Programming in C++

2D Maze Game Development



Submitted by:

Preetika Khatri

L-4 CS&DF

Student ID – 23085138

Submitted on: 21/04/2025

Github URL- https://github.com/preetika47/MazeGame

# Overview of Assessment

For this assessment, we were tasked to create a 2D maze game that runs in a console window. The game involves navigating a player character through a maze, collecting items, avoiding enemies, and reaching an exit, with features like multiple levels.

The game was made in C++ using standard libraries and Windows-specific functions for console manipulation (conio.h, windows.h). The code is structured into a header file (Maze.h) defining game entities and a source file (Maze.cpp) containing the main game logic (MazeGame class) and the main function.

***The source code Maze.cpp:***

// Maze.cpp

#include <iostream>

#include <fstream>

#include <vector>

#include <random>

#include <ctime>

#include <string> // Explicitly include string

#include <windows.h> // For Sleep() and console handles

#include <conio.h> // For \_getch()

#include "Maze.h" // Include the header defining Player and Enemy

using namespace std;

// Windows specific getch alias

#define GETCH() \_getch()

// Renamed from SIZE to avoid potential conflicts (e.g., with windows.h macros)

const int MAZE\_DIMENSION = 10;

class MazeGame {

private:

// Use the renamed constant for the maze dimensions

char maze[MAZE\_DIMENSION][MAZE\_DIMENSION];

Player\* player;

vector<Enemy\*> enemies;

int score;

int moves;

int level;

int collectibles;

HANDLE hConsole; // Handle for console colors

int defaultColor; // Store default console color

// Function to clear the console screen (Windows specific)

void clearScreen() {

COORD coordScreen = { 0, 0 };

DWORD cCharsWritten;

CONSOLE\_SCREEN\_BUFFER\_INFO csbi;

DWORD dwConSize;

if (!GetConsoleScreenBufferInfo(hConsole, &csbi)) return;

dwConSize = csbi.dwSize.X \* csbi.dwSize.Y;

if (!FillConsoleOutputCharacter(hConsole, (TCHAR)' ', dwConSize, coordScreen, &cCharsWritten)) return;

if (!GetConsoleScreenBufferInfo(hConsole, &csbi)) return;

if (!FillConsoleOutputAttribute(hConsole, csbi.wAttributes, dwConSize, coordScreen, &cCharsWritten)) return;

SetConsoleCursorPosition(hConsole, coordScreen);

}

// Function to set text color

void setColor(int colorCode) {

SetConsoleTextAttribute(hConsole, colorCode);

}

void initializeMaze() {

// Use MAZE\_DIMENSION

for (int i = 0; i < MAZE\_DIMENSION; i++) {

for (int j = 0; j < MAZE\_DIMENSION; j++) {

// Use MAZE\_DIMENSION - 1 for bounds

if (i == 0 || i == MAZE\_DIMENSION - 1 || j == 0 || j == MAZE\_DIMENSION - 1 || (i%2 != 0 && j%2 != 0))

maze[i][j] = '#';

else

maze[i][j] = ' ';

}

}

maze[1][1] = ' '; // Ensure start is clear

// Use MAZE\_DIMENSION - 2 for exit position

maze[MAZE\_DIMENSION - 2][MAZE\_DIMENSION - 2] = 'E';

addCollectibles();

}

void addCollectibles() {

static mt19937 gen(static\_cast<unsigned int>(time(0)));

// Use MAZE\_DIMENSION - 2 for distribution bounds

uniform\_int\_distribution<> dis(1, MAZE\_DIMENSION - 2);

collectibles = level + 2;

for (int i = 0; i < collectibles; i++) {

int r, c;

do {

r = dis(gen);

c = dis(gen);

// Use MAZE\_DIMENSION - 2 for exit check

} while (maze[r][c] != ' ' || (r == 1 && c == 1) || (r == MAZE\_DIMENSION - 2 && c == MAZE\_DIMENSION - 2));

maze[r][c] = '\*';

}

}

public:

MazeGame() : score(0), moves(0), level(1), collectibles(0), defaultColor(7) { // Initialize defaultColor

hConsole = GetStdHandle(STD\_OUTPUT\_HANDLE);

CONSOLE\_SCREEN\_BUFFER\_INFO csbi;

if (GetConsoleScreenBufferInfo(hConsole, &csbi)) {

defaultColor = csbi.wAttributes; // Get actual default color

}

initializeMaze(); // Now maze is correctly declared

player = new Player(1, 1); // Uses Player from Maze.h

// Ensure enemy start position is clear before placing

int start\_enemy\_x = 5;

int start\_enemy\_y = 5;

// Basic bounds check for initial enemy placement

if (start\_enemy\_x >= 0 && start\_enemy\_x < MAZE\_DIMENSION && start\_enemy\_y >= 0 && start\_enemy\_y < MAZE\_DIMENSION) {

if (maze[start\_enemy\_x][start\_enemy\_y] == '#') {

maze[start\_enemy\_x][start\_enemy\_y] = ' '; // Clear if wall

}

enemies.push\_back(new Enemy(start\_enemy\_x, start\_enemy\_y)); // Uses Enemy from Maze.h

} else {

// Handle error or place enemy at a default valid spot if 5,5 is out of bounds

if (maze[1][2] == ' ') enemies.push\_back(new Enemy(1,2));

else enemies.push\_back(new Enemy(2,1)); // Failsafe

}

}

~MazeGame() {

delete player;

for (auto enemy : enemies) {

delete enemy;

}

enemies.clear();

}

// --- Public Getters Added ---

int getScore() const { return score; }

int getPlayerX() const { return player->getX(); }

int getPlayerY() const { return player->getY(); }

int getCollectiblesRemaining() const { return collectibles; }

char getCell(int x, int y) const {

// Added bounds checking for safety

if (x >= 0 && x < MAZE\_DIMENSION && y >= 0 && y < MAZE\_DIMENSION) {

return maze[x][y];

}

return '#'; // Return wall if out-of-bounds request

}

// --- End Getters ---

void display() {

clearScreen();

const int COLOR\_WALL = 13; // Magenta / Purple

const int COLOR\_PLAYER = 14; // Yellow

const int COLOR\_ENEMY = 12; // Bright Red

const int COLOR\_ITEM = 11; // Bright Cyan / Yellowish

const int COLOR\_EXIT = 10; // Bright Green

cout << "--- Maze Game --- Level: " << level << " ---\n";

// Use MAZE\_DIMENSION

for (int i = 0; i < MAZE\_DIMENSION; i++) {

for (int j = 0; j < MAZE\_DIMENSION; j++) {

bool drawn = false;

if (i == player->getX() && j == player->getY()) {

setColor(COLOR\_PLAYER);

cout << player->getSymbol() << " ";

drawn = true;

} else {

for (auto enemy : enemies) {

if (i == enemy->getX() && j == enemy->getY()) {

setColor(COLOR\_ENEMY);

cout << enemy->getSymbol() << " ";

drawn = true;

break;

}

}

}

if (!drawn) {

char cell = maze[i][j]; // Safe now: maze is accessible

if (cell == '#') setColor(COLOR\_WALL);

else if (cell == '\*') setColor(COLOR\_ITEM);

else if (cell == 'E') setColor(COLOR\_EXIT);

else setColor(defaultColor); // Empty space

cout << cell << " ";

}

setColor(defaultColor); // Reset color after each cell

}

cout << endl;

}

setColor(defaultColor); // Ensure color is reset after the loop

cout << "Score: " << score << " | Moves: " << moves << " | Collectibles left: " << collectibles << endl;

}

bool movePlayer(char direction) {

int dx = 0, dy = 0;

switch (tolower(direction)) {

case 'w': dx = -1; break;

case 's': dx = 1; break;

case 'a': dy = -1; break;

case 'd': dy = 1; break;

default: return false;

}

int newX = player->getX() + dx;

int newY = player->getY() + dy;

// Use MAZE\_DIMENSION for bounds check

if (newX >= 0 && newX < MAZE\_DIMENSION && newY >= 0 && newY < MAZE\_DIMENSION && maze[newX][newY] != '#') {

if (maze[newX][newY] == '\*') {

score += 10;

collectibles--;

maze[newX][newY] = ' '; // Clear collectible

}

player->move(dx, dy);

moves++;

// It might make more sense to move enemies \*before\* checking game state,

// but we'll keep original logic for now.

moveEnemies();

return true;

}

return false;

}

void moveEnemies() {

// Seed slightly differently or just once if preferred

static mt19937 gen(static\_cast<unsigned int>(time(0)) + 1);

uniform\_int\_distribution<> dis(-1, 1);

for (auto enemy : enemies) {

int dx = 0, dy = 0, newX = 0, newY = 0;

int attempts = 0;

const int maxAttempts = 10; // Prevent infinite loops if stuck

do {

dx = dis(gen);

dy = dis(gen);

// Allow diagonal moves if dx and dy are both non-zero, but prevent standing still

if (dx == 0 && dy == 0) continue;

newX = enemy->getX() + dx;

newY = enemy->getY() + dy;

attempts++;

// Check bounds using MAZE\_DIMENSION and ensure it's not a wall or exit

} while ( !(newX >= 0 && newX < MAZE\_DIMENSION && newY >= 0 && newY < MAZE\_DIMENSION && maze[newX][newY] != '#' && maze[newX][newY] != 'E') &&

attempts < maxAttempts);

// Only move if a valid move was found within attempts

if (attempts < maxAttempts) {

// Check for collision with player \*before\* moving enemy onto player space? Optional.

// if (newX == player->getX() && newY == player->getY()) { /\* Handle potential immediate collision \*/ }

enemy->move(dx, dy);

}

}

}

bool checkGameState() {

// Check for collision with player

for (auto enemy : enemies) {

if (player->getX() == enemy->getX() && player->getY() == enemy->getY()) {

// Use Windows API for color instead of potentially unsupported ANSI codes here

setColor(12); // Bright Red

cout << "\n--- Ouch! Caught by an enemy! ---" << endl;

setColor(defaultColor);

return false; // Game over

}

}

// Check for reaching the exit

// Use MAZE\_DIMENSION-2 to correctly identify exit location defined in initializeMaze

if (player->getX() == MAZE\_DIMENSION - 2 && player->getY() == MAZE\_DIMENSION - 2) {

if (collectibles == 0) {

setColor(10); // Bright Green

cout << "\n--- Level " << level << " Complete! --- Moving to next level..." << endl;

setColor(defaultColor);

level++;

resetLevel();

Sleep(1500); // Pause

return true; // Continue playing (next level)

} else {

// Player is at exit, but hasn't collected everything (handled in main loop display now)

return true; // Continue playing (still on current level)

}

}

return true; // Continue playing (no game-ending event occurred)

}

void resetLevel() {

initializeMaze(); // Re-generates walls, exit, and collectibles

player->x = 1; // Reset player position (direct access okay within class)

player->y = 1;

// Reposition existing enemies, ensuring they aren't in walls

int enemy\_start\_x = 5;

int enemy\_start\_y = 5;

for (size\_t i = 0; i < enemies.size(); ++i) {

// Simple repositioning logic, could be improved

int new\_ex = (enemy\_start\_x + i) % (MAZE\_DIMENSION - 2) + 1; // Spread them out a bit

int new\_ey = (enemy\_start\_y + i) % (MAZE\_DIMENSION - 2) + 1;

// Ensure the spot is clear

while (maze[new\_ex][new\_ey] != ' ' || (new\_ex == 1 && new\_ey == 1)) {

new\_ey++;

if (new\_ey >= MAZE\_DIMENSION - 1) {

new\_ey = 1;

new\_ex++;

if (new\_ex >= MAZE\_DIMENSION - 1) {

new\_ex = 1; // Wrap around

}

}

}

enemies[i]->x = new\_ex; // Direct access okay within class

enemies[i]->y = new\_ey;

maze[new\_ex][new\_ey] = ' '; // Make sure spot is marked traversable if we cleared a wall

}

// Add new enemies based on level

if (level >= 3 && enemies.size() < (level / 2) + 1) {

int new\_enemy\_x = 3, new\_enemy\_y = 3;

// Find an empty spot for the new enemy

while(maze[new\_enemy\_x][new\_enemy\_y] != ' ' || (new\_enemy\_x == 1 && new\_enemy\_y == 1)) {

new\_enemy\_y++;

if(new\_enemy\_y >= MAZE\_DIMENSION - 1) { new\_enemy\_y = 1; new\_enemy\_x++; }

if(new\_enemy\_x >= MAZE\_DIMENSION - 1) new\_enemy\_x = 1; // Wrap around search

}

enemies.push\_back(new Enemy(new\_enemy\_x, new\_enemy\_y));

maze[new\_enemy\_x][new\_enemy\_y] = ' '; // Ensure spot is traversable

}

// moves = 0; // Optional: Reset moves per level? Currently cumulative.

}

void saveGame(const string& filename) {

ofstream file(filename);

if (file.is\_open()) {

file << level << " " << score << " " << moves << " " << collectibles << endl;

file << player->getX() << " " << player->getY() << endl; // Use getters for consistency if desired, though direct access is fine here

file << enemies.size() << endl;

for(const auto& enemy : enemies) {

file << enemy->getX() << " " << enemy->getY() << endl;

}

// Use MAZE\_DIMENSION

for (int i = 0; i < MAZE\_DIMENSION; i++) {

for (int j = 0; j < MAZE\_DIMENSION; j++)

file << maze[i][j];

file << endl;

}

file.close();

cout << "\n--- Game Saved to " << filename << " ---" << endl;

Sleep(1000);

} else {

cerr << "Error: Could not open file " << filename << " for saving!" << endl;

Sleep(1000);

}

}

void loadGame(const string& filename) {

ifstream file(filename);

if (file.is\_open()) {

// Temporary variables for reading

int loadedLevel, loadedScore, loadedMoves, loadedCollectibles;

int px, py;

size\_t enemyCount;

// Read stats

file >> loadedLevel >> loadedScore >> loadedMoves >> loadedCollectibles;

if (file.fail()) { cerr << "Error reading game stats from save file." << endl; file.close(); return; }

// Read player position

file >> px >> py;

if (file.fail()) { cerr << "Error reading player position from save file." << endl; file.close(); return; }

// Read enemy count

file >> enemyCount;

if (file.fail()) { cerr << "Error reading enemy count from save file." << endl; file.close(); return; }

// --- If all reads up to this point succeed, apply changes ---

level = loadedLevel;

score = loadedScore;

moves = loadedMoves;

collectibles = loadedCollectibles;

player->x = px; // Direct access okay within class

player->y = py;

// Clear old enemies and load new ones

for (auto enemy : enemies) { delete enemy; }

enemies.clear();

for(size\_t i = 0; i < enemyCount; ++i) {

int ex, ey;

file >> ex >> ey;

if (file.fail()) {

cerr << "Error reading enemy " << i << " position from save file." << endl;

// Clear potentially partially loaded enemies to avoid issues

for(auto enemy : enemies) { delete enemy; }

enemies.clear();

file.close();

return; // Abort load

}

enemies.push\_back(new Enemy(ex, ey));

}

// Read maze grid

string line;

getline(file, line); // Consume the rest of the line after the last enemy coords

// Use MAZE\_DIMENSION

for (int i = 0; i < MAZE\_DIMENSION; i++) {

if (!getline(file, line) || line.length() < MAZE\_DIMENSION) { // Check line length

cerr << "Error reading maze grid line " << i << " or line too short from save file." << endl;

file.close();

// Potentially revert game state or handle error more gracefully

return; // Abort load

}

for (int j = 0; j < MAZE\_DIMENSION; j++) {

maze[i][j] = line[j];

}

}

file.close();

cout << "\n--- Game Loaded from " << filename << " ---" << endl;

Sleep(1000);

} else {

cerr << "Error: Could not open file " << filename << " for loading!" << endl;

Sleep(1000);

}

}

}; // End of MazeGame class

int main() {

MazeGame game;

char input;

bool running = true;

// Welcome Message (using Windows API for color consistency)

HANDLE hConsoleMain = GetStdHandle(STD\_OUTPUT\_HANDLE);

CONSOLE\_SCREEN\_BUFFER\_INFO csbiMain;

int defaultColorMain = 7;

if (GetConsoleScreenBufferInfo(hConsoleMain, &csbiMain)) {

defaultColorMain = csbiMain.wAttributes;

}

auto SetColorMain = [&](int color) { SetConsoleTextAttribute(hConsoleMain, color); };

SetColorMain(11); // Bright Cyan

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

cout << " Welcome to the Maze Game! " << endl;

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

SetColorMain(defaultColorMain);

cout << " Guide their player (P) through the maze. " << endl;

SetColorMain(15); // Bright White

cout << "----------------------------------------------------------------------------------" << endl;

SetColorMain(defaultColorMain);

cout << " \* Use W to move up, A to move left, S to move down, D to move right. " << endl;

cout << " \* Collect \* for 10 points, reach 'E' after collecting all \* to level up. " << endl;

cout << " \* Avoid 'X' enemies! Collision is Game Over. " << endl;

cout << " \* Save game with 'V', load game with 'L', quit with 'Q'. " << endl;

SetColorMain(15); // Bright White

cout << "----------------------------------------------------------------------------------" << endl;

// Highlight start prompt

SetConsoleTextAttribute(hConsoleMain, BACKGROUND\_BLUE | FOREGROUND\_RED | FOREGROUND\_GREEN | FOREGROUND\_BLUE | FOREGROUND\_INTENSITY); // White text on Blue background

cout << " Press any key to start... ";

SetColorMain(defaultColorMain); // Reset color

cout << endl;

GETCH(); // Wait for key press

while (running) {

game.display(); // Game handles its own colors during display

// Display reminder if at exit but items remain - USE GETTERS

int currentPX = game.getPlayerX();

int currentPY = game.getPlayerY();

// Use getCell and getCollectiblesRemaining getters

if(game.getCell(currentPX, currentPY) == 'E' && game.getCollectiblesRemaining() > 0) {

SetColorMain(14); // Yellow

cout << "\nCollect all '\*' before exiting!" << endl;

SetColorMain(defaultColorMain);

}

SetColorMain(11); // Bright Cyan for prompt

cout << "\nMove (w/a/s/d), Save (v), Load (l), Quit (q): ";

SetColorMain(defaultColorMain);

input = GETCH();

// cout << input << endl; // Optional: Echo input back

switch (tolower(input)) {

case 'w':

case 'a':

case 's':

case 'd':

if (game.movePlayer(input)) {

// Check game state \*after\* potentially moving and enemies moving

running = game.checkGameState();

}

break;

case 'v':

game.saveGame("maze\_save.txt");

// Keep running = true after save

break;

case 'l':

game.loadGame("maze\_save.txt");

// Keep running = true after load attempt

break;

case 'q':

running = false;

cout << "\nQuitting game..." << endl;

break;

default:

// cout << "\nInvalid key!" << endl; // Optional feedback

// Sleep(50); // Optional small delay

break; // Do nothing for invalid input

}

} // Game Over message (using main's color setting)

SetConsoleTextAttribute(hConsoleMain, BACKGROUND\_BLUE | FOREGROUND\_RED | FOREGROUND\_GREEN | FOREGROUND\_BLUE | FOREGROUND\_INTENSITY); // White text on Blue background

cout << "\n\nGame Over! Final Score: ";

SetColorMain(14 | BACKGROUND\_BLUE); // Yellow on Blue background

cout << game.getScore(); // Use getter for score

SetColorMain(defaultColorMain); // Reset to default

cout << endl << endl;

cout << "Press any key to exit." << endl;

GETCH();

return 0; }

***Header file Maze.h***

// Maze.h

#ifndef GAME\_ENTITY\_H // Start of include guard

#define GAME\_ENTITY\_H

// A simple base class for game objects with position and symbol

class GameEntity {

protected: // Protected allows derived classes (Player, Enemy) to access directly

int x, y;

char symbol;

public:

// Constructor to initialize position and symbol

GameEntity(int startX, int startY, char sym) : x(startX), y(startY), symbol(sym) {}

// Virtual destructor is good practice for base classes, though not strictly needed here if no complex cleanup

virtual ~GameEntity() = default; // Use default destructor

// Method to update the entity's position

// Note: We make x and y accessible directly in MazeGame for simplicity in the provided solution,

// but stricter encapsulation would use setters or friend classes.

void move(int dx, int dy) {

x += dx;

y += dy;

}

// Getter methods for position and symbol

int getX() const { return x; } // Mark getters as const

int getY() const { return y; } // Mark getters as const

char getSymbol() const { return symbol; } // Mark getters as const

// Friend declaration (optional): Allows MazeGame direct access to protected/private members if needed

// friend class MazeGame; // Uncomment this if MazeGame needs to directly modify x, y of Player/Enemy

// Make MazeGame::resetLevel and MazeGame::loadGame friends to allow direct modification of x, y

friend class MazeGame; // Grant friendship to the entire class for simplicity here

};

// Player class, derived from GameEntity

class Player : public GameEntity {

public:

// Constructor specifically for Player, setting symbol to 'P'

Player(int startX, int startY) : GameEntity(startX, startY, 'P') {}

};

// Enemy class, derived from GameEntity

class Enemy : public GameEntity {

public:

// Constructor specifically for Enemy, setting symbol to 'X'

Enemy(int startX, int startY) : GameEntity(startX, startY, 'X') {}

};

**The code fulfills the Task Specification given by the module tutor in the following ways:**

### **Basic Framework Requirements(30 marks)**

1. **Making the Game World (10 marks)**: The game world is in a 10x10 grid, made using a 2D array in the MazeGame class. This grid has walls, open paths, collectibles (\*), and an exit. kkkkkkkkkkkkkkkkk
2. **Showing the Maze (10 marks)**: The maze is displayed using symbols:
   * # for walls (solid barriers)
   * P for the player
   * E for the exit
   * \* for collectibles
   * X for enemies
3. **Player Movement (10 marks)**: User can control the player using the **W, A, S, D keys**:
   * W (up), A (left), S (down), D (right). The game listens for their keypress and moves the player around the maze.

### **Game State and Interaction (20 marks)**

1. **Save and Load the Game (10 marks)**: The User can save their progress. The game writes important data like their score, moves, player position, and the layout to a file. Later, they can load the file to resume where they left off.
2. **Tracking the Player (5 marks)**: The game always knows where they are in the maze (their X and Y coordinates) so it can show their position and check for collisions with walls, enemies, or collectibles.
3. **Walls Collision (5 marks)**: If they try to move into a wall (#), the game won't let them. They have to find another way through the maze.
4. **Score and Moves (5 marks)**: At the bottom of the screen, their score and the number of moves they’ve made are shown. They gain points by collecting items and completing levels.

### **Advanced Features (20 marks)**

1. **Collecting Treasure (**\***) (5 marks)**: (\*) are scattered around the maze. When Player grabs one, their score goes up, and it disappears from the maze. To finish a level, they need to collect all of them.
2. **Enemies Move Around (**X**) (5 marks)**: There are enemies (X) that move randomly in the maze. They can’t walk through walls or the exit, but if they touch the Player(P), they lose the game.
3. **Game Over and Winning (5 marks)**:
   * Game Over: If an enemy catches Player, it’s game over.
   * Winning: To win a level, collect all the treasures and reach the exit (E). They move to the next level.
4. **Multiple Levels (5 marks)**: After finishing a level, the next one gets harder. There will be more enemies or treasures to collect. Each level feels like a fresh challenge.

### **Code Structure and Documentation (30 marks)**

1. **Neat Code Files (10 marks)**: The code is well-organized:
   * Header file (Maze.h): Holds the blueprint for the game's classes, like Player, Enemy, etc.
   * Source file (Maze.cpp): Contains the actual game logic and functions.
2. **Smart Class Design (10 marks)**: The code is written in a smart way to avoid repeating work.
3. **Clear Comments (5 marks)**: The code has comments that explain what each part does, like how it moves the player, detects collisions, and handles enemies.
4. **Guide to Playing (5 marks)**: A separate file (README.md) gives instructions on how to set up and play the game, including the controls and features.

**Conclusion:**

In conclusion, my Maze Game C++ code provides a thorough and functional solution. It successfully implements all requirements outlined by the module tutor. It shows a solid understanding and practical application of fundamental C++ concepts, including 2D array manipulation, file I/O operations, user input handling, game state management, and object-oriented principles.

All things considered, finishing this project has been an amazing learning opportunity. I would like to thank the module tutor for giving me this difficult yet useful labor and for the insightful advice they provided during its creation.