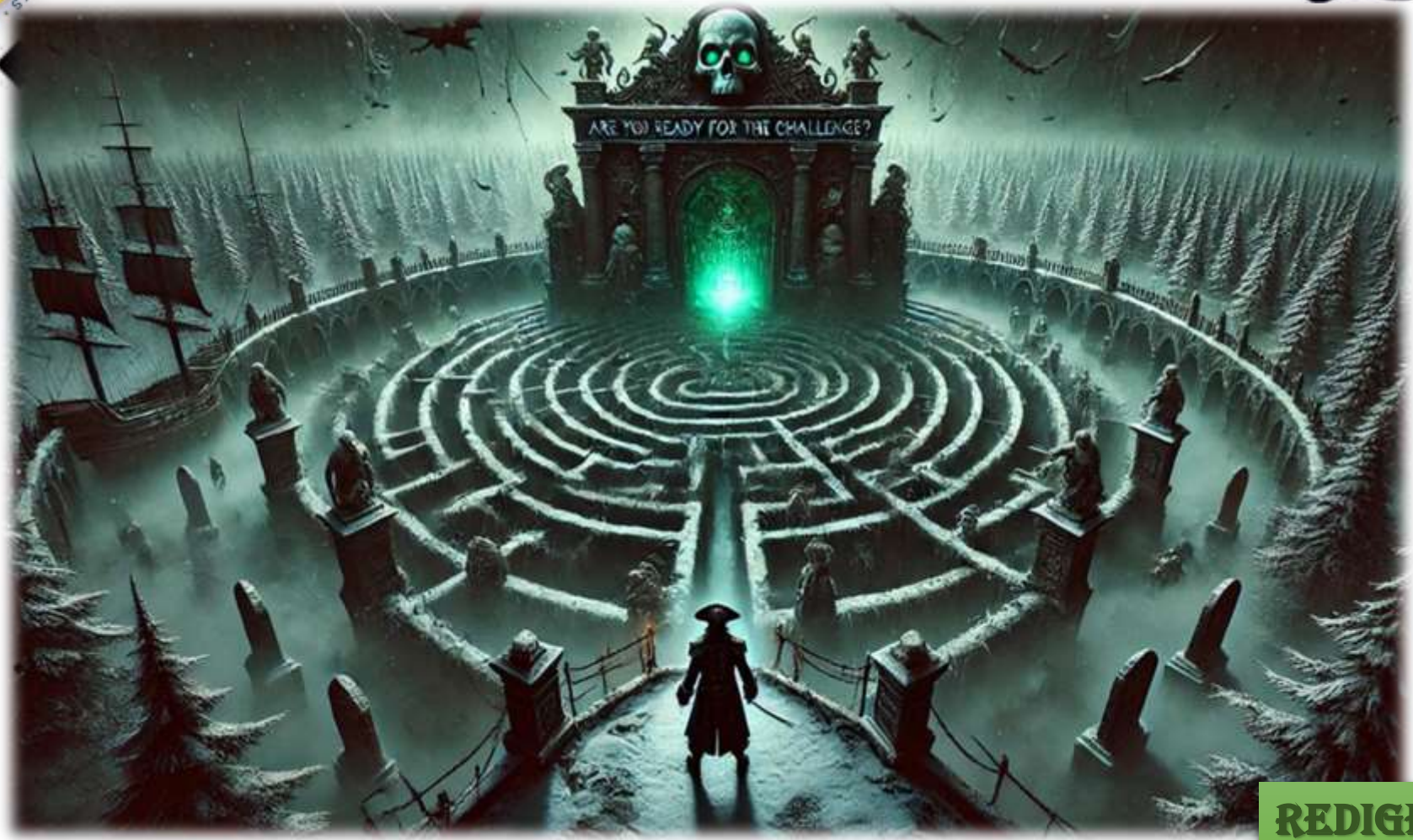
PROJET DE FIN DE MODULE



ENCADRE PAR : Ikram Ben abdel ouahab

REDIGER PAR :
SABER AZIAT
AYMANE EL GHADBANI
SABER BEN HAMDA

Sommaire

- ✓ Définition du bibliothèque rylib
- ✓ Introduction de notre projet
- ✓ L'explication du class cell.h
- ✓ L'explication du class GameController
- ✓ L'explication du class joueur.h
- ✓ L'explication du class maze.h
- ✓ L'explication du main

- **DÉFINITION DU BIBLIOTHÈQUE RYLIB**

La bibliothèque **RyLib** en C++ est une bibliothèque conçue pour faciliter le développement d'applications orientées objet (POO). Elle fournit un ensemble d'outils et de composants réutilisables pour gérer des tâches courantes comme la manipulation des fichiers, des structures de données, des opérations mathématiques, la gestion des événements, ou encore la gestion des interfaces graphiques, selon ses fonctionnalités.

La bibliothèque **Rylib** peut inclure des fonctionnalités comme :

- La gestion des erreurs et des exceptions de manière simplifiée.
- Des algorithmes pratiques pour la manipulation de données.
- Des outils pour la gestion des fichiers et des flux de données.
- Des abstractions pour simplifier les tâches de programmation courantes.

INTRODUCTION DE NOTRE PROJET

L’objectif de notre projet est de développer un jeu de casse-tête (maze) de type labyrinthe en utilisant la
programmation orientée objet (POO) en C++ et la bibliothèque graphique Raylib

. Pour Le jeu , il proposera
une expérience immersive où le joueur devra naviguer dans un labyrinthe généré aléatoirement à
chaque nouvelle partie. Il comprendra trois niveaux de difficulté (facile, moyen, difficile), qui
affecteront la taille du labyrinthe et la complexité de la navigation.

✓ L'explication du class cell.h

1. Inclusion des bibliothèques

Gestion des entrees/sorties (iostream)
Utilisation de conteneures (vector , etc)
Fonctionnalites graphiques (rylib)

1. 2. Variables Membres

. **Int x y :** representer la position de la cellule dans une grille

Bool visited = false : Indique si la cellule a été visitée par l'algorithme de génération ou de résolution du labyrinthe.

Vector<bool> walls ={true, true, true, true };

Indique si la cellule a été visitée par l'algorithme de génération ou de résolution du labyrinthe.

- walls[0] : Mur supérieur
- walls[1] : Mur droit
- walls[2] : Mur inférieur
- walls[3] : Mur gauche.
- **Int defficulte** : Définit la difficulté du labyrinthe. Les valeurs possibles sont :
  - 1 : facile
  - 2: moyenne
  - 3: deficile
- **Int TILE=0** : Taille en pixels d'une cellule, ajustée en fonction de la difficulté.
- **Int thickness = 4** : Épaisseur des murs, utilisée dans le dessin des lignes.

Cell* previous = nullptr : Pointeur vers la cellule précédente pour le traçage ou la résolution du labyrinthe (utile pour le backtracking).

3. Constructeur :

Initialise une cellule avec sa position (x ; y) sa difficulté, et calcule la taille (TILE) correspondante

1. **4. Méthode DRAW :**

Affiche la cellule et ses murs à l'écran dans la fenêtre Raylib

- **Position de la cellule** : calcume a partir des cordonnees x et y
- **Couleur des murs** definir avec la valeures RGBA
- **Dessin des murs :**
- **Fonction DrawLineEx :** permet de dessiner des lignes épaisses avec des positions flottantes.

✓ L'EXPLICATION DU CLASS GAMECONTROLLER

1. Introduction to the Class

The `GameController` class is the central component of the game, managing the game's flow. It handles player input, difficulty selection, game timing, and score management, ensuring that the game runs smoothly and provides an engaging experience for the player.

2. Player Input and Name Selection

At the start of the game, players are prompted to choose between one-player or two-player mode. Based on this choice, they are asked to input their names. In one-player mode, only Player 1's name is requested, while in two-player mode, both players provide their names. The names are then displayed on the screen.

```c++
if (IsButtonPressed(onePlayerButton)) {
    playerChoice = 1;  // One player selected
}
if (IsButtonPressed(twoPlayerButton)) {
    playerChoice = 2;  // Two players selected
}
```

For each player, their name is captured via the `takeinput` function and displayed:

```c++
string name1 = takeinput(player1Name, player1NameLength, maxNameLength, isEnterKeyPressed, isInputActive);
```

3. Difficulty Selection

After selecting the game mode and entering names, players must choose a difficulty level. The options are:

- **Easy (30 seconds)**
- **Medium (70 seconds)**
- **Hard (150 seconds)**

Players select the difficulty using keyboard input, which determines the game duration.

```c++
if (IsKeyPressed(KEY_ONE)) {
    choixDifficulte = 1;  // Easy
} else if (IsKeyPressed(KEY_TWO)) {
    choixDifficulte = 2;  // Medium
} else if (IsKeyPressed(KEY_THREE)) {
    choixDifficulte = 3;  // Hard
}
```

4. Game Timer and Countdown

The core of the gameplay is the countdown timer, which begins once the game starts. The timer is initialized based on the selected difficulty. A visual countdown, alongside a sand clock icon, is displayed to show the remaining time.

```c++
oid miseAJour(bool &jeuTermine) {
    (enCours) {
        float tempsEcoule = GetTime();
        tempsRestant = tempsInitial - tempsEcoule;
        if (tempsRestant <= 0) {
            tempsRestant = 0;
            arreter();
            jeuTermine = true;
        }}}
```

5. Leaderboard and Score Management

The game includes a leaderboard that tracks high scores. At the end of each game, the player's score is calculated and saved. The score is determined by multiplying the remaining time by the difficulty level:

```c++
int calculate_score(int difficulte) {
    return tempsRestant * difficulte;
}
```

Scores are stored and can be loaded or saved to a file for future comparisons:

```c++
void saveScore(const std::string& playerName, float score) {
    std::ofstream file(leaderboardFilePath, std::ios::app);
    if (file.is_open()) {
        file << playerName << ": " << static_cast<int>(score) << "\n";
    }
}
```

6. Key Methods in the GameController Class

- ***`demarrer(int difficulte)`**:* Starts the game with the selected difficulty, initializing the timer:
```c++
void demarrer(int difficulte) {
    tempsInitial = (difficulte == 1) ? 30 : (difficulte == 2) ? 70 : 150;
    enCours = true;
    jeuCommence = true;
}
```
- ***`miseAJour(bool &jeuTermine)`**:* Updates the timer every frame and checks if the game has ended.
```c++
void miseAJour(bool &jeuTermine) { /* Timer update logic */ }
```

- ***`afficherCompteARebours(float x, float y, int tailleTexte, Color couleur)`**:* Displays the countdown timer and a sand clock icon.
```c++
void afficherCompteARebours(float x, float y, int tailleTexte, Color couleur) { /* Display countdown logic */ }
```

- ***`loadLeaderboardScores()`**:* Loads the previous scores from a leaderboard file:
```c++
std::vector<Score> loadLeaderboardScores() { /* Load leaderboard from file */ }
```

***`saveScore(const std::string& playerName, float score)`**:* Saves the current score to the leaderboard.
```c++
void saveScore(const std::string& playerName, float score) { /* Save score logic */ }
```

- ***`calculate_score(int difficulte)`**:* Calculates the score based on remaining time and difficulty.
```c++
int calculate_score(int difficulte) { /* Calculate score logic */ }
```

7. Summary of Features

- ***Player Input**:*
Players input their names and choose between one or two players.
- ***Difficulty Levels**:*
- Players choose from three difficulty levels: Easy, Medium, and Hard.
- ***Timer Management**:*
- A countdown timer controls the game, ending when time runs out.
- ***Leaderboard**:*
- The game tracks scores through a leaderboard stored in a file.
- ***Score Calculation**:*
- Scores are calculated based on the remaining time and difficulty

L'EXPLICATION DU CLASS JOUEUR.

1. Inclusion des bibliothèques

#ifndef et #define : Ces directives empêchent une inclusion multiple du fichier (joueur_h)

#include "maze.h « : contient probablement la définition des classes maze et cell ,Ces classes sont nécessaires pour manipuler le joueur dans le contexte du labyrinthe

1. 2 Définition des attributs de la classe

- ✓ Cell* currentCell : Un pointeur vers la cellule actuelle dans laquelle se trouve le joueur.
- ✓ Int difficulte : Détermine la difficulté du jeu, affectant la taille visuelle des cellules.
- ✓ Int mouvement : Compte le nombre total de mouvements du joueur.
- ✓ Int largeurCellule : Contrôle la taille de la cellule dans laquelle le joueur est dessiné (en pixels).
- ✓ texture2D traisor : Image utilisée pour représenter le joueur graphiquement.
- ✓ Rectangle bckgroundTextureRect : Définit une zone rectangulaire pour afficher la texture du joueur.
- ✓ String player_name : Stocke le nom du joueur.

3. Constructeur de la classe

- ✓ currentCell: La position initiale du joueur dans le labyrinthe.
- ✓ Difficulte :
- ✓ Player_name
- ✓ Condition sur la difficulté : Selon le niveau, la taille des cellules
- ✓ Chargement de texture : La texture « ship.png » est chargée et utilisée pour représenter le joueur graphiquement.
- ✓ Définition de la zone de texture : backgrpundTextureRect positionne et dimensionne l'image du joueur sur l'écran.

1. 4. Mouvement du joueur

- ✓ Selon la direction ('R', 'L', 'D', 'U'), le joueur tente de se déplacer.
- ✓ Vérifie si un mur empêche le mouvement (via currentCell->walls)
- ✓ Si le mouvement est autorisé, le joueur change de cellule(nextCell) et le compteur de mouvements (mouvement) augmente

1. 5. Dessin du joueur

- ✓ Dessine le joueur dans sa cellule actuelle (currentCell).
- ✓ La taille et l'échelle de l'image (scale) sont ajustées en fonction de largeurCellule.

1. 6. Vérification de l'objectif

- ✓ Vérifie si le joueur a atteint la cellule de sortie du labyrinthe (sortieCell)

1. 7. Getters et setters

- ✓ **Getters** : Permettent d'accéder aux coordonnées (x , y) du joueur et au nombre de mouvements effectués.
- ✓ **Setter** : Permet de modifier directement la cellule actuelle du joueur

L'EXPLICATION DU CLASS MAZE.H

Constructor

- Sets up the maze based on the difficulty level.
- Calculates tile sizes (`largeurCellule`) and grid dimensions.
- Initializes each cell in the `grid` vector.
- Loads textures for rendering.

2. Main Loop

The algorithm runs until all cells in the grid are visited:

1. **Mark the Current Cell as Visited**:
   - Set `current->visited = true`.

2. **Check Neighbors**:
   - Use the `checkNeighbors(current)` method to find unvisited neighboring cells.
   - Neighbors are cells directly above, below, to the left, or to the right of the current cell.

3. **Move to a Neighbor**:
   - If there are unvisited neighbors:
     - Randomly choose one neighbor using a random number generator.
     - Remove the wall between the `current` cell and the chosen neighbor using `removeWalls(current, next)`.
     - Push the `current` cell onto the stack.
     - Move the `current` pointer to the chosen neighbor.
   - Increment the `visitedCount`.

4. Backtrack if No Neighbors**:

- If there are no unvisited neighbors:
- Pop a cell from the stack and set it as `current`.
- Continue the process.

5. Add Dead Ends**:

- Occasionally, check if the `current` cell has no neighbors left after moving. If so, treat it as a dead-end cell.
- Ensure the number of dead ends doesn't exceed the `MAX_DEAD_ENDS` limit.

3. Completion

- The loop ends when all cells have been visited (`visitedCount == grid.size()`).
- The bottom-right cell is designated as the `endCell`.

Key Methods in Maze Generation

1. **`checkNeighbors(Cell* current)`**

- Finds all valid, unvisited neighbors of the current cell.
- Returns a randomly selected unvisited neighbor, or `nullptr` if no neighbors are available.

2. **`removeWalls(Cell* current, Cell* next)`**

- Removes walls between two adjacent cells based on their relative positions (left, right, top, bottom).
- This connects the two cells, carving a path in the maze.

3. **`generateMaze()`**

- Combines the steps above to carve paths in the maze and introduce dead ends.

Why DFS with Backtracking Works

Guarantees a Perfect Maze:

- The algorithm generates a maze with exactly one path between any two points (no loops or isolated regions).
- **Efficient and Simple**:
- It uses a stack to manage backtracking and ensures every cell is visited exactly once.
- **Scalable**:

- The method works for grids of any size and can be easily adapted for different levels of difficulty.

Solving the Maze

- - `solveMaze()`:
- The method used for solving the maze in our Maze class is Backtracking from the End Cell to the Start Cell. This is a simple and efficient approach, leveraging the information stored during the maze generation process.
- Key Idea :
- During maze generation, each cell keeps track of its "previous" cell (the cell from which it was visited).
- The solution is determined by tracing back from the endCell to the starting cell using the previous pointers.

Rendering

- `draw()`:
  - Draws the maze grid, background, and solution path (if enabled).
  - Highlights the `endCell` with a treasure marker.
  - Uses Raylib functions to render textures and draw shapes.
. **Utility Methods**
- `toggleSolution()`: Toggles the visibility of the solution path.
- `checkCell(x, y)`: Validates and retrieves a cell at specific coordinates.
- `checkNeighbors(current)`: Finds unvisited neighbors for maze generation.
- `removeWalls(current, next)`: Removes walls between two adjacent cells to carve the maze.

Generation:

- The maze starts carving from the top-left cell.
- Neighbors are randomly chosen, and walls are removed to connect cells.
- The process ends when all cells are visited.

Interactive Features

- Toggle Solution:
  - Press a key to show or hide the solution path (`showSolution` flag).
- Difficulty Levels:
  - Change the difficulty to modify the maze's size and complexity

L'EXPLICATION DU MAIN

1. 1. Les bibliothèques utilisées

#include "raylib.h" : Bibliothèque graphique utilisée pour le rendu 2D.
#include "classes/cell.h", maze.h, joueur.h, chrono.h : Fichiers d'en-tête contenant les définitions des classes utilisées pour :

- ✓ **Cell** : Représentation d'une cellule du labyrinthe.
- ✓ **Maze** : Gestion et génération du labyrinthe.
- ✓ **Joueur** : Contrôle des joueurs dans le labyrinthe.
- ✓ **Chrono** : Gestion du chronomètre et des scores.

1. 2. Les variables globales

- ✓ int LARGEUR_CELLULE = 60; : Définit la largeur des cellules du labyrinthe.
- ✓ int tailleX, tailleY; : Dimensions du labyrinthe.
- ✓ Rectangle toggleSolutionButton et restartButton : Boutons interactifs pour afficher la solution du labyrinthe et redémarrer le jeu.
- ✓ Texture2D restartTexture, toggleTexture : Textures pour personnaliser les boutons avec des images.

1. 3. Fonctions utilitaires

a. DrawButtonWithImage

Affiche un bouton avec une image. Utilise les coordonnées et dimensions définies dans un objet Rectangle.

b. IsButtonPressed

Vérifie si un bouton a été cliqué. Cela repose sur la collision entre la position de la souris et les coordonnées du bouton.

4. Fonction principale

a. Initialisation

- **Fenêtre et FPS** :
- **Musique et textures**

a. b. Configuration du jeu

- ✓ **Difficulté :** Le joueur choisit le niveau de difficulté via un menu.
- ✓ **Labyrinthe :** Génération du labyrinthe basé sur la difficulté.
- ✓ **Joueurs :**
  - Joueur 1 : Positionné à la cellule de départ.
  - Joueur 2 : Optionnel, si le jeu est en mode deux joueurs.

a. c. Boucle principale

1. **Mise à jour :**

   Mise à jour du chronomètre

   Lecture et mise à jour du flux audio

2. **État du jeu**

Le programme gère deux états principaux :

Jeu terminé : Affichage de l'écran "Game Over".

Victoire : Affichage d'un écran de félicitations et possibilité de consulter le tableau des scores.

3. Interactions utilisateur :

Déplacements des joueurs

Boutons :

- **Afficher la solution**
- **Redémarrer**

4. Rendu graphique

À chaque itération, les éléments du jeu sont dessinés

- ✓ Labyrinthe : maze.draw();
- ✓ Joueurs : joueur1.dessiner();
- ✓ Chronomètre : chrono.afficherCompteARebours(10, 10, 20, BLACK);
- ✓ Boutons : DrawButtonWithImage(toggleSolutionButton, toggleTexture);

5. Gestion de fin de partie

a. Victoire

Lorsqu'un joueur atteint la sortie :

Son score est sauvegardé : chrono.saveScore(...).

Affichage d'un écran de victoire avec un message personnalisé.

b. Game Over

Si le temps expire :

Affichage d'un écran "Game Over".

Option pour redémarrer en appuyant sur R.

6. Nettoyage et fin

À la sortie du jeu :
Textures et musique sont déchargées.
Fenêtre fermée proprement.

CONCLUSION SUR LE PROJET DE JEU "MAZE" DÉVELOPPÉ AVEC RAYLIB :

Le développement du jeu "Maze" avec Raylib représente une expérience captivante qui allie créativité, logique et défis techniques. Ce projet met en avant l'utilisation des concepts fondamentaux de la programmation orientée objet, la gestion graphique 2D, et l'interaction utilisateur pour offrir une expérience de jeu fluide et engageante.