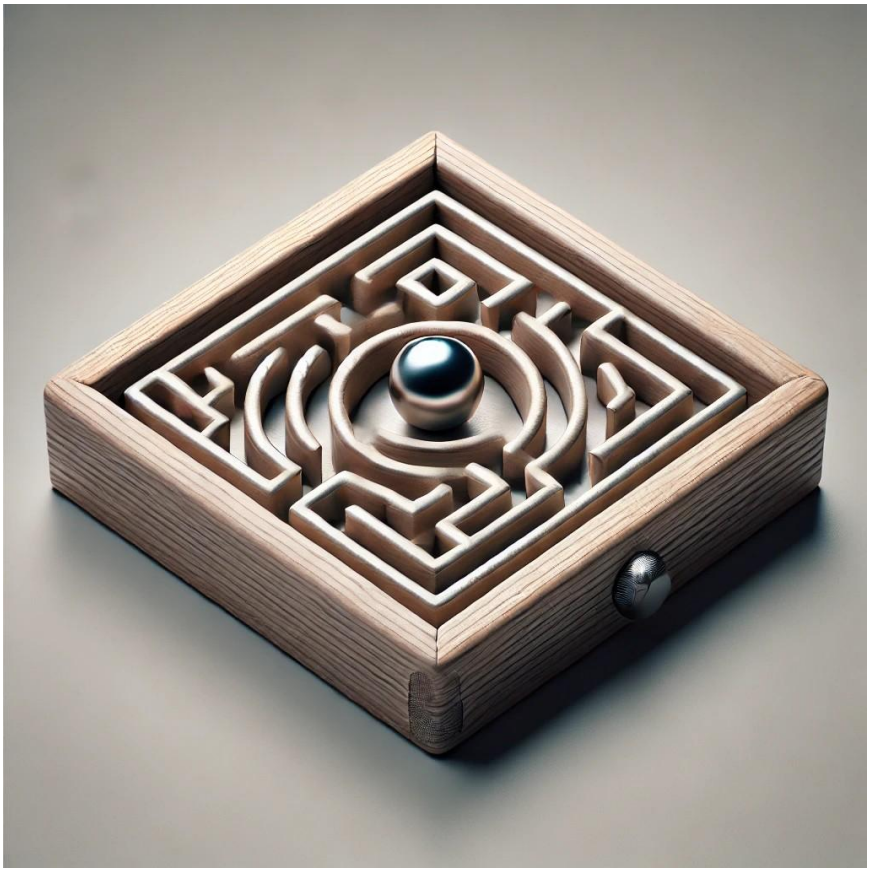
Rapport détaillé sur le code du jeu Labyrinthe



Réalisé par :

Rihab bouhmouch

Salma essaih

Bilal sebti

Abderrahmane amari ghazi

Encadrée par :

Prof Ikram ben abdel ouahab

Introduction :

Ce projet a pour objectif de développer un jeu interactif de labyrinthe en utilisant le langage C++ et la bibliothèque Raylib. Le jeu met en œuvre des concepts fondamentaux de la programmation orientée objet (POO), tels que les classes et les objets, pour modéliser le labyrinthe, le joueur, et la logique du jeu. Une méthode d'algorithme de backtracking (DFS) a été utilisée pour générer un labyrinthe aléatoire et jouable. Ce projet inclut également une interface utilisateur intuitive, un chronomètre en temps réel, et la gestion de différents niveaux de difficulté. Le jeu vise à offrir une expérience immersive et éducative en combinant des concepts algorithmiques et des techniques de développement graphique.

Structure Générale du Programme

Le code est organisé autour de **quatre classes principales** :

1. **Labyrinthe** : Génère et gère le labyrinthe.
2. **Joueur** : Représente le joueur et ses déplacements.
3. **Niveau** : Définit la difficulté du jeu en fonction du choix de l'utilisateur.
4. **Jeu** : Gère les états du jeu, le menu, le chronomètre et la logique de victoire.

Chaque classe a des responsabilités bien définies et interagit avec les autres pour créer une expérience de jeu cohérente.

Classe Labyrinthe : Génération et Gestion d'un Labyrinthe Dynamique

La classe Labyrinthe est le composant central du jeu, responsable de la génération, de la structure et de l'affichage du labyrinthe. Elle modélise le labyrinthe comme une grille de cellules, où chaque cellule possède des attributs tels que des murs et un état visité.

La méthode GenererLabyrinthe() utilise l'algorithme de backtracking avec une pile pour créer un labyrinthe unique à chaque partie. En supprimant les murs entre les cellules adjacentes lors de l'exploration, elle génère des chemins tout en respectant les contraintes structurelles du labyrinthe. Cela garantit une expérience toujours renouvelée et engageante.

L'affichage est géré par la méthode DessinerLabyrinthe(), qui dessine chaque cellule et ses murs à l'écran à l'aide de Raylib. Cela fournit une représentation visuelle claire du labyrinthe et des chemins que le joueur peut emprunter.

1.1. Structure interne : Cell

La classe Labyrinthe repose sur une structure interne appelée **Cell**, qui représente l'unité fondamentale de construction du labyrinthe.

Attributs :

- **Coordonnées spatiales** : Chaque cellule possède des attributs **x** et **y** permettant de la localiser précisément dans la grille.
- **État de visite** : Un booléen **visited** pour suivre l'exploration des cellules lors de la génération du labyrinthe.
- **Gestion des murs** : Quatre booléens (**topWall, rightWall, bottomWall, leftWall**) représentant l'état des murs.
    - `true` indique la présence d'un mur.
    - `false` signifie un passage ouvert.

Rôle :
La structure **Cell** est un élément fondamental. Chaque cellule a des murs initiaux (tous activés). Pendant la génération du labyrinthe, certains murs sont supprimés pour créer des chemins.

1.2. Attributs de la classe Labyrinthe

- **rows, cols** : Nombre de lignes et colonnes du labyrinthe. Définit la taille totale.
- **cellSize** : Taille d'une cellule en pixels, calculée dynamiquement en fonction de la largeur de l'écran.
- **grille** : Une matrice 2D (type `vector<vector<Cell>>`) contenant toutes les cellules.

1.3. Constructeur et Initialisation

Labyrinthe(int r, int c)

- Rôle : Constructeur qui initialise le labyrinthe avec ses dimensions.
- **Paramètres d'entrée** : Prend le nombre de lignes et de colonnes comme arguments.
- **Calcul de la taille des cellules** : Déterminée dynamiquement en fonction de la largeur de l'écran.
- **Création de la grille** : Initialise une grille 2D de cellules avec leurs coordonnées respectives.

Méthode GenererLabyrinthe()

La méthode de génération utilise l'algorithme **backtracking** (parcours en profondeur, **DFS**).

Étapes détaillées :

1. **Initialisation** :
    - La cellule en haut à gauche est marquée comme visitée.
    - Une pile (**stack**) suit les cellules à explorer.
2. **Exploration** :
    - Tant que la pile n'est pas vide :
        - Récupère la cellule courante.

        - Identifie ses voisins non visités.
        - Si des voisins existent :
            - En choisit un au hasard.
            - Supprime les murs entre la cellule courante et le voisin.
            - Ajoute le voisin à la pile et le marque comme visité.
        - Sinon, effectue un retour en arrière (retire la cellule de la pile).

Avantages :

- Garantie d'un labyrinthe totalement connexe.
- Génération unique et aléatoire à chaque exécution.

Méthode DessinerLabyrinthe()

- **Parcours de la grille** : Analyse chaque cellule.
- **Dessin des murs** : Utilise **Raylib** pour dessiner des lignes blanches avec **DrawLine()**.
- **Représentation visuelle** : Respect exact de la topologie générée.

Méthode VerifierSolution()

- Rôle : Vérifie si le joueur a atteint la cellule de sortie.
- **Paramètres d'entrée** : Position actuelle du joueur (**playerX**, **playerY**).
- **Condition de victoire** : Coin inférieur droit du labyrinthe.
- **Retour** : `true` si la condition est remplie.

Classe Joueur : Gestion des Déplacements et de l'Interaction avec le Labyrinthe

La classe Joueur gère la position et l'affichage du joueur dans le labyrinthe. Elle définit les coordonnées actuelles du joueur ainsi que sa couleur et sa taille, permettant une représentation visuelle simple mais efficace.

La méthode Afficher() positionne le joueur à l'écran en fonction des coordonnées de la grille, tandis que la méthode Deplacer() permet de contrôler ses mouvements. Les déplacements sont conditionnés par l'absence de murs, garantissant ainsi une navigation conforme à la structure du labyrinthe**.**

Attributs principaux :

- **x, y** : Coordonnées actuelles du joueur dans la grille.
- **couleur** : Couleur graphique du joueur (par défaut : bleu).
- **taille** : Rayon du cercle représentant le joueur, proportionnel à la taille des cellules.

Constructeur et Initialisation

- **Joueur()** : Initialise la position `(0,0)` et la couleur par défaut (bleue).

- **Initialiser(int cellSize)** : Définit la taille dynamique du joueur.

Méthode Afficher()

- **Calcul des coordonnées graphiques** : Conversion des indices `(x, y)` en pixels.
- **Affichage** : Utilise **DrawCircle()** pour représenter un cercle coloré.

Méthode Deplacer()

- **Entrées utilisateur** : Capture les touches directionnelles.
- **Validation** :
  - Vérifie les limites de la grille.
  - Contrôle l'absence de murs dans la direction souhaitée.
- **Mise à jour** : Modifie la position `(x, y)` si les conditions sont remplies.

Classe Niveau : Sélection de la Difficulté

La classe Niveau est responsable de la gestion des paramètres de difficulté du jeu. Elle définit le nombre de lignes et de colonnes du labyrinthe en fonction du niveau choisi, offrant ainsi une expérience adaptée à chaque type de joueur.

Grâce à la méthode ChoisirNiveau(), les niveaux Facile, Moyen et Difficile sont paramétrés respectivement avec des labyrinthes de tailles croissantes, augmentant ainsi la complexité du jeu. Cette modularité permet une personnalisation rapide et intuitive du défi proposé au joueur.

Attributs :

- **rows, cols** : Dimensions ajustées selon le niveau sélectionné.

Méthode ChoisirNiveau(int niveau)

- **Niveau 1** : 10x10.
- **Niveau 2** : 15x15.
- **Niveau 3** : 20x20.
- **Par défaut** : Niveau moyen si l'entrée est invalide.

Classe Jeu : Gestion Globale

La classe *Jeu* est le cœur de la logique du jeu de labyrinthe, gérant les états du jeu, le chronomètre et l'expérience utilisateur. Elle organise le déroulement de la partie à travers trois états principaux : *MENU, **EN_COURS* et *TERMINE*, offrant une progression structurée depuis la sélection du niveau jusqu'à la victoire.

Un élément clé est la gestion du chronomètre, qui débute dès le lancement du jeu grâce à la méthode DemarrerChrono() et suit dynamiquement le temps écoulé pendant la partie. La

méthode AfficherChrono() met à jour ce temps en continu, ou le fige lorsque le jeu se termine, ajoutant une dimension compétitive et immersive.

Le menu, affiché via AfficherMenu(), présente une interface simple permettant de choisir parmi plusieurs niveaux de difficulté, chacun offrant une complexité croissante. L'écran de victoire, quant à lui, affiche un message de félicitations et le temps total, clôturant le jeu avec une satisfaction pour le joueur.

Grâce à une gestion fluide des états et une séparation claire des responsabilités entre les méthodes, la classe *Jeu* fournit une expérience utilisateur cohérente et adaptable, tout en respectant les principes de la programmation orientée objet.

Attributs :

- **startTime, elapsedTime** : Suivi du chronomètre.
- **EtatJeu** : MENU, EN_COURS, TERMINE.
- **tempsFinal** : Temps total après victoire.

Méthodes

- **DemarrerChrono()** : Démarre le chronomètre.
- **AfficherMenu()** : Affiche le menu principal.
- **AfficherChrono()** : Met à jour et affiche le temps.
- **AfficherVictoire ()** : Affiche un message de victoire.

Interconnexion des Classes

1. **Choix du niveau** : Détermine les dimensions du labyrinthe.
2. **Génération** : Création dynamique d’un labyrinthe unique.
3. **Déplacement** : Interaction fluide entre le joueur et le labyrinthe.
4. **Victoire** : La classe Jeu gère la fin de la partie.

Conclusion :

Le développement de ce jeu de labyrinthe a permis d'explorer plusieurs aspects importants du développement logiciel, notamment l'utilisation d'algorithmes avancés, la programmation orientée objet et les fonctionnalités graphiques offertes par Raylib. Le projet a atteint ses objectifs en proposant un labyrinthe aléatoire à plusieurs niveaux de difficulté, un chronomètre fonctionnel, et une expérience utilisateur fluide. Ce projet constitue une base solide pour des améliorations futures, telles que l'ajout de fonctionnalités avancées comme un système de score, des obstacles dynamiques ou des modes multijoueurs. Cette réalisation reflète l'application concrète de concepts théoriques pour créer un jeu fonctionnel et engageant.