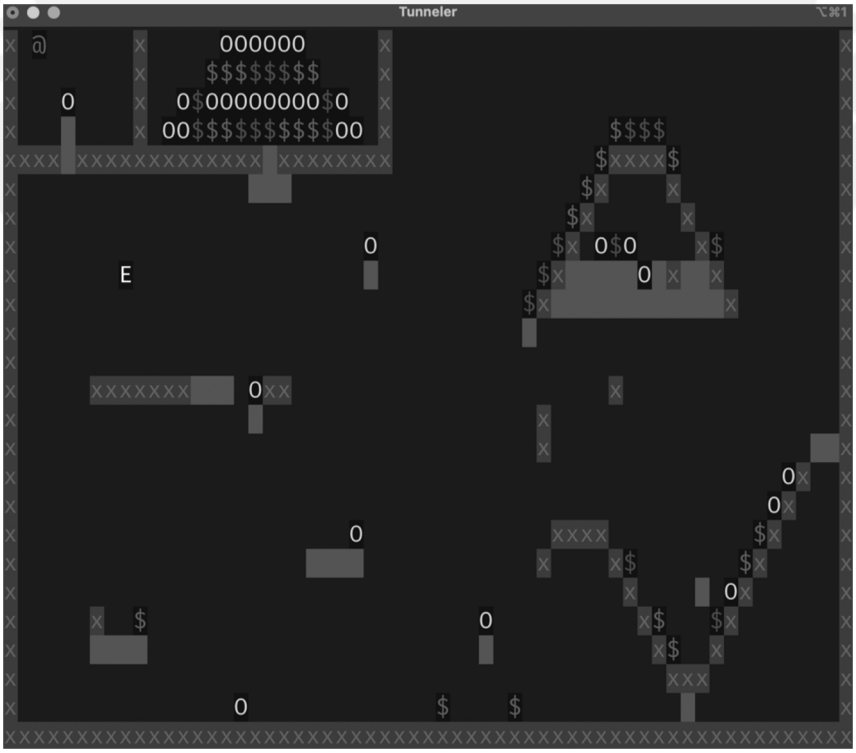


Playground

playground

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
x @ 0    x   0000000000   x                                    x
x        x   $$$$$$$$$$   x   $                $    0   x
x        x   0000000000   x                                    x
x   .    x   $$$$$$$$$$   x             $$$$$                  x
xxxx.xxxxxxxxxxxxx.xxxxxxxx             $xxxx$       0   x
x                ...                    $x    x              x
x                                       $x 00   x              x
x                        0              $x  $     x            x
x       E                .              $x.....0.x..x          x
x                                       $x............x       x
x                                       .                      x
x                                                $$            x
x     xxxxxxx ... 00xx                        x       0        x
x                .                        x                     x
x                                        x     $       $$    ..x
x                                                $     x  x
x        $                                                x   x
x                       0              xxxx           x     x
x                    ....              x    x         x      x
x                                           x    .  x        x
x     x                          0          x     x         x
x     ....                       .            x   x         x
x                                             xxx           x
x                                               .           x
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

↓ After Rendering ↓



macro.h

macro.h

```h
#ifndef TAMEZX_MACRO_H
#define TAMEZX_MACRO_H

#include <cstdio>
#include <string>

enum class Key {
    None, Up, Down, Right, Left
};

const std::string Workspace = "/Users/[redacted]/Developer/C++/Tunneler";

#define TUNNELER printf("\033]50;SetProfile=Tunneler\a")
#define MainTerminal printf("\033]50;SetProfile=Terminal\a")

#define FLUSH fflush(stdout); fflush(stdin)

#define FRAME_ROWS 26
#define FRAME_COLS 60
// 1 / (0.1 seconds) = approximately 10 Frames per Second
#define FRAME_CYCLE (double)0.1

#define WINDOW_RESIZE(_H_, _W_) printf("\033[8;%d;%dt", (_H_), ((_W_) - 1))
#define SCREEN_CLEAR_ALL printf("\033[1;1H\033[2J\033[3J")

#define CURSOR_HIDE printf("\033[?25l")
#define CURSOR_SHOW printf("\033[?25h")
#define CURSOR_POSITION(_row_, _col_) printf("\033[%d;%dH", (_row_), (_col_))

#define RGBA_FG(_R_, _G_, _B_, _A_) printf("\033[38;2;%d;%d;%d;%dm", (_R_), (_G_), (_B_), (_A_))
#define RGBA_BG(_R_, _G_, _B_, _A_) printf("\033[48;2;%d;%d;%d;%dm", (_R_), (_G_), (_B_), (_A_))
#define RGBA_RESET printf("\033[m")

#endif //TAMEZX_MACRO_H

```

This header file provides a collection of macros and constants for managing terminal behavior and appearance.

Header Guard

The ***#ifndef***, ***#define***, and ***#endif*** directives are used to create a header guard, preventing multiple inclusions of the same file.

Enumeration Class: *Key*

Enumeration for keyboard keys with values such as ***None***, ***Up***, ***Down***, ***Right***, and ***Left***.

Constant String: *Workspace*

Specifies the workspace path as *"/Users/***/Developer/C++/Tunneler"*.

TUNNELER and *MainTerminal*

Macros for changing terminal profiles.

FLUSH

Defines a macro for flushing the standard output and input streams.

Frame Dimensions and Cycle

Specifies constants for frame rows, columns, and the frame cycle duration.

Terminal Manipulation

Resize the terminal window using ***WINDOW_RESIZE***.

For clearing the screen and moving the cursor to the top-left position use ***SCREEN_CLEAR_ALL***.

Cursor Control

For hiding and showing the cursor use ***CURSOR_HIDE*** and ***CURSOR_SHOW***.

For setting the cursor position ***CURSOR_POSITION***.

Color Formatting

For setting foreground and background colors using RGBA values using macros ***RGBA_FG***, ***RGBA_BG.***

For resetting color attributes using ***RGBA_RESET***.

Citation:

ANSI escape code. (2023, September 28). In *Wikipedia*. https://en.wikipedia.org/wiki/ANSI_escape_code

Terminal.h

Terminal.h

```
#ifndef TUNNELER_TERMINAL_H
#define TUNNELER_TERMINAL_H

#include <csignal>
#include <cstdlib>
#include <unistd.h>
#include <termios.h>

#include "macro.h"

static struct termios old_termios, new_termios;

class Terminal {
public:
    static void Set();

    static void Reset();
};

#endif //TUNNELER_TERMINAL_H

```

Header Guard

The ***#ifndef***, ***#define***, and ***#endif*** directives are used to create a header guard, preventing multiple inclusions of the same file.

Header File Inclusions

Includes necessary header files, such as <csignal>, <cstdlib>, <unistd.h> and <termios.h>. Also, includes a custom header file named ***macro.h*** (containing macro definitions).

Static Variables

Declares two static instances of the ***termios*** structure: ***old_termios*** and ***new_termios***. These will be used to store the terminal settings.

Class: Terminal

Declares a class named ***Terminal*** with public methods only.

Static Methods

Set(): This static method is responsible for setting the terminal configurations. It modifies the terminal settings for specific functionality.

Reset(): This static method is intended to reset the terminal configurations to their original state, using the stored values of ***old_termios***.

Terminal.cpp

Terminal.cpp

```cpp
#include "Terminal.h"

void Terminal::Reset() {
    RGBA_RESET;
    CURSOR_POSITION(1, 1);
    tcsetattr(STDIN_FILENO, TCSANOW, &old_termios);
    fflush(stdin);
    fflush(stdout);
    CURSOR_SHOW;
    SCREEN_CLEAR_ALL;
}

void Terminal::Set() {
    SCREEN_CLEAR_ALL;
    WINDOW_RESIZE(FRAME_ROWS, FRAME_COLS);
    tcgetattr(STDIN_FILENO, &old_termios);

    new_termios = old_termios; // save it to be able to Reset on exit

    new_termios.c_lflag &= ~(ICANON | ECHO); // set for non-canonical mode, no echo
    new_termios.c_cc[VMIN] = 0;
    new_termios.c_cc[VTIME] = 0;

    tcsetattr(STDIN_FILENO, TCSANOW, &new_termios);

    CURSOR_HIDE;

    fflush(stdout);
}

```

Terminal::Reset()

Color Reset and Cursor Position:
Resets the terminal color attributes using the ***RGBA_RESET*** macro.
Moves the cursor to the top-left position using ***CURSOR_POSITION(1, 1)***.

Restore Terminal Settings:
Restores the original terminal settings saved in ***old_termios*** using ***tcsetattr(STDIN_FILENO, TCSANOW, &old_termios)***.

Terminal::Set()

Save Original Terminal Settings:
Saves the original terminal settings in ***old_termios*** using ***tcgetattr(STDIN_FILENO, &old_termios)***.

Modify Terminal Settings:
Creates a copy of the original settings in ***new_termios***.
Modifies ***new_termios*** for ***non-canonical mode*** and ***no echo*** (disabling line buffering and local echo).
Configures ***VMIN*** and ***VTIME*** to set the terminal to ***non-blocking mode***.
Applies the modified settings using ***tcsetattr(STDIN_FILENO, TCSANOW, &new_termios)***.

GameState.h

GameState.h

```h
#ifndef TUNNELER_GAMESTATE_H
#define TUNNELER_GAMESTATE_H

#include <iostream>
#include <fstream>
#include <unistd.h>

#include "macro.h"
#include "Terminal.h"

using namespace std;

class GameState : public Terminal {
protected:
    char player;
    int pos_row, pos_col;
    unsigned int count;
    int gems_collected;
    bool won, dead;
    char old_screen[FRAME_ROWS][FRAME_COLS];
    char screen[FRAME_ROWS][FRAME_COLS];

    // Called only from the Listen_Event() public function
    static Key Read_Key(const char *, int);

public:
    Key key;

    // Constructors and Destructors
    virtual ~GameState() = default;
    GameState() = default;
    GameState(char);

    // Function Override; Same Prototype, but totally different definition
    void Reset();

    // For the Game
    void load_level(const string &game_level_file_path);
    void backup();
    void find_player_position(char);
    void Listen_Event();

    // Pure Virtual Functions
    virtual void Start(const string &) = 0;
    virtual void Update() = 0;
    virtual void Render() = 0;
};

#endif //TUNNELER_GAMESTATE_H
```

These declarations are quite self-explanatory.

This header file declares a ***GameState*** class, which is a base class for managing the state of a game. It inherits publicly from the ***Terminal*** class and includes several data members and member functions for game state management. Some functions are virtual, indicating that they need to be implemented by derived classes.

GameState.cpp

GameState.cpp

```cpp
#include "GameState.h"

GameState::GameState(char _player) :
        player(_player), pos_row(0), pos_col(0), count(0), gems_collected(0), won(false), dead(false) {}
```

It’s the constructor to define the player.

GameState.cpp

```cpp
void GameState::Reset() {
    for (int r = 0; r < FRAME_ROWS; r++) {
        for (int c = 0; c < FRAME_COLS; c++) {
            screen[r][c] = ' ';
            screen[r][c] = ' ';
        }
    }
}

```

It can reset the screen variables to space [‘ ‘] character.

GameState.cpp

```cpp
void GameState::load_level(const string &game_level_file_path) {
    fstream file(game_level_file_path, ios::in);
    if (file.is_open()) {
        for (int r = 0; r < FRAME_ROWS; r++) {
            for (int c = 0; c < FRAME_COLS; c++) {
                screen[r][c] = file.get();
            }
        }
    } else {
        std::cout << "Sorry! Unable to load the file (" << game_level_file_path << ").\n";
    }
    file.close();
}

```

it’s to load the level or playground in the terminal.

GameState.cpp

```cpp
void GameState::backup() {
    for (int r = 0; r < FRAME_ROWS; r++) {
        for (int c = 0; c < FRAME_COLS; c++) {
            old_screen[r][c] = screen[r][c];
        }
    }
}

```

It’s to backup the latest screen to the previous screen.

GameState.cpp

```cpp
void GameState::find_player_position(char _player) {
    for (int r = 0; r < FRAME_ROWS; r++) {
        for (int c = 0; c < FRAME_COLS; c++) {
            if (screen[r][c] == _player) {
                pos_row = r;
                pos_col = c;
                return;
            }
        }
    }
}

```

It can find the player on the screen.

GameState.cpp

```cpp
Key GameState::Read_Key(const char *buffer, int i) {
    if (buffer[i] == '\033' && buffer[i + 1] == '[') {
        switch (buffer[i + 2]) {
            case 'A':
                return Key::Up;
            case 'B':
                return Key::Down;
            case 'C':
                return Key::Right;
            case 'D':
                return Key::Left;
        }
    }
    return Key::None;
}

void GameState::Listen_Event() {
    char buffer[4096];
    int n_read = (int) read(STDIN_FILENO, buffer, sizeof(buffer));
    Key final_key = Key::None;
    for (int i = 0; i ⩽ n_read - 3; i += 3) {
        Key _key;
        _key = GameState::Read_Key(buffer, i);
        if (_key == Key::None)
            continue;
        final_key = _key;
    }
    key = final_key;
}

```

The ***Listen_Event()*** Function can detect a ***Keyboard-Arrow-Key-Press*** and as per the Key-Press-Event, the ***Read_Key()*** Function can uniquely identify the Key and return the result to the ***Listen_Event()*** Function so that it can register the final Key-Press into the member variable key of the ***GameState*** class.

headers.h

headers.h

```
#ifndef TUNNELER_HEADERS_H
#define TUNNELER_HEADERS_H

#include "macro.h"
#include "Terminal.h"
#include "GameState.h"

#endif //TUNNELER_HEADERS_H

```

Game.h

Game.h

```
#ifndef TAMEZX_GAME_H
#define TAMEZX_GAME_H

#include "../include/headers.h"

class Game : public GameState {
private:
    void handle_player();
    void handle_falling_rocks_gems(int row, int col);
    void handle_rocks_gems(int row, int col);

    void update_all_elements();

public:
    Game(char _player) : GameState(_player) {}

    void Start(const string &);

    void Update();
    void Render();
    void End_Message();
};

#endif //TAMEZX_GAME_H
```

This is the dedicated class for the game.

Game.cpp

Game.cpp

```
1    #include "Game.h"
2
3    void Game::handle_player() {
4        switch (key) {
5            case Key::Up:
6                switch (screen[pos_row - 1][pos_col]) { ... }
25               break;
26           case Key::Down:
27               switch (screen[pos_row + 1][pos_col]) { ... }
42               break;
43           case Key::Right:
44               switch (screen[pos_row][pos_col + 1]) { ... }
67               break;
68           case Key::Left:
69               switch (screen[pos_row][pos_col - 1]) { ... }
92               break;
93           case Key::None:
94           default:
95               break;
96       }
97   }
```

It can control the player's position and properties based on Key-Press.

Game.cpp

```
99   void Game::handle_falling_rocks_gems(int row, int col) {
100      int falling_gem = screen[row][col] == 's';
101
102      // start to fall
103      if (screen[row + 1][col] == ' ') {
104          screen[row][col] = ' ';
105          screen[row + 1][col] = falling_gem ? 's' : 'o';
106          return;
107      }
108
109      if (screen[row + 1][col] == 'O' || screen[row + 1][col] == '$') // check left
110      {
111          // check left
112          if (screen[row][col - 1] == ' ' && screen[row + 1][col - 1] == ' ') { ... }
117          // check right
118          if (screen[row][col + 1] == ' ' && screen[row + 1][col + 1] == ' ') { ... }
123      }
124
125      if (screen[row + 1][col] == 'o' || screen[row + 1][col] == 's') return;
126
127      if (screen[row + 1][col] == '@') dead = 1;
128
129      screen[row][col] = falling_gem ? '$' : 'O';
130  }
```

It handles the falling rock's and gem's position and properties.

Game.cpp

```cpp
void Game::handle_rocks_gems(int row, int col) {
    int gem = screen[row][col] == '$';

    // start to fall
    if (screen[row + 1][col] == ' ') {
        screen[row][col] = gem ? 's' : 'o';
        return;
    }

    if (screen[row + 1][col] == 'O' || screen[row + 1][col] == '$') {
        // check left
        if (screen[row][col - 1] == ' ' && screen[row + 1][col - 1] == ' ') {
            screen[row][col] = gem ? 's' : 'o';
            return;
        }
        // check right
        if (screen[row][col + 1] == ' ' && screen[row + 1][col + 1] == ' ') {
            screen[row][col] = gem ? 's' : 'o';
            return;
        }
    }
}
```

It handles the rock's and gem's position and properties, not the falling ones.

Game.cpp

```cpp
void Game::update_all_elements() {
    // We will iterate over the screen from bottom to top from right to left
    for (int r = FRAME_ROWS - 1; r > 0; r--) {
        for (int c = FRAME_COLS - 2; c > 0; c--) {
            switch (screen[r][c]) {
                case 'p':
                    screen[r][c] = 's';
                    break;
                case 'i':
                    screen[r][c] = 'o';
                    break;
                case 'O':
                case '$':
                    handle_rocks_gems(r, c);
                    break;
                case 'o':
                case 's':
                    handle_falling_rocks_gems(r, c);
                    break;
                default:
                    break;
            }
        }
    }
}
```

It updated all the elements present on the screen based on expected effects.

Game.cpp

```cpp
void Game::Update() {
    memcpy(screen, old_screen, sizeof(screen));

    handle_player();
    update_all_elements();

    ++count;
}
```

It backs up the latest screen. Then, handles the player and all the elements by calling the previously discussed functions.

Game.cpp

```cpp
void Game::Render() {
    for (int r = 0; r < FRAME_ROWS; r++) {
        for (int c = 0; c < FRAME_COLS; c++) {
            char ch = GameState::screen[r][c];

            // Change gem's color frequently
            if (ch == '$' || ch == 's') {
                int l = (int) ((GameState::count + r + 3 * c) % 16) / 8;
                CURSOR_POSITION(r + 1, c + 1);
                if (l == 0) { ... } else { ... }
                continue;
            }

            CURSOR_POSITION(r + 1, c + 1);

            switch (ch) {
                case '\n':
                case '\r':
                    cout << "\n";
                    break;
                case 'x': { ... }
                case '.': { ... }
                case ' ': { ... }
                case 'O': { ... }
                case 'o': { ... }
                case '@': { ... }
                case 'E': { ... }
                default:
                    break;
            }
        }
    }
    fflush(stdout);
}
```

It can render the screen in the terminal as expected.

Game.cpp

```cpp
void Game::End_Message() {
    SCREEN_CLEAR_ALL;
    WINDOW_RESIZE(7, FRAME_COLS);
    CURSOR_POSITION(1, 1);

    int gems;
    ifstream ifile("../files/best_status");
    ifile >> gems;
    ifile.close();

    ofstream ofile("../files/last_status");
    ofile << GameState::gems_collected;
    ofile.close();

    if (GameState::gems_collected > gems) { ... }

    if (GameState::dead) { ... } else if (GameState::won) { ... }
}
```

It shows an end message while exiting the game.

Game.cpp

```cpp
void Game::Start(const string &game_level_file_path) {
    Terminal::Set();
    struct timespec sleep_time;

    GameState::Reset();
    GameState::load_level(game_level_file_path);
    Game::Render();
    GameState::find_player_position(player);
    GameState::backup();

    clock_t start, end;

    // Game Loop
    while (true) {
        start = clock();

        GameState::Listen_Event();
        Game::Update();
        if (won || dead) break;
        Game::Render();
        GameState::backup();

        end = clock();

        double time_taken = (double) (end - start) / CLOCKS_PER_SEC;
        if (time_taken > FRAME_CYCLE) continue;

        sleep_time.tv_sec = 0;
        sleep_time.tv_nsec = (FRAME_CYCLE - time_taken) * 1000000000;
        nanosleep(&sleep_time, NULL);
    }

    Terminal::Reset();
    Game::End_Message();
}
```

It starts the game, maintains the game loop, and does all necessary things to keep playing the game till winning or dying.

And, at last, there's a main() function to start the program altogether.

In conclusion, learning **OOP** with **C++** equips programmers with a powerful set of tools and concepts that enhance code quality, promote efficient development practices, and contribute to the creation of scalable and maintainable software systems.

GitHub Repository

GitHub Link:

https://github.com/ShifatHasanGNS/2107067-OOP-Project.git

Or, Scan This QR Code:

After **October 8, 2023**, you will lose access to this QR Code