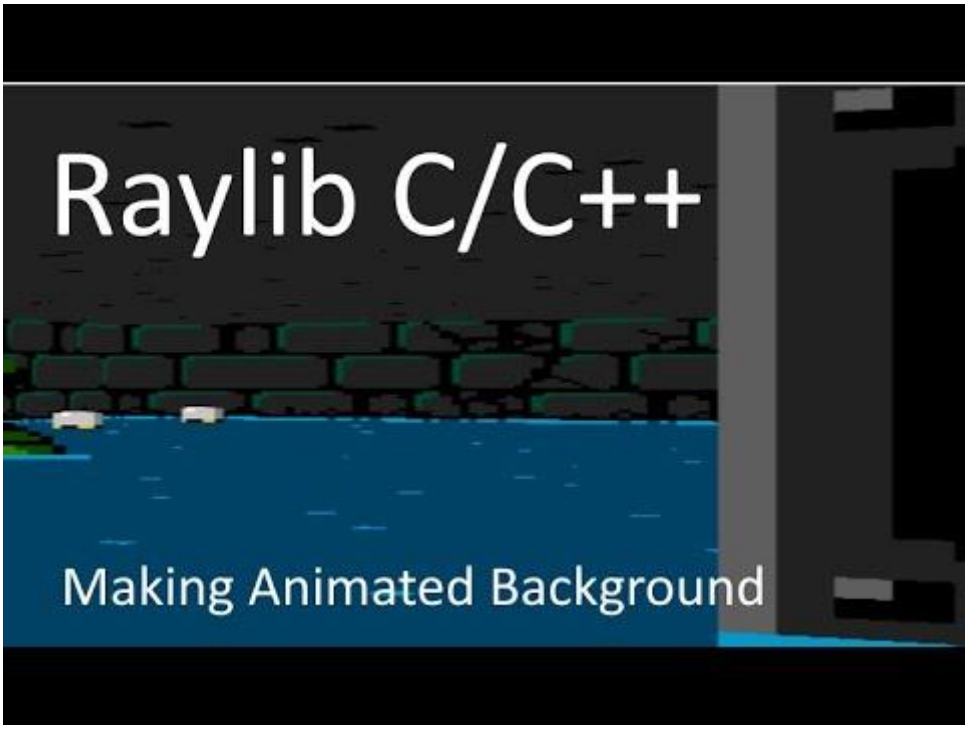
FACULTE DES SCIENCES ET
THECNIQUES DE TANGER

DEPARTEMENT DE INFORMATIQUE

Rapport :

Programmation orientee objet en C++

PFM : POO C++/Raylib



Encadre par :
Ikram Ben abdel ouahab

Realise par :

- Rabih Senhaji Anas
- El Kajdouhi Mohamed Ayman
- Akdi Fouad
- Afazaz Ilyas

Introduction :

Ce rapport présente une analyse détaillée du code source d'un jeu de labyrinthe écrit en C++ utilisant la bibliothèque Raylib. Le jeu propose des fonctionnalités variées, notamment un mode solo ou multijoueur, une sélection de difficulté, et une gestion des scores. Le code est structuré autour de plusieurs classes pour une meilleure organisation et modularité.

Principe de PFM :

Ce projet consiste en un jeu de labyrinthe développé en C++ avec Raylib, où le joueur navigue dans un labyrinthe généré dynamiquement pour atteindre un objectif, avec des niveaux de difficulté variés et un mode multijoueur.

Structure globale :

Le programme est composé des éléments suivants :

- **Classes principales** : Maze, Player, Game, et Level.
- **Enumération** : GameState pour gérer les états du jeu (menu, jeu, sélection de difficulté, etc.).
- **Bibliothèques utilisées** : Raylib pour l'interface graphique et audio.
- **Boucle principale** : Contrôle l'état du jeu et appelle les méthodes associées.

1) Classe Maze (Génération du labyrinthe) :

-Rôle de la classe :

La classe Maze gère la génération et le rendu du labyrinthe. Elle utilise l'algorithme DFS (Depth-First Search) pour créer des chemins complexes. Les cellules du labyrinthe sont modélisées par la structure Cell.

-Structure Cell :

```cpp
// Cell structure for maze generation
struct Cell {
    int x, y;
    bool visited = false;
    bool walls[4] = { true, true, true, true }; // top, right, bottom, left
};
```

Chaque cellule a :

- Ses coordonnées (x, y).
- Un état visited pour suivre si elle a été visitée.
- Des murs représentés par un tableau de booléens.

-Attributs :

- **grid** : Un vecteur qui stocke toutes les cellules du labyrinthe.
- **wallColor** : Permet de modifier dynamiquement la couleur des murs du labyrinthe.

-Algorithme DFS :

L'algorithme DFS est utilisé pour générer un labyrinthe parfait (sans boucles). Il fonctionne en explorant récursivement les cellules voisines non visées.

```cpp
void GenerateMaze() {
    grid.clear();
    grid.resize(mazeCols * mazeRows);

    // Initialize grid
    for (int y = 0; y < mazeRows; ++y) {
        for (int x = 0; x < mazeCols; ++x) {
            grid[y * mazeCols + x] = { x, y };
        }
    }

    // DFS maze generation
    std::stack<Cell*> stack;
    Cell* current = &grid[0];
    current->visited = true;

    while (true) {
        Cell* next = GetUnvisitedNeighbor(*current);
        if (next) {
            stack.push(current);
            RemoveWalls(*current, *next);
            current = next;
            current->visited = true;
        }
        else if (!stack.empty()) {
            current = stack.top();
            stack.pop();
        }
        else {
            break;
        }
    }
}
```

- **Initialisation** : Chaque cellule est marquée comme non visitée avec des murs intactes.
- **Pile** : Une pile (à l'aide de std::stack) stocke les cellules pour retourner en arrière si nécessaire.
- **Parcours DFS** :
  1. Visite une cellule.
  2. Recherche une voisine non visitée (à l'aide de GetUnvisitedNeighbor).
  3. Supprime les murs entre la cellule courante et la suivante (à l'aide de RemoveWalls).
  4. Retourne en arrière si aucune voisine n'est disponible.

-Méthode `GetUnvisitedNeighbor :`

Cette méthode retourne une cellule voisine non visitée de la cellule donnée. Elle est essentielle pour l'algorithme DFS, car elle détermine les prochaines cellules à visiter.

```
Cell* GetUnvisitedNeighbor(const Cell& cell) {
    std::vector<Cell*> neighbors;

    int dx[] = { 0, 1, 0, -1 };
    int dy[] = { -1, 0, 1, 0 };

    for (int i = 0; i < 4; ++i) {
        int nx = cell.x + dx[i];
        int ny = cell.y + dy[i];

        if (nx >= 0 && nx < mazeCols && ny >= 0 && ny < mazeRows) {
            Cell* neighbor = &grid[ny * mazeCols + nx];
            if (!neighbor->visited) {
                neighbors.push_back(neighbor);
            }
        }
    }

    if (!neighbors.empty()) {
        return neighbors[std::rand() % neighbors.size()];
    }
    return nullptr;
}
```

Détails :

- Parcourt les quatre directions cardinales (haut, droite, bas, gauche).
- Vérifie si une cellule voisine existe dans les limites du labyrinthe et n'a pas été visitée.
- Retourne une cellule voisine aléatoire

-Méthode `IsWall` :

Permet de vérifier si un mur existe dans une direction donnée à partir des coordonnées d'une cellule.

```
bool IsWall(int x, int y, int dir) const {
    int index = y * mazeCols + x;
    if (index < 0 || index >= (int)grid.size()) return true;
    return grid[index].walls[dir];
}
```

Détails :

- Prend les coordonnées `(x, y)` et une direction (`dir`) comme arguments.
- Retourne `true` si un mur bloque la direction spécifiée

-Suppression des murs :

```
void RemoveWalls(Cell& a, Cell& b) {
    int dx = b.x - a.x;
    int dy = b.y - a.y;

    if (dx == 1) { a.walls[1] = false; b.walls[3] = false; }
    if (dx == -1) { a.walls[3] = false; b.walls[1] = false; }
    if (dy == 1) { a.walls[2] = false; b.walls[0] = false; }
    if (dy == -1) { a.walls[0] = false; b.walls[2] = false; }
}
```

Cette méthode met à jour les murs entre deux cellules voisines selon leur direction relative.

-Dessin du labyrinthe :

```
void DrawMaze() const {

    for (const Cell& cell : grid) {
        int x = cell.x * cellSize;
        int y = cell.y * cellSize;

        if (cell.walls[0]) DrawLine(x, y, x + cellSize, y, wallColor);           // Top
        if (cell.walls[1]) DrawLine(x + cellSize, y, x + cellSize, y + cellSize, wallColor); // Right
        if (cell.walls[2]) DrawLine(x, y + cellSize, x + cellSize, y + cellSize, wallColor); // Bottom
        if (cell.walls[3]) DrawLine(x, y, x, y + cellSize, wallColor);           // Left
    }
}
```

Chaque mur est dessiné avec DrawLine si son état est true.

2) Classe Player (Gestion du joueur):

-Rôle de la classe :

La classe Player représente le joueur dans le labyrinthe. Elle gère sa position, ses mouvements, et l'affichage.

-Attributs supplémentaires :

- **PLAYERIMAGE** : Texture utilisée pour représenter visuellement le joueur dans le labyrinthe.
- **goalTexture** : Texture utilisée pour représenter l'objectif final.
- **posX, posY** : Coordonnées de la position actuelle du joueur.

-Code principal :

-Positionnement et mise à jour :

```
void SetPosition(int x, int y) {
    posX = x;
    posY = y;
}
```

```cpp
void Update(const Maze& maze, bool isPlayer2 = false, float deltaTime = 0.016f) {
    static float moveCooldown = 0.1f; // Minimum time (in seconds) between moves
    static float timeSinceLastMove = 0.0f;

    // Update the time since the last move
    timeSinceLastMove += deltaTime;

    // If not enough time has passed, do nothing
    if (timeSinceLastMove < moveCooldown) {
        return;
    }
    int newX = posX;
    int newY = posY;

    if (isPlayer2) { // Controls for player 2
        PLAYERIMAGE = LoadTexture("assets/oo_PLAYER.png");
        if (IsKeyDown(KEY_W) && !maze.IsWall(posX, posY, 0)) newY--;
        if (IsKeyDown(KEY_D) && !maze.IsWall(posX, posY, 1)) newX++;
        if (IsKeyDown(KEY_S) && !maze.IsWall(posX, posY, 2)) newY++;
        if (IsKeyDown(KEY_A) && !maze.IsWall(posX, posY, 3)) newX--;
    }
    else { // Default controls
        if (IsKeyDown(KEY_UP) && !maze.IsWall(posX, posY, 0)) newY--;
        if (IsKeyDown(KEY_RIGHT) && !maze.IsWall(posX, posY, 1)) newX++;
        if (IsKeyDown(KEY_DOWN) && !maze.IsWall(posX, posY, 2)) newY++;
        if (IsKeyDown(KEY_LEFT) && !maze.IsWall(posX, posY, 3)) newX--;
    }

    if (newX >= 0 && newX < mazeCols && newY >= 0 && newY < mazeRows) {
        posX = newX;
        posY = newY;
        timeSinceLastMove = 0.0f;
    }
}
```

Gère les mouvements du joueur en fonction des touches pressées et vérifie si le chemin est libre.

Détails :

- Implémente un délai minimum entre les déplacements pour éviter des mouvements trop rapides.
- Différencie les commandes selon qu'il s'agit du joueur 1 ou 2 (mode multijoueur).
- Le joueur peut se déplacer si le chemin n’est pas bloqué par un mur.

-Méthode `HasReachedGoal` :

Permet de déterminer si le joueur a atteint l'objectif du labyrinthe.

```
bool HasReachedGoal() const {
    return posX == mazeCols - 1 && posY == mazeRows - 1;
}
```

Vérifie si les coordonnées du joueur correspondent à celles de l'objectif (coin inférieur droit du labyrinthe)

-Affichage :

```
void Draw() const {
    DrawTexture(PLAYERIMAGE, posX * cellSize, posY * cellSize, WHITE);
}

void DrawGoal() const {
    int goalX = (mazeCols - 1) * cellSize + cellSize / 2;
    int goalY = (mazeRows - 1) * cellSize + cellSize / 2;
    DrawTexture(goalTexture, goalX - goalTexture.width / 2, goalY - goalTexture.height / 2, WHITE);

}
```

Le joueur et l'objectif sont affichés à l'aide de textures préchargées.

3) Classe game(Logique principale du jeu) :

-Rôle de la classe :

La classe Game gère la logique principale du jeu. Elle contrôle l'état du jeu, suit le temps, et orchestre les interactions entre les joueurs et le labyrinthe.

-Attributs supplémentaires :

- **maze** : Instance de la classe `Maze` pour gérer le labyrinthe actuel.
- **player, player2** : Instances de la classe `Player` représentant les joueurs.
- **scores** : Liste des meilleurs temps enregistrés.
- **gameWon** : Indique si le joueur a gagné.
- **timer** : Chronomètre pour mesurer le temps de jeu.
- **showScores** : Indicateur pour afficher ou masquer le tableau des scores.

-Code principal :

-Réinitialisation et gestion des niveaux :

Réinitialise l'état du jeu en recréant un nouveau labyrinthe, en repositionnant les joueurs, et en réinitialisant les variables de score.

```
void Reset() {
    if (gameWon) {
        scores.push_back(timer); // Save the score
        std::sort(scores.begin(), scores.end());
        PlaySound(bgSound);
    }

    maze.GenerateMaze();
    player = Player();
    player2 = Player(); // Reset player 2
    if (isMultiplayer) {

        player2.SetPosition(0, mazeRows - 1); // Set Player 2 to (0, mazeRows-1)
    }

    gameWon = false;
    winnerSoundPlayed = false; // Reinitialization

    timer = 0;
}
```

Détails :

- Ajoute le temps actuel au tableau des scores si le joueur a gagné.
- Regénère le labyrinthe en appelant `Maze::GenerateMaze`.

-Mise à jour du jeu :

Met à jour les événements du jeu, tels que les mouvements des joueurs, la gestion du chronomètre, et la vérification de la condition de victoire.

```
void Update() {
    if (IsKeyPressed(KEY_R)) Reset();

    if (!gameWon) {
        timer += GetFrameTime();
        player.Update(maze);
        if (isMultiplayer) player2.Update(maze, true); // Update player 2 in multiplayer mode


        if (player.HasReachedGoal() || (isMultiplayer && player2.HasReachedGoal())) {
            gameWon = true;
        }

    }
    UpdateMusicControl();


    if (gameWon && !winnerSoundPlayed) { // Turn on the sound once you win
        PlaySound(winnerSound);
        StopSound(bgSound);
        StopSound(hardSound);
        StopSound(PantherSound);
        winnerSoundPlayed = true;
    }
```

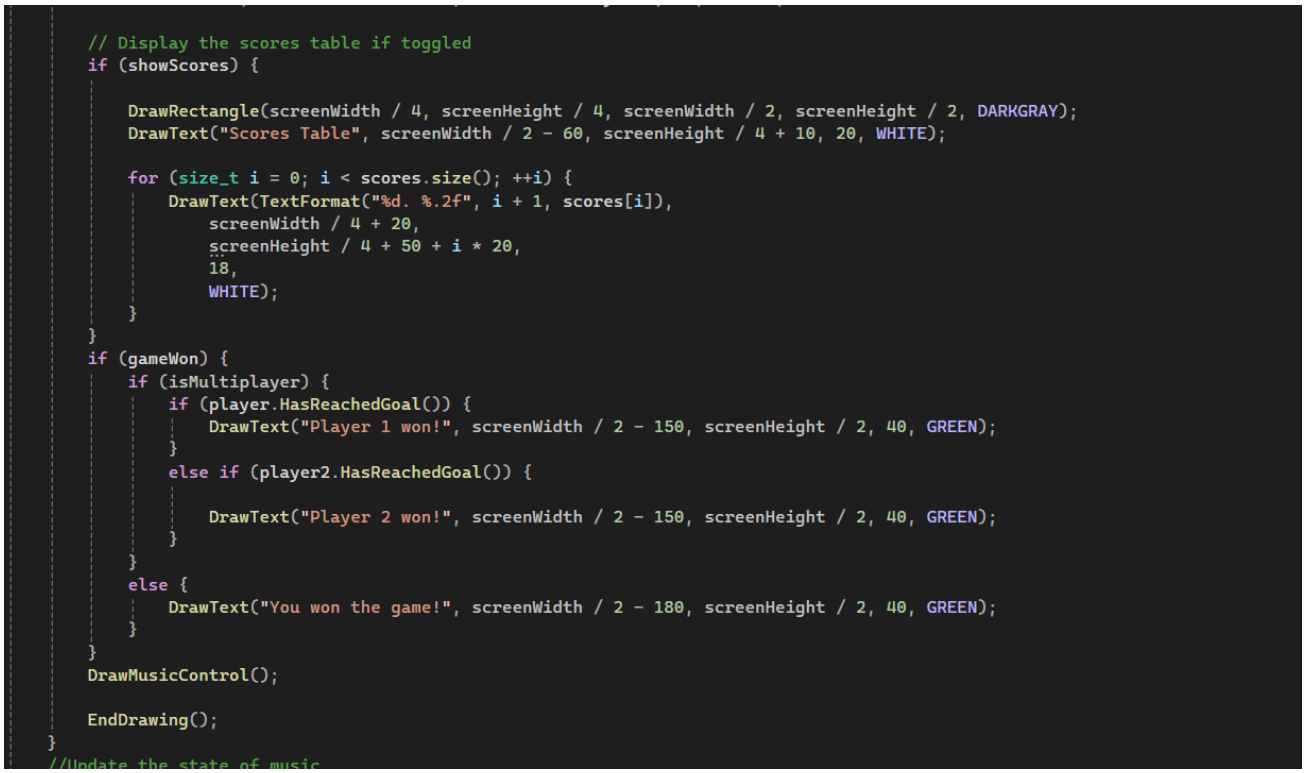
Détails :

- Gère les interactions des joueurs avec le labyrinthe.
- Contrôle les boutons interactifs, tels que "Reset" et "Scores"

-Affichage du jeu :

Dessine les éléments visuels du jeu, y compris le labyrinthe, les joueurs, les scores, et les boutons.

```
}

void Draw() const {
    BeginDrawing();
    ClearBackground(BLACK);

    maze.DrawMaze();
    player.Draw();
    player.DrawGoal();
    if (isMultiplayer) {
        player2.Draw();
        player2.DrawGoal(); // Draw goal for player 2 // Draw goal for both players
    }

    // Display the timer at the bottom of the screen
    DrawText(TextFormat("Time: %.2f", timer), screenWidth / 2 - MeasureText(TextFormat("Time: %.2f", timer), 20) / 2, screenHeight - 30, 20, WHITE);
    // Display the "Scores" button
    Rectangle scoresButton = { screenWidth - 160, screenHeight - 40, 100, 30 };
    DrawRectangleRec(scoresButton, BLUE);

    DrawText("Scores", scoresButton.x + 10, scoresButton.y + 5, 20, WHITE);

    // Display the "RESET" button
    Rectangle resetButton = { screenWidth - 580, screenHeight - 40, 100, 30 };
    DrawRectangleRec(resetButton, BLUE);

    DrawText("Reset", resetButton.x + 10, resetButton.y + 5, 20, WHITE);

    // Display the "RETOUR" button
    Rectangle retourButton = { screenWidth - 300, screenHeight - 40, 100, 30 };
    DrawRectangleRec(retourButton, BLUE);

    DrawText("Back", retourButton.x + 10, retourButton.y + 5, 20, WHITE);
```



```
    // Display the scores table if toggled
    if (showScores) {

        DrawRectangle(screenWidth / 4, screenHeight / 4, screenWidth / 2, screenHeight / 2, DARKGRAY);
        DrawText("Scores Table", screenWidth / 2 - 60, screenHeight / 4 + 10, 20, WHITE);

        for (size_t i = 0; i < scores.size(); ++i) {
            DrawText(TextFormat("%d. %.2f", i + 1, scores[i]),
                screenWidth / 4 + 20,
                screenHeight / 4 + 50 + i * 20,
                18,
                WHITE);
        }
    }
    if (gameWon) {
        if (isMultiplayer) {
            if (player.HasReachedGoal()) {
                DrawText("Player 1 won!", screenWidth / 2 - 150, screenHeight / 2, 40, GREEN);
            }
            else if (player2.HasReachedGoal()) {

                DrawText("Player 2 won!", screenWidth / 2 - 150, screenHeight / 2, 40, GREEN);
            }
        }
        else {
            DrawText("You won the game!", screenWidth / 2 - 180, screenHeight / 2, 40, GREEN);
        }
    }
    DrawMusicControl();

    EndDrawing();
}
//Update the state of music
```

Détails :

- Utilise `Maze::DrawMaze` pour dessiner le labyrinthe.
- Affiche les joueurs à l'aide de `Player::Draw`.
- Affiche un message de victoire lorsque la partie est terminée.

-Gestion des scores :

Les scores sont enregistrés dans un vecteur trié. L'interface utilisateur permet de basculer l'affichage des scores avec un bouton.

4) Classe Level :

-Rôle de la classe :

La classe Level définit les paramètres spécifiques à chaque niveau de difficulté, tels que les dimensions du labyrinthe, la taille des cellules, et la couleur des murs.

-Attributs :

1. **mazeCols** : Nombre de colonnes dans le labyrinthe pour ce niveau.
2. **mazeRows** : Nombre de lignes dans le labyrinthe pour ce niveau.
3. **cellSize** : Taille des cellules du labyrinthe (en pixels).
4. **difficulty** : Niveau de difficulté sélectionné (0 pour facile, 1 pour moyen, 2 pour difficile).
5. **wallColor** : Couleur des murs du labyrinthe, qui change selon la difficulté.

-Configuration des niveaux :

Les niveaux sont divisés en trois catégories : facile, moyen, et difficile. Chaque niveau ajuste dynamiquement les propriétés du labyrinthe.

Cette méthode configure les paramètres du niveau en fonction de la difficulté choisie :

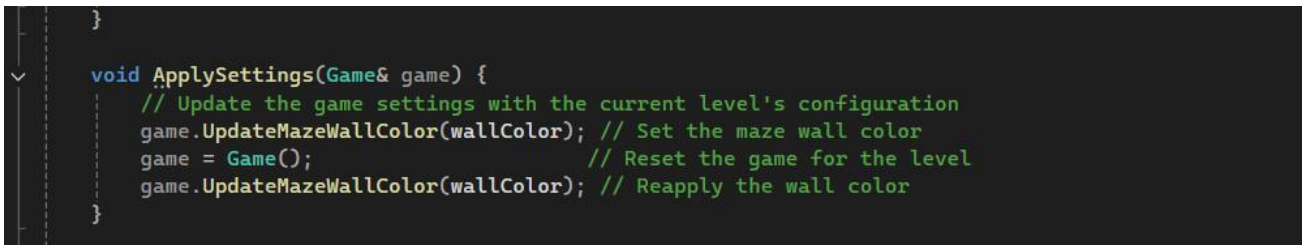
```
//Level class
class Level {
public:
    int mazeCols;
    int mazeRows;
    int cellSize;
    int difficulty;
    Color wallColor;

    Level(int diff) : difficulty(diff), mazeCols(0), mazeRows(0), cellSize(0), wallColor(GREEN) {
        ConfigureLevel();
    }

    void ConfigureLevel() {
        // Set maze dimensions and color based on difficulty
        switch (difficulty) {
        case 0: // Easy
            mazeCols = 12;
            mazeRows = 8;
            cellSize = 67;
            wallColor = GREEN;
            break;
        case 1: // Medium
            mazeCols = 20;
            mazeRows = 13;
            cellSize = 40;
            wallColor = YELLOW;
            break;
        case 2: // Hard
            mazeCols = 25;
            mazeRows = 17;
            cellSize = 32;
            wallColor = RED;
            break;
        }
    }
```

Détails :

- Détermine les dimensions et la taille des cellules du labyrinthe.
- Définit la couleur des murs :
  - **Vert** pour facile.
  - **Jaune** pour moyen.
  - **Rouge** pour difficile.

-Application des paramètres :

Les paramètres du niveau sont appliqués à l'aide de la méthode `ApplySettings`, qui met à jour les attributs de la classe Game, notamment la couleur des murs et la configuration du labyrinthe.



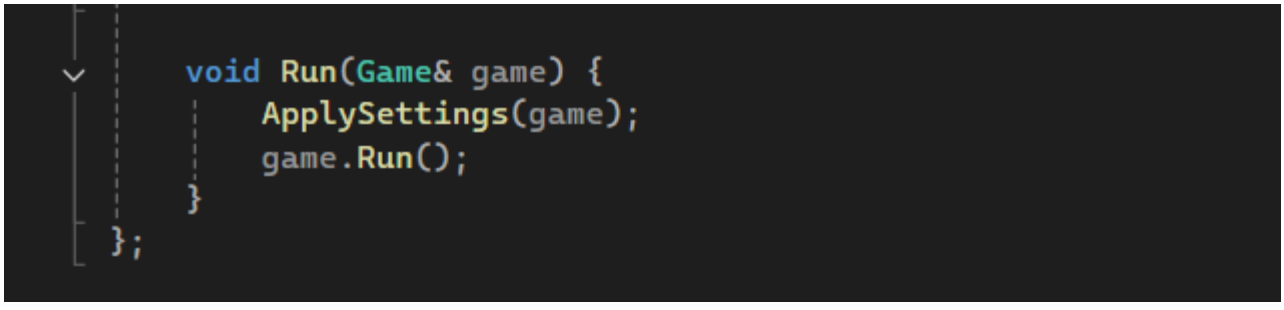
```
}

void ApplySettings(Game& game) {
    // Update the game settings with the current level's configuration
    game.UpdateMazeWallColor(wallColor); // Set the maze wall color
    game = Game();                       // Reset the game for the level
    game.UpdateMazeWallColor(wallColor); // Reapply the wall color
}
```

Détails :

- Met à jour la couleur des murs du labyrinthe dans la classe `Maze` via `Game::UpdateMazeWallColor`.
- Réinitialise la partie en recréant l'objet `Game` avec les nouveaux paramètres

-Méthode `Run` :

Exécute le jeu avec les paramètres configurés.



```
    void Run(Game& game) {
        ApplySettings(game);
        game.Run();
    }
};
```

Détails :

- Appelle `ApplySettings` pour configurer le jeu.
- Lance la boucle principale du jeu avec `Game::Run`

-Interface utilisateur (UI) :

-Gestion des états du jeu :

Le programme utilise l'énumération `GameState` pour naviguer entre plusieurs états, comme le menu principal, la sélection de difficulté, l'aide, et l'écran des développeurs. Chaque état a une interface distincte.

-Exemple : Menu principal :

```
while (!WindowShouldClose()) {
    if (currentState == MENU) {
        // If we are in the menu list
        BeginDrawing();
        DrawTexture(background, 0, 0, RAYWHITE);
        DrawTexture(mazeName, screenWidth / 2 - mazeName.width / 2, 120, WHITE);

        //Drawing play button
        Rectangle playButton = { screenWidth / 2.0f - 140, screenHeight / 2.0f - 45, 344, 74 };
        DrawTexture(startbut, playButton.x, playButton.y, WHITE);
        DrawText("", playButton.x + 35, playButton.y + 10, 50, WHITE);

        if (IsMouseButtonPressed(MOUSE_LEFT_BUTTON) && CheckCollisionPointRec(GetMousePosition(), playButton)) {
            PlaySound(buttonSound);
            currentState = NOMBRE_SELECTION;
        }
        //Help button
        Rectangle helpButton = { screenWidth / 2.0f - 100, screenHeight / 2.0f + 50, 200, 50 };
        DrawTexture(HelpBut, helpButton.x, helpButton.y, WHITE);
        DrawText("", helpButton.x + 50, helpButton.y + 10, 30, RAYWHITE);
        if (IsMouseButtonPressed(MOUSE_LEFT_BUTTON) && CheckCollisionPointRec(GetMousePosition(), helpButton)) {
            PlaySound(buttonSound);
            currentState = HELP;
        }
        //Devs Button
        Rectangle devsButton = { screenWidth / 2.0f - 100, screenHeight / 2.0f + 125, 200, 50 }; // Adjust position and size as needed
        DrawTexture(DevsBut, devsButton.x, devsButton.y, WHITE);
        DrawText("", devsButton.x + 50, devsButton.y + 10, 30, RAYWHITE);

        if (IsMouseButtonPressed(MOUSE_LEFT_BUTTON) && CheckCollisionPointRec(GetMousePosition(), devsButton)) {
            currentState = DEVS;
        }
        // Exit button
        Rectangle exitButton = { screenWidth / 2.0f - 100, screenHeight / 2.0f + 200, 200, 50 };
        DrawTexture(ExitBut, exitButton.x, exitButton.y, WHITE);

            DrawText("", exitButton.x + 50, exitButton.y + 10, 30, RAYWHITE);
            if (IsMouseButtonPressed(MOUSE_LEFT_BUTTON) && CheckCollisionPointRec(GetMousePosition(), exitButton)) {
                PlaySound(buttonSound);
                CloseWindow(); // Exit the game
                break;
            }

            EndDrawing();
        }
```

-Boutons interactifs :

Chaque écran contient des boutons interactifs, tels que les boutons "Scores", "Reset", et "Back". Les clics sur ces boutons déclenchent des actions spécifiques, comme le retour au menu principal ou la réinitialisation du jeu.

Fonctionnalités audio et graphiques :

-Effets sonores :

Le programme utilise plusieurs sons pour enrichir l'expérience utilisateur :

- **buttonSound** : Son joué lors des clics sur les boutons.
- **bgSound** : Musique de fond.
- **winnerSound** : Son joué lors de la victoire.

-Exemple d'implémentation :

```
//Update the state of music
void UpdateMusicControl() {
    Rectangle musicButton = { screenWidth - 760, 560, 145, 30 };
    if (IsMouseButtonPressed(MOUSE_LEFT_BUTTON) && CheckCollisionPointRec(GetMousePosition(), musicButton)) {
        if (isMusicPlaying) {
            PauseSound(bgSound);// Temporarily stop music
            PauseSound(PantherSound);
            PauseSound(hardSound);
        }
        else {
            ResumeSound(buttonSound);// Resume
        }
        isMusicPlaying = !isMusicPlaying; // Reverse the situation
        if (isMusicPlaying) {
            if (!IsSoundPlaying(bgSound)) ResumeSound(bgSound);
            if (!IsSoundPlaying(PantherSound)) ResumeSound(PantherSound);
            if (!IsSoundPlaying(hardSound)) ResumeSound(hardSound);

        }
    }
}
```

-Textures et images :

Les textures sont utilisées pour les joueurs, le but, et les arrière-plans. Elles sont chargées au début de l'exécution et libérées à la fin.

-Mode multijoueur :

Le mode multijoueur permet à deux joueurs de naviguer dans le labyrinthe en simultané. Les commandes sont divisées comme suit :

- Joueur 1 : Touches fléchées.

- Joueur 2 : Touches **W/A/S/D**.

Conclusion :

Ce projet de jeu de labyrinthe illustre l'utilisation efficace de C++ avec la bibliothèque Raylib pour créer une application interactive et visuellement attrayante. Les fonctionnalités telles que les niveaux, l'interface utilisateur intuitive, les effets sonores immersifs, et le mode multijoueur rendent ce projet complet et adapté à un large public.