جامعة عبد المالك السعدي
Université Abdelmalek Essaadi

FACULTE DES SCIENCES ET TECHNIQUES
TANGER
كلية العلوم والتقنيات طنجة

RAPPORT PROJET

ENCADRÉ PAR

PR. IKRAM BENABDELOUAHAB

PREPARÉ PAR

CHENAOUI HIBA
BORKI MAYSSAE
DRIOUECH MANAL

2024 - 2025

PLAN
DU PROJET

Introduction

CONTEXTE

Ce projet a été réalisé dans le cadre de l'apprentissage de la programmation orientée objet (POO) et du développement de jeux 2D. Il consiste à créer un jeu simple de labyrinthe où le joueur doit naviguer dans un labyrinthe et atteindre la sortie. Le labyrinthe est généré automatiquement grâce à un algorithme appelé DFS (Depth-First Search). Pour rendre le jeu visuel et interactif, la bibliothèque Raylib a été utilisée. Ce projet permet de mieux comprendre la création de jeux tout en appliquant des concepts essentiels de programmation et de conception.

OBJECTIFS

Ce projet vise à développer un jeu 2D de labyrinthe avec la bibliothèque Raylib. Les objectifs principaux sont :

- Appliquer la programmation orientée objet (POO)
- Implémenter un algorithme de génération automatique de labyrinthe.
- Utiliser Raylib pour afficher le jeu et gérer les interactions.
- Créer un projet complet .

En réalisant ce projet, nous cherchons non seulement à développer un jeu fonctionnel, mais aussi à renforcer nos compétences techniques en conception logicielle et à découvrir les défis du développement de jeux vidéo.

La Bibliothèque Raylib

QU'EST-CE QUE RAYLIB ?

Raylib est une bibliothèque de programmation en C++ conçue pour faciliter le développement de jeux vidéo, la création des graphes en 2D et d'applications multimédia. Elle est particulièrement appréciée pour sa simplicité et sa facilité d'utilisation, ce qui en fait un excellent choix pour les débutants ainsi que pour les développeurs expérimentés.

POURQUOI UTILISER RAYLIB DANS CE PROJET ?

Dans notre projet de labyrinthe, nous avons principalement utilisé Raylib pour diverses fonctionnalités essentielles. Tout d'abord, Raylib nous a permis de créer et de gérer la fenêtre du jeu en définissant sa taille, son titre et en contrôlant son cycle de vie, notamment son ouverture et sa fermeture. Ensuite, cette bibliothèque a été utilisée pour dessiner les différents éléments du jeu, tels que les murs du labyrinthe, le personnage du joueur et les autres éléments de l'interface. Enfin, Raylib a facilité la gestion des événements utilisateur en détectant les entrées clavier et souris. Cela a été essentiel pour permettre au joueur de se déplacer dans le labyrinthe et pour détecter le niveau choisi par l'utilisateur.

PRINCIPALES FONCTIONNALITÉS UTILISÉES DANS CE PROJET

1. Gestion de la fenêtre

- **InitWindow** : Initialise une fenêtre pour le jeu avec des dimensions et un titre spécifiés.
  *Syntaxe : InitWindow(int width, int height, const char *title);*
- **WindowShouldClose** : Vérifie si l'utilisateur a demandé de fermer la fenêtre.

2. Gestion des dessins

- **BeginDrawing** et **EndDrawing** : Encadrent les instructions de dessin pour mettre à jour l'écran.
- **ClearBackground** : Efface l'écran avec une couleur spécifiée.
  Syntaxe : ClearBackground(Color color);

3. Gestion des entrées utilisateur

- **IsKeyPressed** : Vérifie si une touche du clavier a été pressée.
- **IsMouseButtonPressed** : Vérifie si un bouton de la souris a été pressé.
  Syntaxe : IsMouseButtonPressed(MOUSE_LEFT_BUTTON);

4. Objets graphiques

- **Rectangle** : Représente un rectangle avec des coordonnées (x, y), une largeur et une hauteur.
  Syntaxe : Rectangle rect = {x, y, width, height};
- **Vector2** : Représente un vecteur 2D utilisé pour des positions ou tailles.
- **Color** : Représente une couleur définie par ses composantes RGBA.
  Syntaxe : Color color = {r, g, b, a};

5. Gestion des textures et polices

- **LoadTexture** : Charge une image depuis un fichier pour en faire une texture.
  Syntaxe : Texture2D texture = LoadTexture("Image.png");
- **DrawTexturePro** : Dessine une texture avec des paramètres avancés (position, rotation, échelle).
  Syntaxe : DrawTexturePro(texture, sourceRec, destRec, origin, rotation, color);
- **LoadFont** : Charge une police depuis un fichier pour personnaliser le texte.
  Syntaxe : Font font = LoadFont("myFont.ttf");
- **DrawText** : Dessine un texte à une position donnée avec une couleur et une taille spécifiées.
  Syntaxe : DrawText(text, x, y, fontSize, color);

6. Détection des collisions et gestion de la souris

- **GetMousePosition** : Renvoie la position actuelle de la souris sous forme d'un vecteur 2D.
  Syntaxe : Vector2 mousePosition = GetMousePosition();
- **CheckCollisionPointRec** : Détecte si un point (x, y) est en collision avec un rectangle.
  Syntaxe : bool collision = CheckCollisionPointRec(Vector2{x, y}, rect);

7. Objets utilisés dans Raylib

- **Vector2** : Pour représenter des positions ou des tailles 2D.
  Syntaxe : Vector2 position = {x, y};
- **Texture2D** : Pour gérer les textures (images) dans le jeu.
  Syntaxe : Texture2D texture = LoadTexture("image.png");
- **Font** : Pour afficher du texte avec une police personnalisée.
  Syntaxe : Font font = LoadFont("police.ttf");

Organisation et structure du projet

STRUCTURE DES FICHIERS ET DES DOSSIERS

Dans ce projet, nous avons organisé les fichiers et dossiers de manière logique pour faciliter la gestion du code et la navigation. Voici l'arborescence suivie :

- **maze.hpp:** contient le prototype de la classe maze avec attributs et méthodes utilisées pour dessiner notre labyrinthe d'une manière aléatoire.
- **maze.cpp:** contient l'implémentation des méthodes de la classe maze.

- **player.hpp:** contient le prototype de la classe player avec attributs et méthodes utilisées pour dessiner notre joueur.
- **player.cpp:** contient l'implémentation des méthodes de la classe player.

- **game.hpp:** contient le prototype de la classe game avec attributs et méthodes utilisées pour gérer la boucle principale du jeu.
- **game.cpp:** contient l'implémentation des méthodes de la classe level.

- **main.cpp:** contient la fonction main().

RELATION ENTRE LES CLASSES

La classe principale du projet est Game, qui gère l'ensemble du déroulement du jeu. Elle contient un objet de type Maze, qui est responsable de la génération du labyrinthe ainsi que de la gestion des interactions, notamment les collisions avec les murs. En outre, Game possède également un objet de type Player, qui représente le joueur dans le jeu.

La classe Player s'occupe de la gestion de la position et des déplacements du joueur, en mettant à jour ces informations en fonction des entrées clavier capturées par Game. Ainsi, Game orchestre l'ensemble des éléments du jeu, en s'assurant que le labyrinthe soit généré et que le joueur puisse interagir avec ce dernier en se déplaçant à travers le labyrinthe à l'aide des commandes clavier.

Conception et implémentation des classes

CLASSE MAZE:

I.Conception:

1.Definition:

Dans cette classe, introduit deux classes principales Cell et Maze:

- **Cell:** représente une cellule individuelle (unique) dans un labyrinthe. Chaque cellule contient les informations nécessaires pour construire un labyrinthe et gérer ses propriétés, telles que les murs, la position et l'état de visite.
- **Maze:** représente un labyrinthe complet et de fournir les méthodes nécessaires pour sa génération, son affichage, et la manipulation des cellules qui le composent.

2.Algorithme de génération de labyrinthe

2.1.Initialisation: création d'une grille de carrés

La génération du labyrinthe se déroule en deux étapes principales dans la premiere etapes on commence par créer une grille de carrés ou chaque carré représente une cellule. Au début chaque cellule est créée avec tous ses murs présents.

2.2.Algorithme DFS: Comment fonctionne dans la génération de labyrinthes ?

Dans la deuxième étape, l'algorithme DFS (Depth First Search) supprime progressivement les murs pour créer des passages dans le labyrinthe, en partant d'un état où tous les murs sont intacts. Il garantit que chaque cellule est visitée au moins une fois et que le labyrinthe reste connexe.

L'algorithme commence par choisir une cellule aléatoire comme point de départ, puis marque cette cellule comme visitée. Tant que des cellules voisines non visitées existent, un voisin est sélectionné aléatoirement et le mur entre les deux cellules est supprimé. La cellule voisine devient la nouvelle cellule courante.

Si la cellule actuelle n'a plus de voisins non visités, l'algorithme revient à la cellule précédente en utilisant une pile et continue la recherche jusqu'à ce que toutes les cellules aient été visitées, générant ainsi un labyrinthe complet.

2.3.Pourquoi on a utilisé le DFS pour la génération de labyrinthes ?

L'algorithme DFS pour la génération du labyrinthe présente deux principaux avantages. Tout d'abord, le choix aléatoire des voisins permet de créer des labyrinthes uniques à chaque lancement du jeu, offrant ainsi une expérience variée. De plus, l'algorithme explore toutes les cellules de la grille, garantissant que le labyrinthe final est entièrement connecté, sans aucune zone isolée ou inaccessible.

II.Implémentation:

1.Class Cell

1.1.ATTRIBUTS

- **x** et **y** sont initialisés via le constructeur pour positionner correctement la cellule dans la grille.
- **visited** est initialisé à false, indiquant que la cellule n'a pas encore été explorée.
- **walls[0..3]** sont initialisés à true, ce qui signifie que tous les murs sont présents par défaut.

1.2.MÉTHODES

Le Constructeur : Le constructeur de la classe Cell permet d'initialiser une cellule avec ses coordonnées *(x et y)* dans la grille du labyrinthe. Il configure également l'état initial de la cellule en marquant tous ses murs comme présents (walls à true) et en définissant la cellule comme non visitée (visited à false).

2.Class Maze

2.1.ATTRIBUTS

- **width** et **height** permet la définition des dimensions du labyrinthe.
- **cellSize** pour afficher la taille d'une cellule.
- **offsetX** et **offsetY** définie le décalage pour positionner le labyrinthe.
- **grid** est un vecteur qui définit une grille de 2D.
- **cellStack** est une pile pour l'algorithme de génération.

2.2.MÉTHODES

Le Constructeur: Le constructeur *Maze::Maze* initialise le labyrinthe en définissant ses dimensions *(width, height)*, la taille des cellules *(cellSize)* et les offsets *(offsetX, offsetY)*. *Une grille* bidimensionnelle est créée, chaque cellule étant représentée par *un objet Cell* avec *ses coordonnées (x, y)*. Les bordures du labyrinthe sont adaptées en ouvrant les murs gauche et droit au milieu pour créer une entrée et une sortie. Ensuite, *l'algorithme* ***DFS*** est utilisé pour générer les chemins du labyrinthe de manière procédurale, en commençant par la cellule en haut à gauche. Enfin, *srand(time(0))* garantit une génération aléatoire des labyrinthes à chaque exécution.

La méthode DRAW : Commence par *une double boucle for* pour parcourir chaque cellule de *la grille*. La boucle extérieure parcourt les lignes de la grille, en utilisant la variable ***y*** pour l'index de la ligne. La boucle intérieure parcourt les colonnes de chaque ligne, en utilisant la variable ***x*** pour l'index de la colonne;

Pour chaque cellule située aux coordonnées *(x, y),* la méthode drawCell est appelée avec la cellule correspondante de la grille *(grid[y][x])* comme argument. Cette méthode dessine les murs de la cellule sur la grille. En parcourant ainsi toutes les cellules du labyrinthe, la méthode **draw** garantit que chaque cellule est correctement dessinée sur la grille, ce qui permet de visualiser l'ensemble du labyrinthe

.

La méthode generateMaze commence par marquer la cellule de départ, passée en paramètre, comme visitée en définissant *start->visited* à true. Ensuite, cette cellule est ajoutée à une pile *(cellStack)* pour suivre les cellules à explorer.

La méthode entre ensuite dans la boucle *while* qui continue tant que la pile n'est pas vide. À chaque itération de la boucle, la cellule au sommet de la pile est récupérée et stockée dans un pointeur current. Ensuite, la méthode **getRandomNeighbor** est appelée avec current comme argument pour obtenir un voisin aléatoire de la cellule actuelle.

Si un voisin valide est trouvé (c'est-à-dire que next n'est pas un pointeur nul), ce voisin est marqué comme visité en définissant **next->visited** à true. Ensuite, la méthode removeWalls est appelée pour supprimer les murs entre la cellule actuelle (current) et le voisin choisi (next). Après cela, le voisin est ajouté à la pile pour être exploré ultérieurement.

Si aucun voisin valide n'est trouvé (c'est-à-dire que next est un pointeur nul), cela signifie que la cellule actuelle n'a plus de voisins non visités. Dans ce cas, la cellule est retirée de la pile, permettant ainsi de revenir en arrière et d'explorer d'autres chemins.

Cette méthode continue d'explorer et de générer le labyrinthe jusqu'à ce que toutes les cellules accessibles aient été visitées, créant ainsi un labyrinthe parfait sans cycles ni passages morts.

La méthode hasWall: de la classe Maze permet de vérifier s'il y a un mur dans une direction donnée à partir d'une position spécifique (x, y) sur la grille du labyrinthe **grid[y][x].walls[direction].**

getRandomNeighbor: La méthode commence par créer un vecteur neighbors pour stocker les voisins non visités de la cellule donnée. Ensuite, elle extrait les coordonnées **x et y** de la cellule.

La méthode vérifie ensuite les quatre directions possibles *(haut, droite, bas, gauche)* pour trouver les voisins non visités. Pour chaque direction, elle vérifie si la position du voisin est valide (c'est-à-dire qu'elle ne sort pas des limites de la grille) et si le voisin n'a pas encore été visité. Si ces conditions sont remplies, le voisin est ajouté au vecteur neighbors.

Après avoir vérifié toutes les directions, la méthode vérifie si le vecteur neighbors n'est pas vide. Si ce n'est pas le cas, elle sélectionne un voisin aléatoire en générant un index aléatoire randIndex et retourne le voisin correspondant.
Si le vecteur neighbors est vide, cela signifie qu'il n'y a pas de voisins non visités, et la méthode retourne nullptr.

removeWalls: La méthode commence par calculer la différence en coordonnées entre les deux cellules en utilisant les variables *dx et dy*. La variable dx est calculée comme la différence entre les coordonnées x de a et b, tandis que dy est calculée comme la différence entre les coordonnées y de a et b. Ces valeurs permettent de déterminer la position relative des deux cellules.
Ensuite, la méthode vérifie la valeur de dx pour déterminer la position de a par rapport à b.

- **Si dx est égal à 1**, cela signifie que la cellule a est à droite de la cellule b. Dans ce cas, le mur à gauche de a (indiqué par a->walls[3]) est supprimé en le définissant sur false, et le mur à droite de b (indiqué par b->walls[1]) est également supprimé.
- **Si dx est égal à -1**, cela signifie que a est à gauche de b. Dans ce cas, le mur à droite de a (indiqué par a->walls[1]) est supprimé, et le mur à gauche de b (indiqué par b->walls[3]) est également supprimé.

La méthode procède ensuite à vérifier la valeur de dy pour déterminer la position verticale des cellules.

- **Si dy est égal à 1**, cela signifie que a est en dessous de b. Dans ce cas, le mur au-dessus de a (indiqué par a->walls[0]) est supprimé, et le mur en dessous de b (indiqué par b->walls[2]) est également supprimé.
- **Si dy est égal à -1**, cela signifie que a est au-dessus de b. Dans ce cas, le mur en dessous de a (indiqué par a->walls[2]) est supprimé, et le mur au-dessus de b (indiqué par b->walls[0]) est également supprimé.

drawCell : La méthode commence par calculer les coordonnées x et y de la cellule à dessiner, en utilisant la position de la cellule (cell.x et cell.y), la taille de la cellule (cellSize), ainsi qu'un décalage (offsetX et offsetY). L'épaisseur des murs est fixée à 3 unités, et la couleur des murs est définie par la constante DEEP_PINK, qui utilise des valeurs RGBA (rouge, vert, bleu, alpha) pour une couleur rose profond.

Ensuite, la méthode vérifie la présence de chaque mur de la cellule en utilisant le tableau walls de la cellule. Si un mur est présent, la méthode utilise la fonction DrawRectangle pour dessiner un rectangle représentant ce mur avec la couleur définie. Voici comment chaque mur est dessiné :

- **Si le mur supérieur** (cell.walls[0]) est présent, un rectangle est dessiné au-dessus de la cellule.
- **Si le mur droit** (cell.walls[1]) est présent, un rectangle est dessiné à droite de la cellule.
- **Si le mur inférieur** (cell.walls[2]) est présent, un rectangle est dessiné en dessous de la cellule.
- **Si le mur gauche** (cell.walls[3]) est présent, un rectangle est dessiné à gauche de la cellule.

CLASSE PLAYER :

I.Conception:

1.Definition:

La classe Player représente le personnage joueur dans le jeu de labyrinthe. C'est ou on a encapsuler toutes les informations et les comportements liés au joueur, tels que sa position, son apparence et ses mouvements.

II.Implémentation:

2.1.ATTRIBUTS

- **x, y** : définissent les coordonnées du joueur dans la grille du labyrinthe. Ils représentent respectivement la colonne et la ligne où se trouve le joueur.
- **texture** : Attribut de type Texture2D où on stocke l'image utilisée pour représenter visuellement le joueur à l'écran. Elle est chargée à partir du fichier "player.png" lors de la création de l'objet Player.
- **hasWon** : Un booléen qui indique si le joueur a atteint la sortie du labyrinthe.

2.2.MÉTHODES

Player(int startX, int startY) : Le constructeur initialise les coordonnées de départ du joueur en fonction des paramètres *startX et startY*.
Nous charge également la texture du joueur à partir du fichier qu'on a spécifié.

void draw(int cellSize, int offsetX, int offsetY) : Cette méthode est responsable de l'affichage du joueur à l'écran.

- Elle calcule la position exacte du joueur en fonction de sa position dans la grille, de la taille des cellules et des décalages *(x et y).*
- La position du joueur en cellules (x, y) est convertie en pixels en utilisant la taille des cellules (cellSize).
- Ajoute un décalage (offsetX, offsetY) pour que le joueur soit aligné le labyrinthe dans la fenêtre.
- Divise cellSize par 2 pour centrer le joueur au milieu de la cellule.
- Elle dessine ensuite un cercle pour représenter le corps du joueur .

void move(const Maze &maze, int key) : Cette méthode gère les déplacements du joueur en fonction des touches pressées.

void move(const Maze &maze, int key) : Cette méthode gère les déplacements du joueur en fonction des touches pressées.
Le but c'est de déplacer le joueur dans le labyrinthe en fonction de la direction donnée par une touche et vérifier si le chemin est libre.

Elle vérifie si le joueur peut se déplacer dans la direction souhaitée en appelant la méthode **hasWall** de la classe Maze. Si le déplacement est possible, elle met à jour les coordonnées du joueur en conséquence.
if (key == KEY_UP && !maze.hasWall(x, y, 0)) y--;

- On vérifie en premier lieu si la touche correspondante (flèche directionnelle) a été appuyée.
- On appelle la méthode maze.hasWall pour vérifier s'il y a un mur dans la direction indiquée , sachant que les paramètres de hasWall sont :
  - *x, y : Position actuelle du joueur.*
  - *0, 1, 2, 3 : Directions respectivement pour haut, droite, bas, gauche.*
- Si aucun mur n'est présent, on met à jour les coordonnées du joueur :
  y-- pour le haut , **x++** pour la droite , **y++** pour le bas, **x--** pour la gauche.

~Player() : Le destructeur de la classe est utilisé pour libérer la mémoire occupée par la texture du joueur en appelant la fonction UnloadTexture(texture);

.

CLASSE GAME:

I.Conception:

1.Definition:

- **options:** représente une structure qui définit une option avec un rectangle (rect) et un texte (text) , Elle est utilisée pour les options de menu.
- **Jeu:** La classe Game représente le cœur de l'application et sert de point central pour gérer le déroulement global du jeu. Elle orchestre la coordination entre les différentes classes et assure la logique principale du jeu.

II.Implémentation:

1.Struct options

- **rect:** Object de type Rectangle, cette structure Rectangle est utilisée pour définir la position et la taille d'une option. Elle permet de dessiner l'option à l'écran et de gérer les interactions utilisateur, comme les clics.
- **text:** Une chaîne de caractères (std::string) qui contient le texte associé à l'option. Ce texte est généralement affiché à l'intérieur ou à côté du rectangle défini par rect. Il représente l'étiquette ou la description de l'option dans le menu.

2.Class Jeu

2.1.ATTRIBUTS

- **maze :** Objet de type Maze qui représente le labyrinthe du jeu.
- **player :** Objet de type Player qui représente le joueur du jeu.
- **screenWidth et screenHeight :** Entiers représentant respectivement la largeur et la hauteur de l'écran.
- **cellSize :** Taille des cellules du labyrinthe, utilisée pour calculer les dimensions graphiques.
- **offsetX et offsetY :** Décalages horizontal et vertical pour positionner le labyrinthe sur l'écran.
- **font1 :** Objet de type Font, utilisé pour dessiner du texte à l'écran.

- **etat_jeu :** L'énumération Etat représente les différentes phases du jeu (accueil, menu, transition, partie en cours, fin).
- **partie_active :** Booléen indiquant si une partie est en cours.
- **haswon :** Booléen indiquant si le joueur a gagné la partie.
- **transitionframes :** Compteur d'images utilisé pour gérer les transitions entre les états du jeu.
- **framestowait :** Constante définissant le nombre d'images à attendre lors des transitions.
- **countdown :** Compteur utilisé pour des décomptes avant que le jeu commence.
- **chronometre :** Chronomètre en secondes pour suivre le temps écoulé pendant le jeu.
- **levels et endoptions :** Vecteurs de options pour gérer les niveaux et les options de fin de jeu.
- **Couleurs :** Constantes définissant différentes couleurs utilisées dans le jeu **(LIGHT_PINK, HOT_PINK, DEEP_PINK, PEACH_PUFF).**

1.2.Méthodes

Le constructeur: Jeu::Jeu**(int width, int height, int cellSize)** initialise un objet de la classe Jeu en configurant les attributs du labyrinthe, du joueur, des dimensions de l'écran, et des états du jeu.
Il place le joueur au centre gauche du labyrinthe, fixe les dimensions de l'écran et les décalages, et initialise des états comme le chronomètre et les indicateurs de victoire.
De plus, il charge une police de caractères et configure les options de niveaux du jeu avec des positions et étiquettes spécifiques. En résumé, il prépare l'objet Jeu pour le déroulement du jeu en initialisant tous les attributs nécessaires.

Void gerer_evenements () : Cette fonction est l'élément central de la gestion des événements dans notre jeu. Elle est responsable de la gestion des différents états du jeu, du contrôle des entrées utilisateur et de l'affichage des éléments interactifs à l'écran.
Elle commence par une série de vérifications sur l'état actuel du jeu, représenté par la variable etat_jeu. Selon l'état, la fonction exécute différentes actions. Voici les états gérés par la fonction :

- **START :** L'écran d'accueil, où l'utilisateur peut appuyer sur un bouton pour commencer le jeu.

Ici La fonction commence par récupérer la position actuelle de la souris à l'écran.
Ensuite, elle crée un bouton "Start" sous forme de rectangle avec des coins arrondis et affiche le texte "Start" au centre.
Un contrôle de collision est effectué pour vérifier si la souris se trouve sur le bouton. Si tel est le cas, le bouton est mis en surbrillance avec une bordure rose foncée pour indiquer qu'il est sélectionné. Si le joueur clique sur le bouton (avec le bouton gauche de la souris), l'état du jeu est modifié et passe à MENU, ce qui permet de démarrer la prochaine phase du jeu.

- **MENU :** L'écran de menu principal, où un menu de jeu est affiché.

Si l'état est MENU, la fonction drawmenu() est appelée pour afficher l'interface du menu du jeu. Cette fonction gère l'affichage des différents éléments du menu, permettant au joueur de naviguer et de choisir parmi les options disponibles, comme commencer une nouvelle partie, accéder aux réglages, ou quitter le jeu.

- **TRANSITION :** Si l'état est TRANSITION, la fonction preparetoi() est appelée pour dire annoncer que la partie commencera.

- **EN_COURS :** L'état pendant que le jeu est en cours.

Création des boutons "Home" et "Replay" : Des boutons sont créés pour retourner au menu ("Home") et recommencer la partie ("Replay"). Ils sont placés respectivement dans le coin inférieur gauche et le coin inférieur droit.

- Si le bouton "Home" est cliqué, l'état du jeu passe à MENU.
- Si le bouton "Replay" est cliqué, la fonction reinitialiser_partie() est appelée pour redémarrer le jeu.

Le temps écoulé depuis le début du jeu est affiché en bas de l'écran. Le temps est mis à jour chaque frame avec chronometre += GetFrameTime().
Si les touches fléchées sont pressées, le joueur se déplace dans le labyrinthe, et la fonction verifierVictoire() est appelée pour vérifier s'il a gagné.

- **FIN :** L'écran de fin de jeu, où le joueur voit son temps final.

Lorsque le jeu est terminé, le texte **"You won!"** est affiché avec le temps final du joueur. Ce temps est calculé et affiché en haut de l'écran.
Et finalement en fait appelle a la fonction replay() pour donner a l'utilisateur le choix de rejouer ,afficher **menu ou quitter le jeu**.

Void VerifierVictoire (): Cette méthode est utilisée pour déterminer si le joueur a atteint la case de fin du jeu.

- Elle vérifie si les coordonnées du joueur (player.x et player.y) correspondent exactement aux coordonnées de la case de fin.
- Si le joueur est sur la case de fin, les lignes suivantes s'exécutent (haswon = true) ,cette ligne indique qu'une victoire a été obtenue.
- etat_jeu = FIN; : Cette ligne change l'état du jeu à "FIN", ce qui signifie que le jeu est terminé.
- countdown = 3; : Cette ligne initialise un compte à rebours de 3 secondes (probablement pour afficher un message de victoire avant de revenir au menu principal).

Void drawmenu (): gère l'affichage et l'interactivité du menu principal du jeu. Elle commence par détecter la position de la souris avec **GetMousePosition,** puis dessine chaque bouton du menu à l'aide de **DrawRectangleRounded** avec une couleur par défaut (LIGHT_PINK) et affiche le texte correspondant à l'intérieur. Si la souris survole un bouton **(CheckCollisionPointRec),** une bordure en couleur (HOT_PINK) est dessinée pour indiquer l'interactivité.

Lorsqu'un bouton est cliqué (IsMouseButtonPressed), l'état du jeu (etat_jeu) passe en TRANSITION, le compteur de transition est réinitialisé, et la taille des cellules du labyrinthe (cellSize) est ajustée en fonction du bouton sélectionné, permettant à l'utilisateur de choisir le niveau de difficulté (facile, moyen ou difficile). Cette méthode combine affichage visuel, gestion des événements et configuration des paramètres du jeu.

Void Preparetoi ():
Cette méthode est utilisée pour gérer l'animation et la transition avant le début d'une nouvelle partie :

- La méthode commence par afficher un message **"Get Ready!"** à l'écran en utilisant la police font1, avec une taille de 55, un espacement de 2, et la couleur LIGHT_PINK. Le message est positionné à des coordonnées spécifiques ({180, 350}) sur l'écran.

- Ensuite, un compte à rebours (countdown) est affiché au centre de l'écran, utilisant également la couleur LIGHT_PINK. La position du texte est calculée pour être centrée horizontalement (screenWidth / 2 - 15) et placé à 240 pixels du haut de l'écran.
- La méthode incrémente transitionframes et réduit le compte à rebours **(countdown)** de 1 chaque fois que transitionframes atteint un multiple de framestowait.
- Lorsque le compte à rebours atteint zéro ou moins, l'état du jeu (etat_jeu) est mis à jour pour passer à EN_COURS, indiquant que la partie commence. La méthode demarrer_partie est alors appelée pour initialiser la nouvelle partie, et le chronomètre (chronometre) est réinitialisé à zéro pour commencer à mesurer le temps de jeu.

Void demarrer_partie () : Cette méthode sert à initialiser une nouvelle partie du jeu

- Elle calcule les dimensions exactes du labyrinthe en fonction de la taille de l'écran et de la taille des cellules.
- Elle détermine l'emplacement où le labyrinthe sera dessiné sur l'écran.
- Elle crée un nouvel objet Maze avec les dimensions calculées précédemment. Cet objet contient toutes les informations sur la structure du labyrinthe (murs, passages).
- Elle place le joueur à sa position de départ, généralement dans un coin du labyrinthe.
- Elle met à zéro le chronomètre pour commencer à compter le temps de jeu.
- Elle affiche sur l'écran les premières informations du jeu, comme le temps écoulé.

Void reinitialiser_partie () : La méthode reinitialiser_partie de la classe Jeu sert à réinitialiser la partie en cours en redémarrant une nouvelle partie.

Void replay (): Cette méthode gère l'affichage d'un menu interactif à la fin du jeu, permettant au joueur de choisir entre retourner au menu principal, redémarrer la partie, ou quitter le jeu.

-Elle commence par obtenir la position actuelle de la souris avec GetMousePosition() pour détecter l'interaction utilisateur.
-Un vecteur endoptions contient les trois boutons du menu (Home,Restart,Quit) ,chacun défini par :

- Un rectangle (rect) pour la zone cliquable.
- Un texte (text) pour le label du bouton.

-Pour chaque option dans le vecteur :

- Le bouton est dessiné avec un rectangle arrondi (DrawRectangleRounded).
- Le texte correspondant est affiché avec DrawTextEx, correctement positionné à l'intérieur du bouton.

-Si la souris survole un bouton (détecté via CheckCollisionPointRec), le rectangle est entouré d'une bordure visuelle (HOT_PINK) pour indiquer qu'il est sélectionné.
-Si le bouton est cliqué (via IsMouseButtonPressed(MOUSE_LEFT_BUTTON)), une action est exécutée :

- **"Home"** : Retourne au menu principal en mettant etat_jeu = MENU.
- **"Restart"** : Lance une transition pour redémarrer la partie en mettant etat_jeu = TRANSITION.
- **"Quit"** : Ferme immédiatement la fenêtre du jeu avec CloseWindow().

Void boucle_principale () :représente la boucle principale du jeu, gérant les événements, l'affichage et la mise à jour des éléments. Elle commence par initialiser la fenêtre du jeu avec les dimensions spécifiées (screenWidth et screenHeight) et fixe le nombre d'images par seconde à 60 avec SetTargetFPS.

Dans la boucle infinie (tant que la fenêtre n'est pas fermée), elle nettoie l'écran avec la couleur d'arrière-plan définie (PEACH_PUFF) et appelle la méthode gerer_evenements pour traiter les interactions utilisateur.

Si l'état du jeu (etat_jeu) est défini comme EN_COURS, la méthode met à jour le chronomètre (chronometre) en ajoutant le temps écoulé depuis le dernier cadre, dessine le labyrinthe (maze.draw()), affiche le joueur (player.draw) en fonction de la taille des cellules et des décalages, puis dessine une texture (par exemple, une image décorative).

Enfin, la méthode termine chaque itération de la boucle avec EndDrawing et, après la sortie de la boucle, ferme la fenêtre avec CloseWindow.

main():

La fonction main est le point d'entrée de l'application.Elle permet de définir les dimensions de la fenêtre et la taille des cellules du labyrinthe.
Ensuite, un objet Jeu est créé, en lui passant ces paramètres pour configurer le jeu. Enfin, la fonction boucle_principale() de l'objet Jeu est appelée, ce qui lance la logique principale du jeu, incluant la gestion des événements et l'affichage.

Résultats et Mécanismes du Jeu

Notre jeu de labyrinthe offre une expérience interactive où le joueur peut choisir le niveau de difficulté avant de commencer. Pendant le jeu, le temps écoulé est calculé en continu, permettant au joueur de suivre ses progrès. Si le joueur se retrouve bloqué, il a la possibilité de recommencer le niveau depuis le début. Une fois le niveau terminé, un message de victoire est affiché, et le joueur peut choisir de rejouer le même niveau, retourner au menu principal pour sélectionner un autre niveau ou quitter le jeu. Cette flexibilité garantit une expérience fluide et engageante, tout en permettant au joueur de personnaliser son parcours et de tester ses compétences à différents niveaux.

Conclusion

En utilisant les principes fondamentaux de la programmation orientée objet (POO) en C++, ainsi que la bibliothèque graphique Raylib, ce projet a permis de développer un jeu interactif .Grâce à la génération dynamique de labyrinthes et à l'intégration de trois niveaux de difficulté, l'expérience de jeu offre une progression adaptée à divers profils de joueurs, tout en mettant en avant les concepts de conception procédurale.