université abdelmalek essaâdi Faculté des Sciences et Techniques de Tanger

Développement d'un jeu de casse-tête type labyrinthe en C++ et Raylib

Réalisé par :

Oussama Benaaros

Roumaissa Sbeih

Ayoub Lahlaibi

Chaimaa Maziane

Encadré par :

Pr Ikram Ben abdel ouahab

Introduction :

Ce rapport présente la conception et le développement d'un jeu de casse-tête de type labyrinthe en C++ utilisant la bibliothèque graphique Raylib. Le jeu proposera une expérience immersive où le joueur devra naviguer dans un labyrinthe généré aléatoirement à chaque nouvelle partie. Il comprendra trois niveaux de difficulté, chacun affectant la taille du labyrinthe et la complexité de la navigation.

Que veut dire Raylib :

Raylib est une bibliothèque d'interface de programmation graphique simple, gratuite et multiplateforme, cherchant à réduire les obstacles à la conception de jeux vidéo 2D et 3D. Elle est particulièrement populaire en raison de son minimalisme, ce qui en fait un excellent choix pour les personnes qui commencent tout juste dans le développement de jeux ainsi que pour les développeurs qui ont besoin de prototyper rapidement des idées.

Objectifs du projet :

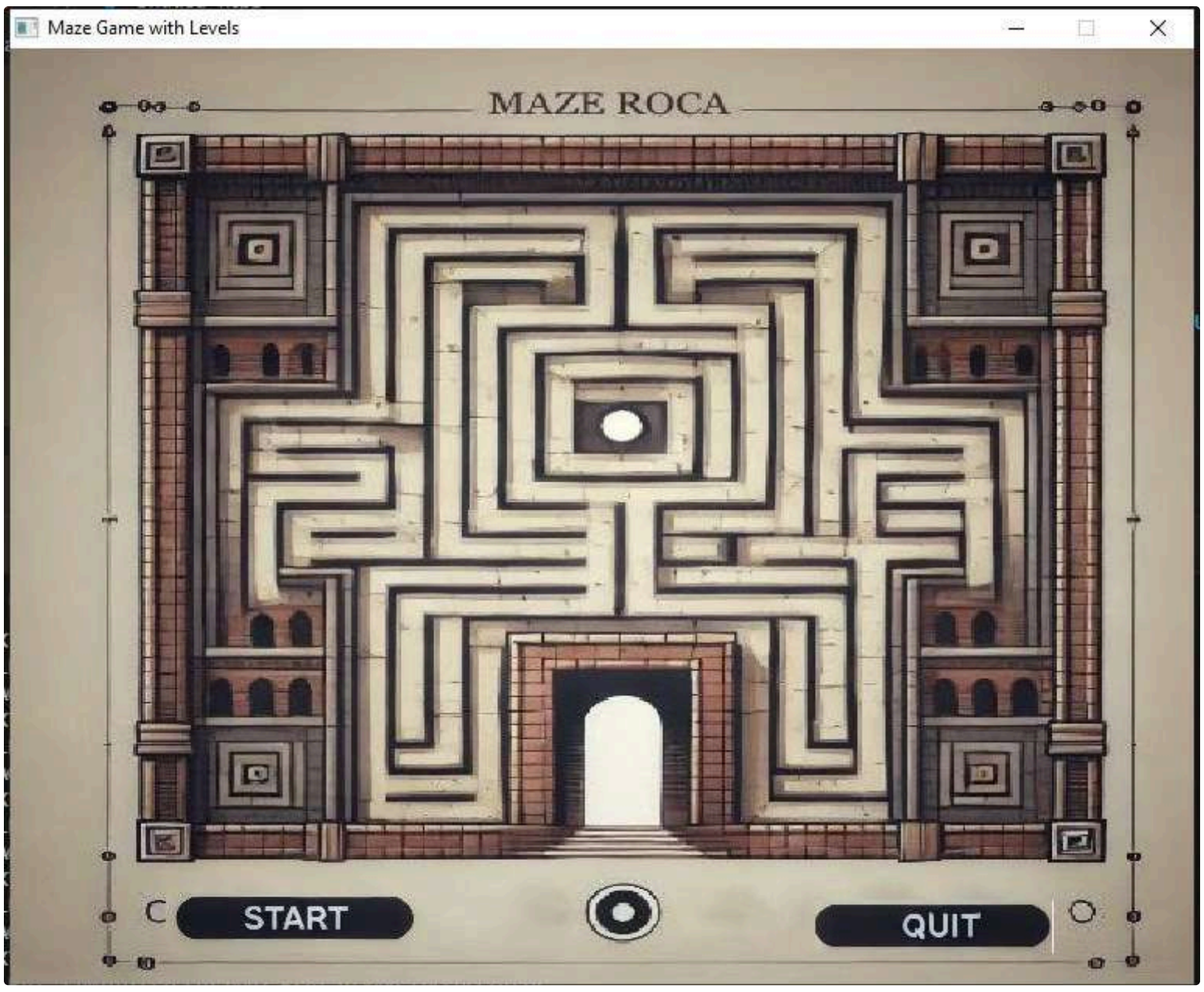
L'objectif principal de ce projet est de créer un jeu de casse-tête de type labyrinthe en C++ qui offre une expérience de jeu engageante et stimulante. Le jeu sera développé en utilisant la bibliothèque graphique Raylib, ce qui permettra une interface utilisateur intuitive et des effets visuels attrayants. Le projet vise également à illustrer les principes de la programmation orientée objet (POO) en C++.

Utilisation de la programmation orientée objet :

La programmation orientée objet (POO) sera au cœur de la conception du jeu. Les éléments du labyrinthe, tels que les murs, les passages et le joueur, seront représentés par des classes distinctes. Chaque classe aura ses propres attributs (propriétés) et méthodes (actions), ce qui permettra une gestion efficace et flexible du jeu.

Par exemple, la classe "Mur" pourrait avoir des attributs pour sa position, sa taille et sa couleur, et des méthodes pour afficher le mur à l'écran. La classe "Joueur" pourrait avoir des attributs pour sa position, sa vitesse et sa direction, et des méthodes pour déplacer le joueur dans le labyrinthe.

Interface Graphique :



Code :

```
BeginDrawing();
        ClearBackground(RAYWHITE);

        if (gameState == MENU) {
            DrawTexturePro(background, sourceRect, destRect, Vector2{ 0, 0 }, 0.0f, WHITE);
            DrawTexture(buttonStart, (int)buttonStartRect.x, (int)buttonStartRect.y, WHITE);
            DrawTexture(buttonExit, (int)buttonExitRect.x, (int)buttonExitRect.y, WHITE);

            if (CheckCollisionPointRec(GetMousePosition(), buttonStartRect) && IsMouseButtonPressed(
MOUSE_BUTTON_LEFT)) {
                gameState = LEVEL_SELECTION;
            }
            if (CheckCollisionPointRec(GetMousePosition(), buttonExitRect) && IsMouseButtonPressed(
MOUSE_BUTTON_LEFT)) {
                gameState = EXIT;
            }
```

Contexte général :

- `BeginDrawing()` : Démarre le processus de dessin pour la frame actuelle. Toutes les opérations graphiques après cet appel seront dessinées à l'écran.
- `ClearBackground(RAYWHITE)` : Efface l'écran en appliquant une couleur de fond, ici **RAYWHITE** (un blanc légèrement bleuté).

Cette section initialise une nouvelle frame et prépare un fond propre avant de dessiner quoi que ce soit.

Gestion de l'état `MENU` :

Le code gère spécifiquement le cas où l'état du jeu est `MENU` :

1. **Affichage du fond et des boutons :**

Dessine le **fond du menu** avec :

- `background` : Texture (image) de fond.
- `sourceRect` : Partie de l'image source à dessiner.
- `destRect` : Où et comment la texture est affichée à l'écran.
- `Vector2{ 0, 0 }` : Point d'origine pour l'affichage (ici, aucun décalage).
- `0.0f` : Aucune rotation.
- `WHITE` : Couleur appliquée (pas de teinte).
- Dessine deux boutons (images) :
  - `buttonStart` et `buttonExit` : Les textures des boutons "Start" (Démarrer) et "Exit" (Quitter).
  - Les positions sont extraites des rectangles `buttonStartRect` et `buttonExitRect`, converties en entiers.

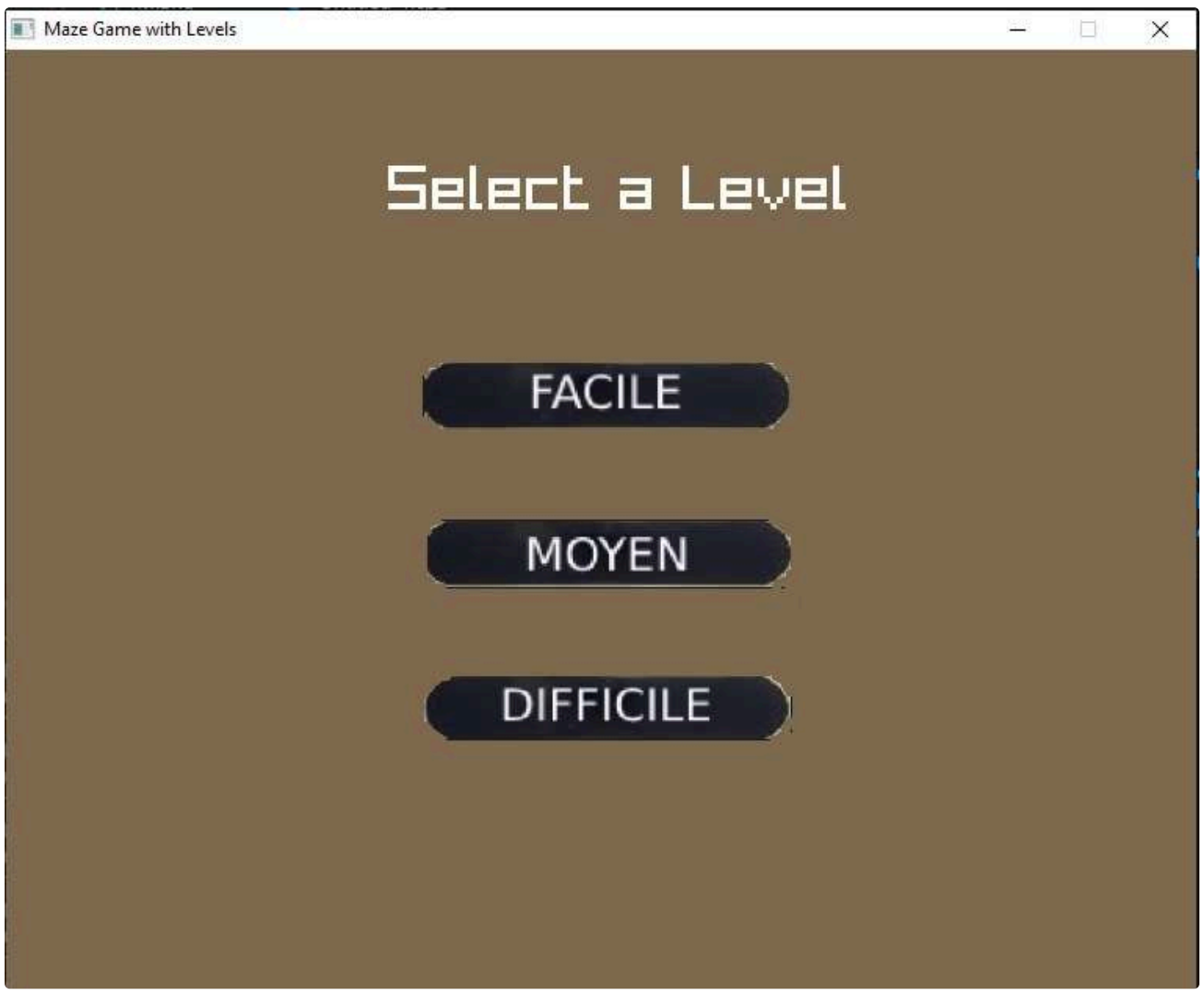
Ces trois appels définissent l'apparence du menu principal.

2. **Interactions avec les boutons :**

- **Vérifie une collision entre la souris et le bouton "Start"** :
- `CheckCollisionPointRec` : Teste si la position actuelle de la souris (`GetMousePosition()`) se trouve à l'intérieur du rectangle `buttonStartRect`.
- `IsMouseButtonPressed(MOUSE_BUTTON_LEFT)` : Vérifie si le bouton gauche de la souris est cliqué.
- **Action** : Si la souris clique sur "Start", l'état du jeu passe à `LEVEL_SELECTION` (l'écran de sélection de niveau).

- **Vérifie une collision entre la souris et le bouton "Exit"** :
- Même logique que pour "Start", mais avec le rectangle `buttonExitRect`.
- **Action** : Si la souris clique sur "Exit", l'état du jeu passe à `EXIT` (préparer la sortie du jeu).

Sélection de niveaux d'un jeu :



Code :

```
ClearBackground(BROWN);
            DrawText("Select a Level", windowWidth / 2.5 - 60, 70, 40, WHITE);

            Rectangle buttonEasyRect = { (screenWidth - buttonEasy.width) / 2.0f, 200.0f, (float)buttonEasy.width, (float)buttonEasy.height };
            Rectangle buttonMediumRect = { (screenWidth - buttonMedium.width) / 2.0f, 300.0f, (float)buttonMedium.width, (float)buttonMedium.height };
            Rectangle buttonHardRect = { (screenWidth - buttonHard.width) / 2.0f, 400.0f, (float)buttonHard.width, (float)buttonHard.height };

            DrawTexture(buttonEasy, (int)buttonEasyRect.x, (int)buttonEasyRect.y, WHITE);
            DrawTexture(buttonMedium, (int)buttonMediumRect.x, (int)buttonMediumRect.y, WHITE);
            DrawTexture(buttonHard, (int)buttonHardRect.x, (int)buttonHardRect.y, WHITE);
```

Contexte général :

1. **Objectif :**
    - Afficher un écran de sélection de niveaux.
    - Proposer trois niveaux de difficulté : **Facile**, **Moyen**, et **Difficile**.
    - Organiser graphiquement les boutons correspondants.
2. **Utilisation des rectangles :**
    - Les rectangles définissent les zones des boutons pour les aligner correctement à l'écran et pour gérer les interactions.

Explication du code

1. Effacement et configuration du fond

- `ClearBackground(BROWN)` : Remplit tout l'écran avec une couleur de fond marron (**BROWN**), effaçant ce qui était affiché dans la frame précédente.

2. Affichage du titre

- DrawText : Affiche un texte sur l'écran.
- "Select a Level" : Texte du titre.
- windowWidth / 2.5 - 60 : Coordonnée X pour centrer approximativement le texte (ajustement manuel avec -60).
- 70 : Coordonnée Y pour positionner le texte en haut de l'écran.
- 40 : Taille de la police. WHITE : Couleur du texte.

3. Définition des rectangles des boutons

Les boutons sont alignés verticalement au centre de l'écran.

- `buttonEasyRect` :
    - Définit la position et la taille du bouton **"Facile"**.
    - `(screenWidth - buttonEasy.width) / 2.0f` : Position X pour centrer horizontalement le bouton (calculé en soustrayant la largeur du bouton à celle de l'écran, puis en divisant par deux).
    - `200.0f` : Position Y pour placer le bouton à une hauteur fixe.
    - `buttonEasy.width` **et** `buttonEasy.height` : Dimensions du bouton.

Des rectangles similaires sont définis pour les autres boutons :

`300.0f` et `400.0f` : Espacement vertical entre les boutons.

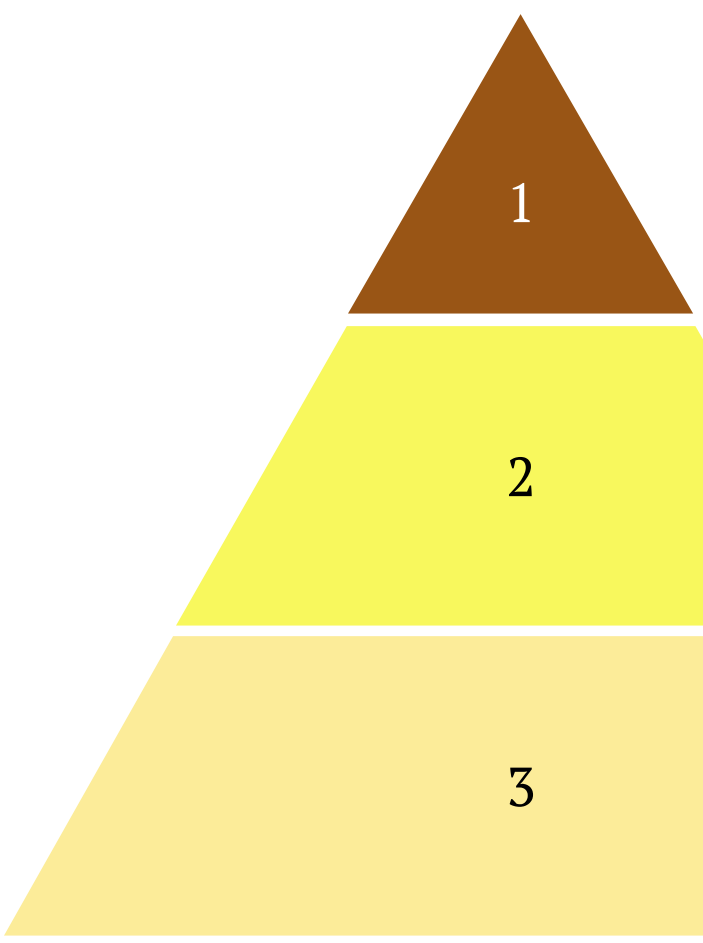
4. Affichage des boutons

`DrawTexture` : Affiche chaque bouton comme une image.

- `buttonEasy`, `buttonMedium`, `buttonHard` : Textures correspondant aux niveaux de difficulté.
- `buttonEasyRect.x` **et** `buttonEasyRect.y` : Coordonnées (X, Y) où le bouton est affiché (converties en entiers pour correspondre à la fonction).
- `WHITE` : Couleur appliquée aux textures (aucune modification ici).

Organisation visuelle résultante

1. **Fond** : L'écran est entièrement marron (**BROWN**).
2. **Titre** : Texte **"Select a Level"** affiché en haut, centré horizontalement.
3. **Boutons** :
    - Trois boutons alignés verticalement (Facile, Moyen, Difficile), centrés horizontalement.
    - Espacement fixe entre les boutons (100 pixels).



Difficile

Le niveau difficile met les joueurs à l'épreuve avec des ennemis rapides et agressifs

Moyen

Le niveau moyen propose un défi équilibré, combinant des mécaniques plus complexes .

Facile

Le niveau facile offre une expérience accessible et détendue, avec des défis simplifiés,

Level :

Code :

```
if (CheckCollisionPointRec(GetMousePosition(), buttonEasyRect) && IsMouseButtonPressed(MOUSE_BUTTON_LEFT)) {
                WIDTH = 25;
                HEIGHT = 25;
                initMaze();
                generateMaze(0, 1);
                gameState = PLAYING;
            }
            if (CheckCollisionPointRec(GetMousePosition(), buttonMediumRect) && IsMouseButtonPressed(MOUSE_BUTTON_LEFT)) {
                WIDTH = 25;
                HEIGHT = 25;
                initMaze();
                generateMaze(3, 4);
                gameState = PLAYING;
            }
            if (CheckCollisionPointRec(GetMousePosition(), buttonHardRect) && IsMouseButtonPressed(MOUSE_BUTTON_LEFT)) {
                WIDTH = 25;
                HEIGHT = 28;
                initMaze();
                generateMaze(7, 3);
                gameState = PLAYING;
            }
```

1. Détection des clics sur les boutons

Chaque bloc commence par vérifier si le joueur a cliqué sur un bouton de niveau :

- `CheckCollisionPointRec(GetMousePosition(), buttonEasyRect)` :
  - Vérifie si la position actuelle de la souris (obtenue avec `GetMousePosition()`) se trouve à l'intérieur du rectangle du bouton correspondant (ici, `buttonEasyRect` pour le niveau facile).
- `IsMouseButtonPressed(MOUSE_BUTTON_LEFT)` :
  - Vérifie si le bouton gauche de la souris est cliqué.

Si ces deux conditions sont vraies, le code exécute des actions spécifiques pour ce niveau.

2. Configuration des paramètres du labyrinthe

Chaque niveau configure la largeur et la hauteur du labyrinthe, ainsi que la génération du labyrinthe en fonction de la difficulté :

Niveau Facile

- `WIDTH` **et** `HEIGHT` :
  - Dimensions fixes du labyrinthe : 25x25 cases (standard pour ce jeu).
- `initMaze()` :
  - Initialise la structure ou les données du labyrinthe (par exemple, une matrice vide ou les bases nécessaires à sa génération).
- `generateMaze(0, 1)` :
  - Génère un labyrinthe en utilisant des paramètres spécifiques à la difficulté facile :
    - `0` **et** `1` : Ces arguments pourraient indiquer :
      - Le nombre de murs ou de chemins bloqués.
      - La densité des obstacles ou ennemis.
      - La position de départ ou la complexité générale.
- `gameState = PLAYING` :
  - Change l'état du jeu à `PLAYING`, ce qui commence la partie avec le labyrinthe généré.

Niveau Moyen

- Dimensions identiques (25x25), mais les paramètres de génération changent :
  - `generateMaze(3, 4)` :
    - Des obstacles plus nombreux ou des chemins plus complexes (par exemple, 3 obstacles majeurs et 4 chemins plus difficiles).

Niveau Difficile

- Le niveau difficile modifie légèrement la taille du labyrinthe :
- `WIDTH = 25` **et** `HEIGHT = 28` : Augmentation de la hauteur pour ajouter de la complexité verticale.
- `generateMaze(7, 3)` :
- Génère un labyrinthe encore plus complexe :
  - **7 obstacles principaux**.
  - **3 chemins critiques**, mais probablement très denses ou étroits.

3. Transition vers l'état de jeu actif

Après la configuration, le jeu passe en mode **PLAYING** :

- `gameState = PLAYING` :
  - Change l'état global du jeu pour indiquer que le joueur est maintenant en train de jouer avec le labyrinthe configuré.

Gère les déplacements du joueur :

Code :

```
else if (gameState == PLAYING) {
             if (!win && !gameOver) {
            if (IsKeyPressed(KEY_UP) && maze[playerX][playerY - 1] == 0) {
                playerY--;
                if (!timerStarted) {
                    timerStarted = true;
                    startTime = GetTime();
                }
                score += 10;
            }
            if (IsKeyPressed(KEY_DOWN) && maze[playerX][playerY + 1] == 0) {
                playerY++;
                if (!timerStarted) {
                    timerStarted = true;
                    startTime = GetTime();
                }
                score += 10;
            }
            if (IsKeyPressed(KEY_LEFT) && maze[playerX - 1][playerY] == 0) {
                playerX--;
                if (!timerStarted) {
                    timerStarted = true;
                    startTime = GetTime();
                }
                score += 10;
            }
            if (IsKeyPressed(KEY_RIGHT) && maze[playerX + 1][playerY] == 0) {
                playerX++;
                if (!timerStarted) {
                    timerStarted = true;
                    startTime = GetTime();
                }
                score += 10;
            }
        }
```

Contexte général :

- **Objectif principal** : Permettre au joueur de se déplacer dans le labyrinthe à l'aide des touches directionnelles (haut, bas, gauche, droite).
- **Conditions de déplacement** :
- Le joueur peut se déplacer uniquement si la case suivante dans le labyrinthe est accessible (valeur 0 dans la matrice `maze`).
- **Autres fonctionnalités** :
- Démarrage du chronomètre lorsqu'un premier mouvement est effectué.
- Mise à jour du score pour chaque mouvement.

Explication ligne par ligne

1. **Vérification de l'état du jeu**

- `gameState == PLAYING` :
- Assure que le joueur est dans l'état de jeu actif.
- `!win && !gameOver` :
- Le joueur peut se déplacer uniquement si la partie n'est ni gagnée (`win`) ni perdue (`gameOver`).

2. **Déplacements dans le labyrinthe**

Déplacement vers le haut

- `IsKeyPressed(KEY_UP)` :
  - Vérifie si la touche fléchée **"haut"** a été pressée.
- `maze[playerX][playerY - 1] == 0` :
  - Vérifie que la case située **au-dessus** du joueur est accessible (**0** dans la matrice `maze` signifie une case vide ou traversable).
- `playerY--` :
  - Déplace le joueur vers le haut en diminuant sa position Y dans la matrice.

Déplacement vers le bas

- Identique au déplacement vers le haut, sauf que la position Y augmente pour descendre.

Déplacement vers la gauche

- Vérifie si la case à gauche est accessible et déplace le joueur vers la gauche en diminuant sa position X.

Déplacement vers la droite

Vérifie si la case à droite est accessible et déplace le joueur en augmentant sa position X.

3. **Démarrage du chronomètre**

- `!timerStarted` :
- Vérifie si le chronomètre n’a pas encore démarré (lors du tout premier mouvement du joueur).
- `timerStarted = true` :
- Indique que le chronomètre a été activé.

- `startTime = GetTime()` :
- Enregistre le temps actuel (en secondes depuis le début du programme) pour suivre la durée de la partie.

4. **Mise à jour du score**

Chaque déplacement ajoute **10 points** au score du joueur.

Gestion de temps et score :

Code :

```
float currentTime = timerStarted && !win ? GetTime() - startTime : startTime;
        DrawText(TextFormat("Time: %.2f s", currentTime), WIDTH * CELL_SIZE + 20, 20, 20, BLACK);

        DrawText(TextFormat("Score: %d", score), WIDTH * CELL_SIZE + 20, 60, 20, BLACK);

        DrawRectangleRec(restartButton, BROWN);
        DrawText("Restart", restartButton.x + 10, restartButton.y + 10, 20, WHITE);

        DrawRectangleRec(exitButton, BROWN);
        DrawText("Quit", exitButton.x + 20, exitButton.y + 10, 20, WHITE);
```

1. Calcul du temps actuel

- `timerStarted && !win` :
- Cette condition vérifie que :
  a. Le chronomètre a commencé (`timerStarted` est vrai).
  b. La partie n'est pas encore gagnée (`!win`).
- `GetTime() - startTime` :
- Si les conditions ci-dessus sont remplies, le temps écoulé depuis le début de la partie est calculé.
- `startTime` :
- Si le chronomètre n'a pas commencé ou que la partie est gagnée, la valeur de `startTime` est affichée.

2. Affichage du temps écoulé

`TextFormat("Time: %.2f s", currentTime)` :

- Formate une chaîne de texte pour afficher le temps en secondes avec deux décimales (`%.2f`).

`DrawText(...)` :

- Affiche le texte formatté à l'écran.

Position :

- `WIDTH * CELL_SIZE + 20` : Position en X, à droite du labyrinthe.
- `20` : Position en Y, en haut de l'écran.
- `20` : Taille du texte
- `BLACK` : Couleur du texte (noir).

3. Affichage du score

`TextFormat("Score: %d", score)` :

- Formate une chaîne de texte pour afficher le score sous forme d'entier (`%d`).
- `DrawText(...)` :

  Affiche le score à l'écran.

Position :

- `WIDTH * CELL_SIZE + 20` : Position en X, à droite du labyrinthe (même alignement que le temps).
- `60` : Position en Y, en dessous de l'affichage du temps.
- `20` : Taille du texte.
- `BLACK` : Couleur du texte (noir).

4. Bouton "Restart" (Redémarrer)

`DrawRectangleRec(restartButton, BROWN)` :

- Dessine un rectangle représentant le bouton de redémarrage.
- `restartButton` :
  - Une structure de type `Rectangle` contenant les coordonnées (x, y) et les dimensions (largeur et hauteur) du bouton.
- `BROWN` : Couleur de remplissage du rectangle (marron).
- `DrawText(...)` :
- Ajoute le texte **"Restart"** à l'intérieur du bouton.
- **Position** :
  - `restartButton.x + 10` et `restartButton.y + 10` : Position légèrement décalée par rapport au coin supérieur gauche du bouton.
  - `20` : Taille du texte.
  - `WHITE` : Couleur du texte (blanc).

5. Bouton "Quit" (Quitter)

- Identique au bouton "Restart", sauf :
  - **Texte** : "Quit".
  - **Position du texte** :
    - `exitButton.x + 20` : Décalage horizontal légèrement différent pour centrer le texte "Quit".
    - `exitButton.y + 10` : Décalage vertical constant.

le joueur a gagné :

Code :

```
if (win) {
    DrawRectangle(0, 0, windowWidth, windowHeight, BROWN);

    // Texte "You Win" au centre
    const char* winText = "Congratulations, You Win!";
    int winTextWidth = MeasureText(winText, 40);
    int winTextX = (windowWidth - winTextWidth) / 2;
    int winTextY = (windowHeight / 2) - 100; // Plus haut que le centre
    DrawText(winText, winTextX, winTextY, 40, WHITE);

    // Texte du temps final au centre
    const char* finalTimeText = TextFormat("Final Time: %.2f s", currentTime);
    int finalTimeWidth = MeasureText(finalTimeText, 20);
    int finalTimeX = (windowWidth - finalTimeWidth) / 2;
    int finalTimeY = (windowHeight / 2) - 40; // Position centrale
    DrawText(finalTimeText, finalTimeX, finalTimeY, 20, BLACK);

    // Texte du score final au centre
    const char* finalScoreText = TextFormat("Final Score: %d", score);
    int finalScoreWidth = MeasureText(finalScoreText, 20);
    int finalScoreX = (windowWidth - finalScoreWidth) / 2;
    int finalScoreY = finalTimeY + 40; // Juste en dessous du temps final
    DrawText(finalScoreText, finalScoreX, finalScoreY, 20, BLACK);

    // Bouton "Play Again" centré
    Rectangle playAgainButton = { windowWidth / 2 - 60, finalScoreY + 60, 120, 40 };
    Rectangle exitWinButton = { windowWidth / 2 - 60, playAgainButton.y + 60, 120, 40 };

    DrawRectangleRec(playAgainButton, WHITE);
    DrawText("Play Again", playAgainButton.x + 15, playAgainButton.y + 10, 20, BLACK);
// Centré par rapport au bouton

    DrawRectangleRec(exitWinButton, WHITE);
    DrawText("Exit", exitWinButton.x + 35, exitWinButton.y + 10, 20, BLACK);
// Centré par rapport au bouton

    // Détection de clic sur les boutons
    if (CheckCollisionPointRec(GetMousePosition(), playAgainButton) && IsMouseButtonPressed(
MOUSE_BUTTON_LEFT)) {
        main(); // Redémarrer le jeu
    }
    if (CheckCollisionPointRec(GetMousePosition(), exitWinButton) && IsMouseButtonPressed(
MOUSE_BUTTON_LEFT)) {
        CloseWindow(); // Quitter le jeu
    }
```

1. Fond de l'écran de victoire

`DrawRectangle(0, 0, windowWidth, windowHeight, BROWN)` :

- Cette fonction dessine un rectangle qui couvre tout l'écran de jeu, avec une couleur de fond **marron** (**BROWN**). Cela permet de changer l'apparence de l'écran et de créer une zone dédiée à l'affichage de la victoire.

2. Texte "You Win" (Félicitations)

- `const char* winText = "Congratulations, You Win!"` : Définit le message affiché lorsqu'un joueur gagne.
- `MeasureText(winText, 40)` : Calcule la largeur du texte à afficher avec une taille de police de 40 pixels.
- **Positionnement du texte** :
- `winTextX = (windowWidth - winTextWidth) / 2` : Centre le texte horizontalement.
- `winTextY = (windowHeight / 2) - 100` : Positionne le texte un peu au-dessus du centre vertical de la fenêtre.
- `DrawText(winText, winTextX, winTextY, 40, WHITE)` : Affiche le message **"Congratulations, You Win!"** en blanc (**WHITE**) à la position calculée.

3. Affichage du temps final

- `const char* finalTimeText = TextFormat("Final Time: %.2f s", currentTime)` : Formate le texte pour afficher le temps final, en secondes, avec deux décimales.
- `MeasureText(finalTimeText, 20)` : Calcule la largeur du texte avec une taille de police de 20 pixels.
- **Positionnement du texte** :
- `finalTimeX = (windowWidth - finalTimeWidth) / 2` : Centre le texte horizontalement.
- `finalTimeY = (windowHeight / 2) - 40` : Positionne le texte juste au centre verticalement, mais légèrement plus bas que le texte de victoire.
- `DrawText(finalTimeText, finalTimeX, finalTimeY, 20, BLACK)` : Affiche le temps final en noir (**BLACK**) à la position calculée.

4. Affichage du score final

- `const char* finalScoreText = TextFormat("Final Score: %d", score)` : Formate le texte pour afficher le score final en utilisant un entier.
- `MeasureText(finalScoreText, 20)` : Calcule la largeur du texte avec une taille de police de 20 pixels.
- **Positionnement du texte** :
  - `finalScoreX = (windowWidth - finalScoreWidth) / 2` : Centre le texte horizontalement.
  - `finalScoreY = finalTimeY + 40` : Place le texte juste en dessous du texte du temps final.
- `DrawText(finalScoreText, finalScoreX, finalScoreY, 20, BLACK)` : Affiche le score final en noir (**BLACK**) à la position calculée.

5. Bouton "Play Again" (Jouer à nouveau)

- `Rectangle playAgainButton = { windowWidth / 2 - 60, finalScoreY + 60, 120, 40 }` :
- Crée un rectangle pour le bouton **"Play Again"** centré horizontalement avec une largeur de 120 pixels et une hauteur de 40 pixels.
- `DrawRectangleRec(playAgainButton, WHITE)` : Dessine le rectangle du bouton en blanc (**WHITE**).
- `DrawText("Play Again", playAgainButton.x + 15, playAgainButton.y + 10, 20, BLACK)` :
- Affiche le texte **"Play Again"** au centre du bouton, avec un décalage de 15 pixels en X et 10 pixels en Y pour le centrer dans le rectangle.

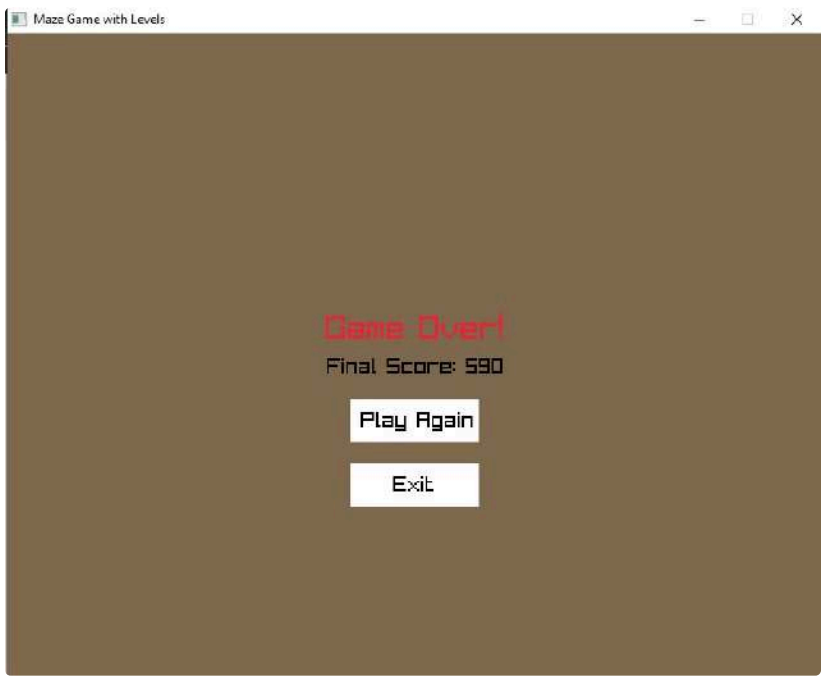
6. Bouton "Exit" (Quitter)

- `Rectangle exitWinButton = { windowWidth / 2 - 60, playAgainButton.y + 60, 120, 40 }` :
- Crée un rectangle pour le bouton **"Exit"**, juste en dessous du bouton "Play Again".
- `DrawRectangleRec(exitWinButton, WHITE)` : Dessine le rectangle du bouton en blanc (**WHITE**).
- `DrawText("Exit", exitWinButton.x + 35, exitWinButton.y + 10, 20, BLACK)` :
- Affiche le texte **"Exit"** au centre du bouton, avec un décalage de 35 pixels en X et 10 pixels en Y pour le centrer dans le rectangle.

7. Détection des clics sur les boutons

- `CheckCollisionPointRec(GetMousePosition(), playAgainButton)` : Vérifie si la position actuelle de la souris est à l'intérieur du bouton "Play Again".
- `IsMouseButtonPressed(MOUSE_BUTTON_LEFT)` : Vérifie si le bouton gauche de la souris a été cliqué.
- Si le joueur clique sur le bouton **"Play Again"**, la fonction `main()` est appelée pour redémarrer le jeu.
- Si le joueur clique sur le bouton **"Exit"**, la fenêtre du jeu est fermée avec `CloseWindow()`.

Joueur perd :



Code :

```c
else if (gameOver) {
DrawRectangle(0, 0, windowWidth, windowHeight, BROWN);

// Texte "Game Over!" centré
const char *gameOverText = "Game Over!";
int gameOverTextWidth = MeasureText(gameOverText, 40);
int gameOverTextX = (windowWidth - gameOverTextWidth) / 2;
int gameOverTextY = (windowHeight / 2) - 80; // Légèrement au-dessus du centre
DrawText(gameOverText, gameOverTextX, gameOverTextY, 40, RED);

// Texte "Final Score" centré
const char *finalScoreText = TextFormat("Final Score: %d", score);
int finalScoreWidth = MeasureText(finalScoreText, 20);
int finalScoreX = (windowWidth - finalScoreWidth) / 2;
int finalScoreY = (windowHeight / 2) - 40; // Juste en dessous du "Game Over"
DrawText(finalScoreText, finalScoreX, finalScoreY, 20, BLACK);

// Bouton "Play Again" centré
Rectangle restartGameButton = { windowWidth / 2 - 60, (windowHeight / 2) + 20, 120, 40 };
DrawRectangleRec(restartGameButton, WHITE);
DrawText("Play Again", restartGameButton.x + 20, restartGameButton.y + 10, 20, BLACK);

// Bouton "Exit" centré
Rectangle exitGameButton = { windowWidth / 2 - 60, restartGameButton.y + 60, 120, 40 };
DrawRectangleRec(exitGameButton, WHITE);
DrawText("Exit", exitGameButton.x + 35, exitGameButton.y + 10, 20, BLACK);
```

1. Fond de l'écran "Game Over"

`DrawRectangle(0, 0, windowWidth, windowHeight, BROWN)` :

- Cette ligne dessine un rectangle qui couvre toute la fenêtre du jeu avec une couleur de fond **marron** (**BROWN**). Cela crée une zone distincte pour l'écran de fin de jeu, marquant la fin de la partie.

2. Texte "Game Over" (Perdu)

- `const char *gameOverText = "Game Over!"` : Définit le texte à afficher pour indiquer que la partie est terminée et que le joueur a perdu.
- `MeasureText(gameOverText, 40)` : Calcule la largeur du texte **"Game Over!"** avec une taille de police de 40 pixels.
- **Positionnement du texte** :
- `gameOverTextX = (windowWidth - gameOverTextWidth) / 2` : Calcule la position horizontale pour centrer le texte dans la fenêtre.
- `gameOverTextY = (windowHeight / 2) - 80` : Positionne le texte légèrement au-dessus du centre vertical de la fenêtre.
- `DrawText(gameOverText, gameOverTextX, gameOverTextY, 40, RED)` : Affiche le texte **"Game Over!"** en rouge (**RED**) à la position calculée.

3. Affichage du score final

- `const char *finalScoreText = TextFormat("Final Score: %d", score)` : Formate le texte pour afficher le score final du joueur. La variable `score` est l'entier représentant le score du joueur.
- `MeasureText(finalScoreText, 20)` : Calcule la largeur du texte avec une taille de police de 20 pixels.
- **Positionnement du texte** :
- `finalScoreX = (windowWidth - finalScoreWidth) / 2` : Calcule la position horizontale pour centrer le texte dans la fenêtre.
- `finalScoreY = (windowHeight / 2) - 40` : Positionne le texte juste en dessous du message **"Game Over!"**.
- `DrawText(finalScoreText, finalScoreX, finalScoreY, 20, BLACK)` : Affiche le texte du score final en noir (**BLACK**) à la position calculée.

4. Bouton "Play Again" (Rejouer)

- `Rectangle restartGameButton = { windowWidth / 2 - 60, (windowHeight / 2) + 20, 120, 40 }` :
- Crée un rectangle pour le bouton **"Play Again"** (Rejouer). Ce bouton est centré horizontalement avec une largeur de 120 pixels et une hauteur de 40 pixels. Il est placé juste en dessous du texte du score final.
- `DrawRectangleRec(restartGameButton, WHITE)` : Dessine le rectangle du bouton **"Play Again"** en blanc (**WHITE**).
- `DrawText("Play Again", restartGameButton.x + 20, restartGameButton.y + 10, 20, BLACK)` :
- Affiche le texte **"Play Again"** au centre du bouton, avec un décalage de 20 pixels en X et 10 pixels en Y pour le centrer dans le rectangle.

5. Bouton "Exit" (Quitter)

- `Rectangle exitGameButton = { windowWidth / 2 - 60, restartGameButton.y + 60, 120, 40 }` :
- Crée un rectangle pour le bouton **"Exit"** (Quitter). Ce bouton est placé juste en dessous du bouton **"Play Again"**, avec une largeur de 120 pixels et une hauteur de 40 pixels.
- `DrawRectangleRec(exitGameButton, WHITE)` : Dessine le rectangle du bouton **"Exit"** en blanc (**WHITE**).
- `DrawText("Exit", exitGameButton.x + 35, exitGameButton.y + 10, 20, BLACK)` :
- Affiche le texte **"Exit"** au centre du bouton, avec un décalage de 35 pixels en X et 10 pixels en Y pour le centrer dans le rectangle.

Conclusion et perspectives futures

Le développement d'un jeu de casse-tête de type labyrinthe en C++ et Raylib est un projet stimulant qui permet d'explorer les principes de la programmation orientée objet et de la conception de jeux.

Le jeu pourrait être enrichi avec des fonctionnalités supplémentaires, telles que des modes de jeu multijoueur, des niveaux supplémentaires, des graphismes améliorés et des effets sonores immersifs.

Ce projet est une excellente occasion d'apprendre et de se développer en tant que développeur de jeux, et il ouvre la voie à de nombreuses perspectives futures pour créer des jeux plus complexes et plus ambitieux.