Université Abdelmalek Essaadi
Faculté des Sciences et Techniques de Tanger

PROJET

DEVELOPPEMENT D'UN JEU DE CASSE-TETE DE STYLE LABYRINTHE (MAZES) EN C++ AVEC RAYLIB

- AIT SIOUI SALMA
- ZOUJARI IDRISSI AYA
- FARHAOUI AYOUB
- BENHAMMOU MOHAMED

Professeur :

Ikram Ben Abdel Ouahab

Présentation générale du projet :

Ce projet consiste en la conception et le développement d'un jeu de casse-tête de type labyrinthe, réalisé en programmation orientée objet (POO) avec le langage C++ et la bibliothèque graphique Raylib. Ce jeu propose une expérience immersive où le joueur guide un personnage à travers un labyrinthe généré aléatoirement, afin de retrouver son chemin. Avec trois niveaux de difficulté et une ambiance sonore engageante, notre projet se distingue par sa combinaison de défis techniques et d'une narration captivante.

Motivation pour le choix du projet :

L'idée derrière ce projet est de créer une expérience ludique et mémorable en intégrant une dimension narrative. L'histoire du jeu met en scène un garçon qui s'est perdu dans un labyrinthe mystérieux en essayant de rentrer chez lui. Le joueur est appelé à l'aider dans cette quête, renforçant ainsi son engagement.

Pour immerger davantage le joueur, une introduction dramatique a été conçue, avec un texte et une musique qui posent une ambiance émotive dès le début. Lorsque le joueur atteint l'objectif et trouve la maison, un son chaleureux de bienvenue, **"Welcome Home"**, vient marquer la fin du jeu, offrant une conclusion satisfaisante et réconfortante à l'histoire. Ces éléments sonores jouent un rôle clé dans l'immersion, enrichissant l'expérience globale

Présentation des technologies utilisées :

- **C++**
- **Raylib** : bibliothèque graphique intuitive offre un cadre simple et efficace pour la création de jeux 2D
- **Modele de programmation orientée objet(POO)**

Spécifications Fonctionnelles :

1. Génération du labyrinthe

2. Mécaniques de jeu

3. Interface graphique

4. Ambiance sonore

5. Narration

6. Système de sauvegarde et scores

7. Réinitialisation et rejouabilité

Analyse du code source

1. ***Inclusion des bibliothèques nécessaires pour le développement graphique et mathématique.***

- raylib.h : Fournit les fonctions pour le graphisme, l'audio et l'interaction utilisateur.
- raymath.h : Fournit des fonctions mathématiques supplémentaires comme la gestion des vecteurs.

2. ***Pour définir la largeur et la hauteur de la fenêtre graphique (1280x720 pixels dans ce cas).***

```
#define SCREEN_WIDTH 1280
#define SCREEN_HEIGHT 720
```

3. **Définiton d'une constante de couleur pour le menu. Ici, RAYWHITE est un blanc avec une légère teinte grise.**

```
#define MENU_COLOR RAYWHITE
```

4. **Pour une sécurité de type et lisibilité accure on a utiliser ces trois enumération**:

```
enum class GameState { MENU, PLAYING, FINISHED };
```

Description : Cette énumération représente les différents **états globaux** dans lesquels le jeu peut se trouver.

Valeurs possibles :

- MENU
- PLAYING
- FINISHED

5. **Représentation des différents** niveaux de difficulté **du jeu.**

```
enum class Difficulty { EASY, MEDIUM, HARD };
```

6. **États spécifiques de l'écran de menu du jeu**

```
enum class MenuState { STORY, LEVEL_SELECT };
```

7. **Création d' une fenêtre avec les dimensions spécifiées et le titre "Maze Game".**

```
InitWindow(SCREEN_WIDTH, SCREEN_HEIGHT, "Maze Game");
```

8. **Activation du système audio pour permettre la lecture des sons et musiques.**

```
InitAudioDevice();
```

9. ***Définition de la fréquence d'images en* Fixant la limite de rafraîchissement à 60 images par seconde**

```
SetTargetFPS(60);
```

10. **Chargement d'une image de fond à partir d'un fichier (dd.jpg) pour l'utiliser comme texture.**

```
Texture2D backgroundTex =
LoadTexture("resources/dd.jpg");
```

11. **Chargement de la musique**

```
Sound menuMusic = LoadSound("resources/AA.mp3");
SetSoundVolume(menuMusic, 0.5f);
```

- **LoadSound :** Charge une musique de menu.et meme chose pour les autes sons
- **SetSoundVolume :** Définit le volume de cette musique pour chaque son correspondant

12. **Initialisation des variables pour le menu**

```
bool gameStarted = false;
float menuTimer = 0.0f;
```

- **gameStarted :** Indique si le jeu a démarré (false = non démarré).
- **menuTimer :** Garde le temps écoulé dans le menu, utilisé pour des animations ou des transitions.

Déclaration de la variable :

- **std::vector<StoryText>** est un conteneur utilisé pour stocker une collection d'objets de type **StoryText.**

Déclaration d'entier:

```
int currentStoryText = 0;
```

Cette variable est utilisée pour garder une trace de l'index du texte d'histoire actuel affiché.

```
std::unique_ptr<Maze> maze:
```

Pointeur unique: Utilisé pour gérer la mémoire du labyrinthe de manière plus sécurisée, garantissant que la mémoire est libérée automatiquement lorsqu'elle n'est plus utilisée.

```
std::unique_ptr<Player> player:
```

- **Pointeur unique**: Utilisé pour gérer la mémoire du joueur.

13. Commande pour jouer la musique du menu

Fonction PlaySound:

```
while (!WindowShouldClose()) {
 BeginDrawing();
 ClearBackground(RAYWHITE);
 DrawTexture(backgroundTex, 0, 0, WHITE);
```

La boucle **while** s'exécute tant que l'utilisateur n'a pas encore fermé la fenêtre.

BeginDrawing() est une fonction qui existe dans la bibliothèque **Raylib** dans le module : **rcore** : ***void BeginDrawing(void)*** , sa fonctionnalité est de configurer le canevas pour commencer à dessiner.

ClearBackground() est une fonction qui existe aussi dans la bibliothèque Rayib dans le module : **rcore** : ***void ClearBackground(Color color),*** elle sert à définir la couleur d'arrière-plan.

RAYWHITE est une couleur qui existe dans le bibliothèque **Raylib** dans **colors** : ***#define RAYWHITE(Color){ 245, 245, 245, 255 }.***

DrawTexture() est une fonction qui existe dans la bibliothèque **Raylib** dans le module : **rtextures** *: void DrawTexture(Texture2D texture, int posX, int posY, Color tint)* , elle sert a dessiner une Texture2D. Une texture est une image qui a été chargée en mémoire GPU et qui peut être utilisée pour le rendu graphique dans ce cas est backgroudTex qui est associé à le path du fond d'écran qu'on a utilisé, posX et posY sont les positions horizontale et verticale en pixels dont la texture sera dessinée dans notre cas c'est à la position (0,0) qui correspond au coin supérieur à gauche. Pour tint ce paramètre permet d'appliquer une teinte à la texture lors de son dessin, elle est définie par un objet de type Color, qui peut modifier la couleur d'affichage de la texture, on a utilisé WHITE pour affichée la texture sans modification.

```
if (!IsSoundPlaying(menuMusic) && gameState ==
GameState::MENU) {
        PlaySound(menuMusic);
    }
```

If statement ici vérifie si la la musique n'est pas en cours de lecture en utilisant **IsSoundPlaying()** qui est une fonction de la bibliothèque **Raylib** dans le module : **raudio** : *bool IsSoundPlaying(Sound sound);* , elle est utilisée pour vérifier si un son (menuMusic) est actuellement en cours de lecture. Aussi si gameState qui représente les différents états pour passer d'une phase à l'autre du jeu est dans l'état MENU. Si c'est deux conditions sont vérifié le programme va commencer à joué la musique avec **PlaySound()** qui est une fonction de la bibliothèque **Raylib** dans le module : **raudio** : *void PlaySound(Sound sound);* , elle est utilisée pour jouer un son.

```
switch(gameState)
```

Dans cette partie on va traité chaque état du jeu et implémenté le logique du jeu principale en utilisant switch entre les différents états de gameState : MENU, PLAYING, FINISHED.

```
case GameState::MENU: {
            ClearBackground(MENU_COLOR);
```

Dans un premier lieu l'état MENU :

On utilise ClearBackground () pour définir la couleur de l'arrière-plan dans ce cas c'est MENU_COLOR la constante de l'objet Color qu'on a déjà défini avant pour la couleur black.

```
if (menuState == MenuState::STORY) {
                menuTimer += GetFrameTime();
```

Maintenant if menuState qui est un objet (sous-état) de la classe MenuState égale à STORY, le compteur menuTimer qu'on a déjà déclaré au début et initialisé à 0 va être incrémenté par la valeur retourné de la fonction GetFrameTime() qui appartient à la bibliothèque Raylib du module : rcore : float GetFrameTime(void); , elle utilisée pour obtenir le temps écoulé, en secondes, depuis le dernier cadre dessiné. Cela correspond à ce qu'on appelle souvent le delta time, qui est essentiel pour gérer les animations et les mouvements dans les jeux de manière indépendante du taux de rafraîchissement. L'utilisation de cette dernière dans notre code c'est de crée l'effet de l'apparition du texte histoire qu'on va voir après.

```
for (size_t i = 0; i < storyTexts.size(); i++) {
if (storyTexts[i].isVisible) {

   storyTexts[i].alpha = fmin(1.0f,
storyTexts[i].alpha + TEXT_FADE_SPEED *
GetFrameTime());
```

Dans la boucle for on déclare un index de type size_t, size_t est un type entier non signé (toujours positif), utilisé pour représenter des tailles et des indices et défini dans les bibliothèques standard C++ aui garanti de pouvoir stocker la taille maximale de n'importe quel objet en mémoire comme il est recommandé pour l'indexation de conteneurs comme les vecteurs, tableaux.

Puisque on veut afficher le texte du storyTexts qui est à la base un vecteur il faut utilisé ce type d'indexation, pour i égale zéro jusqu'à qu'il soit inférieur à la taille de storyTexts (storyTexts.size() retourn un size_t à son tour) et on incrémente par 1. On vérifie si storyTexts de l'élément à la position i doit être affiché seule les texts marqués true vont seront traités.

La propriété alpha détermine la transparence de l'élément storyTexts à la position déterminer, où 0.0 représente complètement transparent et 1.0 complètement opaque. Pour TEXT_FADE_SPEED c'est une constante qui définit la vitesse à laquelle le texte doit devenir visible (ou disparaître). Plus cette valeur est élevée, plus le changement de transparence se fera rapidement. On multiplie cette dernière avec GetFrameTime() pour obtenir une valeur qui représente l'incrément de transparence à appliquer pour ce cadre spécifique, rendant l'animation fluide et dépendante du temps.

Pour le fmin c’est une fonction qui retourne la plus petite des deux valeurs passées en argument. Dans ce cas, elle compare 1.0f (complètement visible) avec storyTexts.apha + la mutliplication qu’on a expliqué précédemment. Cette fonction garantit que la valeur de alpha ne dépasse jamais 1.0. La ligne complète met à jour la valeur alpha de l'élément de texte à l'index i. Elle augmente progressivement cette valeur en fonction du temps écoulé depuis le dernier cadre.

```
Color textColor = MENU_TEXT_COLOR;
   textColor.a = (unsigned char)(255 *
storyTexts[i].alpha);
   int textWidth = MeasureText(storyTexts[i].text,
30);
   DrawText(storyTexts[i].text, SCREEN_WIDTH/2 -
textWidth/2, (int)storyTexts[i].y, 35, textColor);
                    }
                }
```

D’abord on initialise textColor to MENU_TEXT_COLOR qui est la couleur blanche.

Dans la deuxième ligne, le a est un propriété de la structure Color qui determine la transparence de la couleur. Dans Raylib la transparence est souvent représentée par un nombre flottant compris entre 0.0 et 1.0. Une valeur de 0.0 signifie complètement transparent, tandis qu'une valeur de 1.0 signifie complètement opaque. Cependant, les canaux alpha dans la structure Color sont généralement stockés sous forme d'entiers non signés (type unsigned char), qui acceptent des valeurs entières allant de 0 à 255. Cela permet une représentation plus précise et plus efficace de la transparence dans le rendu graphique.

Ensuite on mesure la taille du texte pour le faire centré en utilisant la fonction **MeasureText()** qui se trouve dans la bibliothèque **Raylib** dans le module **rtext** : ***int MeasureText(const char *text, int fontSize) ;*** , char*text est le paramètre de la chaîne de caractères dont on souhaite mesurer sa largeur et fontSize est la taille de la police à utiliser pour mesurer le texte.

DrawText() représente une fonction de la library **Raylib** dans le module **rtext** : ***void DrawText(const char *text, int posX, int posY, int fontSize, Color color);*** , est utilisée pour dessiner du texte à l'écran. Le premier paramètre représente le texte à afficher, le deuxième spécifie la position horizontale (en pixels) sur l'écran où le texte sera dessiné le choix de SCREEN_WIDTH/2 - textWidth/2 va nous permettre de positionner le texte au centre horizontal de l'écran , le troisième définit la position verticale (en pixels) sur l'écran où le texte sera

dessiné (int)storyTexts[i].y contient la coordonnée Y pour cet élément de texte spécifique, le cast (int) garantit que la valeur est un entier, ce qui est requis par la fonction DrawText(), après fontSize indique la taille de la police en pixels pour le texte dessiné et dernièrement c'est la couleur du texte à dessiner.

```
if (menuTimer >= 2.0f &&
static_cast<std::vector<StoryText>::size_type>(curre
ntStoryText) < storyTexts.size() - 1) {
   currentStoryText++;
   storyTexts[currentStoryText].isVisible = true;
   menuTimer = 0;
}
```

Toujours dans le même if qui définit l'état STORY, on check si menuTimer dépasse ou soit égale à 2.0f qui est l'équivalent à 2 secondes et si currentStoryText est inférieur à la taille du vecteur storyTexts -1, cela signifie que l'on veut s'assurer qu'il y a encore des éléments à afficher dans le vecteur avant d'essayer d'accéder au prochain élément. En soustrayant 1, on évite d'accéder à un index qui serait hors limites, ce qui pourrait provoquer un comportement indéfini ou un plantage du programme.

Pour cette partie static_cast<std::vector<StoryText>::size_type> représenter les indices et les tailles des vecteurs de manière sécurisée et portable. Cela garantit que le type est compatible avec les indices du vecteur, ce qui est important pour éviter des erreurs de type.

On incrémente currentStoryText pour afficher les autres lignes de l'histoire si les deux conditions sont vérifiées. Ensuite on met à jour la propriété isVisible de l'élément courant dans le vecteur storyTexts, en le définissant sur true. Cela indique que ce texte doit être affiché à l'écran.

Et finalement on réinitialise menuTimer à zéro pour qu'on fait la même opération pour l'élément suivant.

```
if (IsKeyPressed(KEY_SPACE)) {
                    menuState =
MenuState::LEVEL_SELECT;
                    menuTimer = 0;
                }
```

Cette if statement vérifie si la touche espace à été appuyer par l'utilisateur ou pas, IsKeyPressed() est une fonction bool de la bibliothèque Raylib du module **rcore** : ***bool IsKeyPressed(int key);***

Si c'est true on va passer à l'état suivant qui est choisir quel niveau (LEVEL_SELECT). On réinitialise le compteur menuTimer au cas ou notre programme n'est pas vérifié les deux conditions de if précédente.

Ici ça termine le premier if qui vérifié si on est dans le sous état du menu qui est histoire.

```
else {
       const char* title = "Choose Your Path";
DrawText(title, SCREEN_WIDTH/2 - MeasureText(title,
45)/2, 200, 70, MENU_TEXT_COLOR);
```

Maintenant on passe au deuxième sous état qui est choisir le niveau LEVEL_SELECT.

D'abord title est déclaré comme un pointeur vers un tableau de caractères qui est la chaîne littérale "Choose Your Path", comme déjà vu la fonction **DrawText()** utilise comme un premier paramètre un pointeur de type char. De plus en C++ les chaînes de caractères sont en réalité des tableaux de caractères terminés par un caractère nul ('\0'), les chaînes littérales sont généralement stockées dans une section spéciale de la mémoire et le pointeur permet d'accéder à cette mémoire sans avoir à copier la chaîne entière dans une nouvelle variable, cela permet également d'économiser de l'espace mémoire. Le type const char* indique que le contenu pointé ne doit pas être modifié.

Après La fonction MeasureText(title, 45) mesure la largeur du texte pour centrer correctement le titre horizontalement sur l'écran. La position Y est fixée à 200 pixels, et la taille de la police est spécifiée comme 70 pixels , et pour la couleur elle est définie par MENU_TEXT_COLOR (blanche).

```
DrawText("1 - A Gentle Start ",
                        SCREEN_WIDTH/2 - 280, 400,
50, MENU_TEXT_COLOR);
                DrawText("2 - The Winding Path ",
                        SCREEN_WIDTH/2 - 260, 500,
50, MENU_TEXT_COLOR);
                DrawText("3 - The Grand Maze ",
SCREEN_WIDTH/2 - 240, 600, 50, MENU_TEXT_COLOR);
```

Dans la fenêtre on affiche les différents level chaque texte occupe 100 pixel et puisque la taille de police est 50 pixels donc chaque ligne va occupée un peut près de 50 pixel de hauteur donc la distance entre chaque ligne sera inférieur à 50 pixel. Aussi on a essayé de décaler chaque ligne par 20 pixels de largeur.

```
if (IsKeyPressed(KEY_ONE) || IsKeyPressed(KEY_TWO)
|| IsKeyPressed(KEY_THREE)) {

selectedDifficulty = IsKeyPressed(KEY_ONE) ?
Difficulty::EASY :
IsKeyPressed(KEY_TWO) ? Difficulty::MEDIUM :
Difficulty::HARD;
```

Pour sélectionné le niveau de difficulté on on utilise if statement IsKeyPressed pour chaqu'un des trois buttons pour voire si un des trois à été sélectionné par l'utilisateur, après on utilise ternary operator pour précisé qu'elle choix on va traité après.

Si selectedDifficulty est KEY_ONE le niveau de difficulté est EASY, Si c'est KEY_TWO c'est MEDIUM sinon c'est HARD.

```
int mazeSize = (selectedDifficulty ==
Difficulty::EASY) ? 13 :
(selectedDifficulty == Difficulty::MEDIUM) ? 15 :
20;
```

Ici on définit la taille du labyrinthe selon le choix qui a été pris par l'utilisateur, toujours en utilisant ternary operator si c'est le premier niveau alors il est defini à 13 cellules dans une dimension, si c'est le deuxième choix il va être défini sur 15 sinon c'est 20.

```
maze = std::make_unique<Maze>(mazeSize);
```

Cette ligne crée une nouvelle instance de la classe Maze, en utilisant la taille déterminée précédemment (mazeSize). L'utilisation
de std::make_unique permet de gérer automatiquement la mémoire avec un pointeur unique.

```
player = std::make_unique<Player>(CELL_SIZE * 1.5f,
CELL_SIZE * 1.5f);
```

Ici, une nouvelle instance de la classe Player est créée. Les paramètres passés au constructeur sont calculés en multipliant CELL_SIZE par 1.5, ce qui va définir la position initiale du joueur dans le labyrinthe.

```
gameState = GameState::PLAYING;
gameStartTime = GetTime();
                            }      }
break;
        }
```

La première ligne indique qu'il y a un changement d'état, le jeu a commencé et que le joueur peut interagir avec le labyrinthe. Pour **GetTime()** c'est un fonction de **Raylib** dans le module **rcore** : ***double GetTime(void);*** , qui sert a obtenir le temps en secondes depuis le début du programme et qui va être stockée dans gameStartTime pour obtenir après un score.

La première case qui est liée au Menu est terminé.

```
case GameState::PLAYING: {
            bool moved = player->Update(*maze);
```

Maintenant pour le deuxième cas si l'état du jeu est PLAYING.

Après la variable bool définie par la méthode Update de l'objet player, en lui passant le labyrinthe. Cette méthode met à jour la position du joueur en fonction des entrées (les touches de directions) et retourne un booléen true or false indiquant si le joueur a effectivement bougé.

```
if (moved) {
                PlaySound(stepSound);
            }
```

Cette ligne permet d'avoir l'effet du son si le joueur a bougé.

```
int mazeSize = maze->GetSize();
```

Appelle la méthode GetSize() sur l'objet maze pour obtenir la taille actuelle du labyrinthe.

```
int mazePixelSize = mazeSize * CELL_SIZE;
            int offsetX = (SCREEN_WIDTH -
mazePixelSize) / 2;
            int offsetY = (SCREEN_HEIGHT -
mazePixelSize) / 2;
```

La première ligne calcule la taille totale en pixels du labyrinthe en multipliant la taille en cellules par CELL_SIZE, qui représente la taille d'une cellule en pixels.

La deuxième ligne calcule le décalage horizontal nécessaire pour centrer le labyrinthe sur l'écran, et la troisième le décalage vertical nécessaire pour centrer le labyrinthe sur l'écran.

```
BeginMode2D((Camera2D){
.offset = {(float)offsetX, (float)offsetY},
.target = {0, 0},
.rotation = 0,
.zoom = 1.0f
            });
```

La fonction **BeginMode2D()** appartient à la bibliothèque **Raylib** dans le module **rcore** : ***void BeginMode2D(Camera2D camera);*** , elle démarre un mode de dessin 2D. Les paramètres définissent comment la caméra doit se comporter. Pour .offset Définit où la caméra doit commencer à dessiner, basé sur les décalages calculés précédemment. Pour .target définit le point cible que la caméra regarde, et .rotation indique qu'il n'y a pas de rotation appliquée à la caméra et pour .zoom définit un zoom normal.

```
maze->Draw();


int exitX = (mazeSize-2) * CELL_SIZE;
int exitY = (mazeSize-2) * CELL_SIZE;

DrawRectangle(exitX, exitY + CELL_SIZE/3, CELL_SIZE,
2*CELL_SIZE/3, HOUSE_COLOR);
```

La première ligne appelle la méthode Draw() sur l'objet maze, qui dessine visuellement le labyrinthe à l'écran.

Après on déclare deux entiers exitX et exitY qui vont calculer la position X et Y de la sortie du labyrinthe.

Ensuite on dessine le rectangle qui est associé à la fin du jeu, **DrawRectangle()** appartient à la bibliothèque **Raylib** dans le module **rshapes** : ***void DrawRectangle(int posX, int posY, int width, int height, Color color);*** , la posX est défini par exitX pour avoir le labyrinthe du level hard entouré exactement par le coin supérieur gauche fenêtre. Pour posY c'est la coordonnée Y du coin supérieur gauche du rectangle. Ici, exitY a également été calculé comme (mazeSize - 2) * CELL_SIZE, représentant la position Y de la sortie, en ajoutant CELL_SIZE / 3, on déplace le rectangle un tiers de la taille d'une cellule vers le bas par rapport à exitY. Cela signifie que le rectangle qui représente la maison ne commence pas exactement au niveau de la sortie, mais un peu plus bas, ce qui donne une apparence plus réaliste en laissant de l'espace pour le toit, et pour CELL_SIZE il représente la largeur du rectangle en l'utilisant on s'assure que la maison a une largeur égale à celle d'une cellule dans le labyrinthe.

Pour height ce paramètre représente la hauteur du rectangle on définit la hauteur de la maison à deux tiers de la taille d'une cellule.

Pour color dans ce cas est la couleur utilisée pour dessiner le rectangle représentant la maison on l'a déjà défini comme const dans maze.h.

```
Vector2 roofTop = {(float)(exitX + CELL_SIZE/2.0f),
(float)exitY};
Vector2 roofLeft = {(float)exitX, (float)(exitY +
CELL_SIZE/3.0f)};
Vector2 roofRight = {(float)(exitX + CELL_SIZE),
(float)(exitY + CELL_SIZE/3.0f)};
```

Vector2 roofTop crée un vecteur 2D qui représente un point dans l'espace 2D. Le Vector2 est généralement une structure ou une classe qui contient deux coordonnées : X et Y.

- {(float)(exitX + CELL_SIZE/2.0f), (float)exitY} définit les coordonnées de ce vecteur.
- (float)(exitX + CELL_SIZE/2.0f) calcule la coordonnée X du sommet du toit. Il est positionné au milieu de la largeur de la maison, donc exitX est augmenté de la moitié de CELL_SIZE. Cela place le sommet du toit au centre horizontal de la maison. Le cast (float) garantit que la valeur est un float.
- (float)exitY : Utilise la même coordonnée Y que celle de la sortie, ce qui signifie que le sommet du toit est situé directement au-dessus de la maison. De même le cast (float) garantit que la valeur est un float.

Vector2 roofLeft crée un vecteur 2D pour représenter le coin gauche du toit.

- {(float)exitX, (float)(exitY + CELL_SIZE/3.0f)} définit les coordonnées de ce vecteur.
- (float)exitX le coordonnée X est le même que celle de la maison, donc le coin gauche est aligné avec le côté gauche de la maison.
- (float)(exitY + CELL_SIZE/3.0f) le coordonnée Y est augmentée d'un tiers de CELL_SIZE, ce qui déplace le coin gauche du toit vers le haut, au-dessus de la maison.

Vector2 roofRight crée un vecteur 2D pour représenter le coin droit du toit.

- {(float)(exitX + CELL_SIZE), (float)(exitY + CELL_SIZE/3.0f)} définit les coordonnées de ce vecteur.
- (float)(exitX + CELL_SIZE) : La coordonnée X est augmentée d'une cellule complète, plaçant le coin droit à droite de la maison.
- (float)(exitY + CELL_SIZE/3.0f) comme pour le coin gauche, cette coordonnée Y est également augmentée d'un tiers de CELL_SIZE, ce qui aligne le coin droit à la même hauteur que le coin gauche.

```
DrawTriangle(roofTop, roofLeft, roofRight, RED);
DrawRectangle(exitX + CELL_SIZE/3, exitY +
CELL_SIZE/2,CELL_SIZE/3, CELL_SIZE/2, DARKBROWN);
DrawCircle(exitX + 2*CELL_SIZE/3 - 5, exitY +
3*CELL_SIZE/4, 3, GOLD);
```

DrawTriangle() est une fonction de **Raylib** du module **rshapes** : ***void DrawTriangle(Vector2 v1, Vector2 v2, Vector2 v3, Color color);*** , cette fonction dessine un triangle sur l'écran en utilisant les trois points définis précédemment. Les points passés à cette fonction sont roofTop qui indique le sommet du toit, roofLeft le coin gauche du toit, roofRight le coin droit du toit, et la couleur RED du triangle (le toit).

DrawRectangle() permet de dessiner la porte de la maison. Dans ce cas exitX est la position X de la maison. En ajoutant CELL_SIZE/3 va déplacer la porte vers la droite d'un tiers de la largeur d'une cellule, cela permet à la porte d'être centrée horizontalement par rapport à la maison, laissant un espace égal de chaque côté. Pour exitY est la position Y de la maison en ajoutant CELL_SIZE/2 on place le bas de la porte au milieu de la hauteur de la maison, cela signifie que le haut de la porte sera au-dessus du bas de la maison. La largeur de la porte est définie comme un tiers de la taille d'une cellule, elle permet à la porte d'être proportionnelle à la maison tout en étant suffisamment large pour être visible. La hauteur de la porte est définie comme la moitié de la

taille d'une cellule, ceci donne à la porte une hauteur raisonnable qui correspond à l'échelle générale du dessin et dernièrement la couleur de la porte DARKBROWN.

DrawCircle() est une fonction de **Raylib** du module **rshapes** : ***void DrawCircle(int centerX, int centerY, float radius, Color color);***

- CenterX est exitX + 2*CELL_SIZE/3 – 5 on essaye de positionner horizontalement la poignée à deux tiers de largeur à partir du côté gauche de la maison, puis la déplaceé légèrement vers la gauche (de 5 pixels).
- CenterY est exitY + 3*CELL_SIZE/4 on place verticalement la poignée à trois quarts de hauteur depuis le bas de la maison.
- Radius on le donne une valeur 3, un petit rayon pour représenter une poignée standard, cela donne un aspect proportionnel et réaliste sans être trop grand.
- Color qui est dorée elle est souvent utilisée pour les poignées de porte, ce qui ajoute une touche esthétique et attire l'attention sur cet élément.

```
player->Draw();

EndMode2D();
```

La première ligne appelle la méthode Draw() sur l'objet player, ce qui dessine le joueur à sa position actuelle dans le labyrinthe. Le joueur est dessiné après le labyrinthe et les éléments statiques (comme la maison), ce qui garantit qu'il apparaît au-dessus des autres éléments graphiques.

La deuxième ligne termine le mode 2D dans lequel on avait commencé à dessiner. Cela signifie que tous les dessins effectués
entre BeginMode2D et EndMode2D sont maintenant rendus sur l'écran.

```
float currentTime = GetTime() - gameStartTime;
DrawText(TextFormat("Time: %.1f", currentTime), 10,
10, 50, WHITE);
```

Le currentTime est pour mettre en vue de l'utilisateur combien de temps il a passé et aussi utilisé dans le score ça montre combien de temps s'est écoulé depuis que le jeu a commencé en soustrayant le temps de début (gameStartTime) du temps actuel obtenu par GetTime().

DrawText() est une fonction de **Raylib** du module **rtext** : ***void DrawText(const char *text, int posX, int posY, int fontSize, Color color);*** , elle affiche le temps écoulé sur l'écran dans un format lisible.Le texte formaté avec un nombre flottant affichant une décimale (%.1f). La position (10, 10) où le texte sera

affiché sur l'écran (10 pixels du bord gauche et supérieur). Une taille de police de 50 et la couleur du texte est blanche (WHITE).

```
Vector2 pos = player->GetPosition();

if (pos.x > (mazeSize-2) * CELL_SIZE && pos.y > 
(mazeSize-2) * CELL_SIZE) {
```

La première ligne récupère les coordonnées actuelles du joueur en appelant GetPosition() sur l'objet player.

Puis on vérifie si les coordonnées X et Y du joueur dépassent celles correspondant à un point juste avant les bords inférieurs droit du labyrinthe, cela signifie que si le joueur dépasse ces coordonnées, il a atteint ou dépassé l'emplacement désigné comme sortie.

```
gameState = GameState::FINISHED;
gameEndTime = currentTime;
PlaySound(victorySound);
victoryTimer = 0;
characterHomePos = pos;
 }
            break; // Fin du cas PLAYING
        }
```

Si la condition est vérifié on change d’état vers le dernier qui est FINISHED.

On enregistre currentTime dans la variable gameEndTime pour l’afficher après.

Le son utilisé cette fois est victorySound qui est un paramètre de la fonction PlaySound, ce son déjà existe dans le fichier resource .

characterHomePos est une variable qui représente la position initiale du personnage dans le jeu.
pos est la position actuelle du personnage.

Le cas de l’état Playing est terminé.

```
case GameState::FINISHED: {
            victoryTimer += GetFrameTime();
```

Ce bloc de code s'exécute lorsque l'état du jeu est défini sur FINISHED, cela signifie que le joueur a atteint la sortie du labyrinthe.

La ligne dedans le dernier cas met à jour un chronomètre qui mesure combien de temps s'est écoulé depuis que le joueur a gagné.

GetFrameTime() retourne le temps écoulé depuis la dernière image (en s), ce qui permet d'accumuler le temps total dans victoryTimer.

```
const char* congrats = "Home Sweet Home!";
float scale = 1.0f + 0.1f * sinf(victoryTimer *
5.0f);
float rotation = sinf(victoryTimer * 3.0f) * 5.0f;
```

Ici on déclare une chaîne de caractères contenant le message à afficher lorsque le joueur gagne. Le message "Home Sweet Home!" célèbre la victoire du joueur.

sinf(victoryTimer * 5.0f) utilise la fonction sinus pour créer un effet de pulsation, le facteur 5.0f détermine la vitesse de l'effet. Pour le 1.0f + 0.1f * l'échelle varie entre 0.9 et 1.1, ce qui fait que le texte va légèrement grandir et rétrécir. La variable scale va enregistrer une échelle pour appliquer un effet de pulsation au texte.

sinf(victoryTimer * 3.0f) utilise également la fonction sinus, mais avec un facteur 3.0f, ce qui donne une rotation plus lente, * 5.0f : La valeur résultante est multipliée par 5, ce qui détermine l'ampleur de la rotation. La variable rotation enregistre une rotation pour appliquer un effet de rotation au texte.

```
Vector2 textPos = {
                SCREEN_WIDTH/2.0f,
                SCREEN_HEIGHT/2.0f - 60
            };
```

Le Vector2 textPos crée un vecteur pour définir la position du texte à afficher

CREEN_WIDTH/2.0f va centrer horizontalement le texte sur l'écran et SCREEN_HEIGHT/2.0f - 60 positionne verticalement le texte légèrement au-dessus du centre vertical (60 pixels plus haut), ce qui permet d'éviter qu'il ne soit trop centré sur l'écran.

```
DrawTextPro(GetFontDefault(), congrats, textPos,
{MeasureText(congrats, 60) * scale/2.0f, 40},
rotation, 100, 6, VIOLET);
```

```
DrawText(TextFormat("Your Score: %.1f seconds",
gameEndTime),SCREEN_WIDTH/2 - 100,SCREEN_HEIGHT/2 +
50,50,YELLOW);
```

DrawTextPro() appartient à la bibliothèque **Raylib** dans le module **rtext** : ***void DrawTextPro(Font font, const char *text, Vector2 position, Vector2 origin, float rotation, float fontSize, float spacing, Color tint);*** , elle sert à dessiner le texte avec des effets supplémentaires comme la taille, la rotation et l'échelle. GetFontDefault récupère la police par défaut pour dessiner le texte, congrats est le message à afficher. Le textPos c'est la position calculée pour centrer le texte.

MeasureText(congrats, 60) mesure la largeur du texte pour une taille de police 60, et pour * scale/2.0f c'est pour appliquer l'échelle calculée pour donner un effet visuel dynamique, 40 est la hauteur elle est fixée, rotation applique une rotation au texte, 100 définit la taille du texte en pixels, 6 définit l'espacement entre les lignes et VIOLET est la couleur du texte.

Pour le score on l'affiche par DrawText() le temps écoulé depuis que le joueur a gagné (stocké dans gameEndTime). En utilisant Utilise TextFormat() pour formater le score avec une décimale, et SCREEN_WIDTH/2 centré horizontalement moins 100, donc légèrement décalé vers la gauche, aussi positionné juste en dessous du message de félicitations (SCREEN_HEIGHT/2 + 50) pour une bonne visibilité.

```
 if (victoryTimer > 1.0f) {
float alpha = (sinf(victoryTimer * 2.0f) + 1.0f) *
0.5f;
Color promptColor = WHITE;
promptColor.a = (unsigned char)(255 * alpha);
DrawText("Press SPACE to play again",SCREEN_WIDTH/2
- 120, SCREEN_HEIGHT/2 + 200,50,promptColor);
            }
```

Le if statement vérifie si plus d'une seconde s'est écoulée depuis que le joueur a gagné avant d'afficher le message pour recommencer. Puis on tilise une fonction sinus pour créer un effet de clignotement sur le message, la valeur varie entre 0 et 1, ce qui permet d'ajuster la transparence (alpha) entre complètement transparent et complètement opaque.

Ensuite on crée une couleur blanche pour le message et ajuste son alpha en fonction du calcul précédent.

On affiche un message invitant le joueur à appuyer sur ESPACE pour recommencer.

La position est centrée horizontalement mais décalée vers la gauche (-120) et placée plus bas (+200) sur l'écran.

```
if (IsKeyPressed(KEY_SPACE)) {
gameState = GameState::MENU;
menuTimer = 0;
```

On vérifie si le touches appuyé est espace pour retourné au MENU et joué encore une fois, c'est pour ça on met gameState égale au GameStat MENU, le menuTimer est réinitialiser à zéro pour le réutilise.

```
for (auto& text : storyTexts) {
text.alpha = 0.0f;
text.isVisible = false;
                }
break;
        }
    }
```

Cette for loop en utilisant iterator rend tous les textes invisibles et remet seulement visible celui correspondant au choix des level.

C'était le dernier cas de switch statement, on à traité les trois états possibles.

Pour Désactiver le son on ajoute un bouton qui va permettre d'activer ou désactivé le son.

```
Rectangle soundButton = { SCREEN_WIDTH - 70, 20, 50,
50 };
DrawRectangleRec(soundButton, BLANK);
```

Rectangle est une structure défini dans **Raylib** dans le module **rshapes** : ***Rectangle GetShapesTextureRectangle(void);*** , la position horizontale du coin supérieur gauche du rectangle. Ici, SCREEN_WIDTH - 70 place le rectangle à 70 pixels du bord droit de l'écran, pour la position verticale du coin supérieur gauche. Ici, 20 pixels à partir du haut de l'écran.

Pour DrawRectangleRec on a déjà utilisé cette fonction soundButton à été défini dans la ligne qui précède et blank une couleur prédéfinie dans **Raylib** qui

représente une couleur transparente. Cela permet de créer une zone cliquable sans afficher de rectangle visible.

```
if (soundOn) {
DrawCircle(soundButton.x + soundButton.width / 2,
soundButton.y + soundButton.height / 2, 20, GREEN);
            }
else {
DrawCircle(soundButton.x + soundButton.width / 2,
soundButton.y + soundButton.height / 2, 20, RED);
            }
```

Condition if statement vérifie si le son est activé (soundOn est un booléen). Si elle est vérifié on dessine un cercle de rayon 20 et position on les détermine relativement avec soundButton, pour les couleurs si soundOn est True le cercle est dessiné en vert pour indiquer que le son est activé, si soundOn est false le cercle est dessiné en rouge pour indiquer que le son est désactivé.

```
if (CheckCollisionPointRec(GetMousePosition(),
soundButton) &&
IsMouseButtonPressed(MOUSE_LEFT_BUTTON)) {
                soundOn = !soundOn;
                if (soundOn) {
                    PlaySound(menuMusic);
                } else {
                    StopSound(menuMusic);
                }
            }
```

Dans cette if statement on vérifie les clicks , si la position a actuelle de la souris obtenue par GetMousePosition se trouve à l'intérieur des limites de soundButton et si le bouton gauche de la souris a été pressé au moment de l'appel. Si c'est deux cas sont vérifié c'est-à-dire que l'utilisateur veut activé l'état inverse de ceci donc on met soundOn égale à l'inverse soundOn. Si soundOn est True alors la fonction PlaySound() joue la musique, si soundOn est false alors StopSound() arrête la musique.

```
if (soundOn && !IsSoundPlaying(menuMusic)) {
                PlaySound(menuMusic);
            } else if (!soundOn &&
IsSoundPlaying(menuMusic)) {
                StopSound(menuMusic);
            }
```

Cette section assure que la musique est jouée ou arrêtée en fonction de l'état actuel de soundOn. Si le son est activé (soundOn) et que la musique n'est pas déjà en cours (!IsSoundPlaying(menuMusic)), alors on joue la musique. Inversement, si le son est désactivé (!soundOn) et que la musique est actuellement en cours (IsSoundPlaying(menuMusic)), alors on arrête la musique.

```
    UnloadSound(menuMusic);
    UnloadSound(stepSound);
    UnloadSound(victorySound);
    CloseAudioDevice();
    UnloadTexture(backgroundTex);

    CloseWindow();
    return 0;
}
```

Tout c’est fonction son intègrer dans la bibliothèque **Raylib**, UnloadSound() libèrent les ressources audio associées aux sons spécifiés (menuMusic, stepSound, victorySound). Cela libère de la mémoire et évite les fuites de mémoire.

CloseAudioDevice() ferme le dispositif audio utilisé par Raylib. Cela garantit que toutes les ressources audios sont correctement libérées avant que le programme ne se termine.

UnloadTexture(backgroundTex) : Libère les ressources associées à une texture de fond utilisée dans l'application.

CloseWindow() : Ferme la fenêtre graphique créée par Raylib.

return 0; : Indique que le programme s'est terminé avec succès.

Player.h

Rôle principal de player.h

Ce fichier d'en-tête (header file) déclare une classe appelée Player. Cette classe représente un joueur dans un jeu de labyrinthe. Elle définit les propriétés (attributs) et les actions (méthodes) nécessaires comme son mouvement et son dessin à l'écran.

La classe Player est conçue pour gérer un joueur dans un jeu, avec des fonctionnalités pour gérer la position, la vitesse, les collisions, et le dessin du joueur à l'écran.

Explication détaillée des différentes parties de cette classe :

1. **Directive de préprocesseur qui empêche les inclusions multiples du fichier d'en-tête.**

```
1  #pragma once // Empêche les inclusions multiples de ce fichier d'en-tête
```

- si ce fichier est inclus plusieurs fois dans d'autres fichiers, son contenu ne sera compilé qu'une seule fois, évitant les erreurs de redéfinition.

2. **fichier d'en-tête inclut la définition de la classe Maze, qui est utilisée pour gérer les murs ou la structure d'un labyrinthe dans ce contexte.**

```
2  #include "maze.h" // Inclut la définition de la classe Maze
```

3. **Inclusion de bibliothèque graphique Raylib légère pour la création d'applications graphiques**

```
3  #include "raylib.h" // Inclut la bibliothèque raylib pour la gestion des graphiques
```

4. **Classe player :**

```
5    class Player { // Déclaration de la classe Player
```

5. Section privée

```
6    private: // Début de la section privée, accessible uniquement à la classe elle-même
7    Vector2 position; // Position actuelle du joueur
```

- Représente la position actuelle du joueur dans le jeu. Vector2 est une structure de la bibliothèque raylib qui contient deux valeurs flottantes (x et y) pour représenter une position en 2D.

```
8    float radius; // Rayon du joueur (utilisé pour les collisions)
```

- Représente le rayon du joueur. Cela peut être utilisé pour gérer les collisions, en supposant que le joueur est représenté par un cercle.

```
9    float speed; // Vitesse de déplacement du joueur
```

- Représente la vitesse de déplacement du joueur.

```
10   bool isMoving; // Indique si le joueur est en mouvement
```

- Indique si le joueur est actuellement en mouvement. C'est un booléen qui est true si le joueur se déplace, et false sinon.

Les méthodes privées suivantes sont également définies :

```
12   public: // Début de la section publique, accessible depuis l'extérieur de la classe
13   Player(float x, float y); // Constructeur de la classe Player initialisant la position du joueur
```

- Constructeur de la classe Player qui initialise la position du joueur. Il prend deux flottants représentant les coordonnées x et y de la position initiale.

```
14   bool Update(const Maze& maze); // Met à jour l'état du joueur en fonction du labyrinthe
```

- Met à jour l'état du joueur en fonction du labyrinthe. Cette méthode prend une référence constante à un objet Maze et retourne un booléen.

```
15      void Draw() const; // Dessine le joueur à l'écran
```

- Dessine le joueur à l'écran. Cette méthode est constante, ce qui signifie qu'elle ne modifie pas l'état de l'objet Player.

```
16      // Retourne la position actuelle du joueur
17      Vector2 GetPosition() const;
18      };
```

- Retourne la position actuelle du joueur. Cette méthode est également constante.

6. Attributs principaux :

- Position (position) : Où se trouve le joueur dans le jeu.
- Rayon (radius) : Taille du joueur (utile pour les collisions et le dessin).
- Vitesse (speed) : À quelle vitesse le joueur peut bouger.
- État (isMoving) : Indique si le joueur est en mouvement.

7. Actions principales :

- Update() : Gère les mouvements et interactions du joueur avec l'environnement.
- Draw() : Affiche le joueur sur l'écran.
- GetPosition() : Permet à d'autres objets de savoir où est le joueur.

Player.Cpp

Le fichier player.cpp implémente un joueur capable de :

- Se déplacer dans un labyrinthe tout en respectant les limites (pas de traversée des murs).
- S'afficher visuellement comme un cercle avec des détails minimalistes (yeux et sourire).
- Retourner sa position actuelle pour une utilisation par d'autres composants.

1. *Inclusion des bibliothèques*

```
1  #include "player.h" // Inclut le fichier d'en-tête pour la classe Player
2  #include <cmath> // Inclut la bibliothèque cmath pour les fonctions mathématiques
3  #include <raymath.h> // Inclut la bibliothèque raymath pour les opérations vectorielles
```

- player.h : Contient les déclarations de la classe Player.
- cmath : Fournit des fonctions mathématiques standard.
- raymath : Fournit des fonctions pour les opérations vectorielles.

2. Constructeur de la classe Player

```
5  // Constructeur de la classe Player, initialise la position , le rayon, la vitesse, et l'état de mouvement du joueur
6  Player::Player(float x, float y)
7  : position{x, y}, radius{CELL_SIZE * 0.4f},
8  speed{CELL_SIZE}, isMoving{false} {}
```

Initialise un objet Player avec une position (x, y).

- radius est défini comme 40% de la taille d'une cellule (CELL_SIZE).
- speed est défini comme la taille d'une cellule.
- isMoving est initialisé à false, indiquant que le joueur ne bouge pas au départ.

3. La méthode bool Player::Update(const Maze& maze)

Met à jour la position du joueur en fonction des touches appuyées (haut, bas, gauche, droite) et vérifie si le déplacement est valide (pas de collision avec un mur dans le labyrinthe).

Src\player.cpp

```
10  // Méthode pour mettre à jour la position du joueur en fonction des entrées et du labyrinthe
11  bool Player::Update(const Maze& maze) {
12  bool moved = false; // Variable pour suivre si le joueur a bougé
```

4. Détection des touches si en etat appuyée :

```
14  // Accepter une nouvelle entrée de mouvement seulement si le joueur ne bouge pas actuellement
15  if (!isMoving) {
16  Vector2 newPosition = position; // Nouvelle position potentielle du joueur
17  bool shouldMove = false; // Indicateur pour savoir si le joueur doit bouger
```

- Si la touche droite, gauche, haut ou bas est appuyée, la nouvelle position du joueur est calculée :

```
19  // Vérifie si la touche droite est pressée
20  if (IsKeyPressed(KEY_RIGHT)) {
21  newPosition.x += CELL_SIZE; // Déplace la position vers la droite
22  shouldMove = true; // Indique que le joueur doit bouger
23  }
24  // Vérifie si la touche gauche est pressée
25  else if (IsKeyPressed(KEY_LEFT)) {
26  newPosition.x -= CELL_SIZE; // Déplace la position vers la gauche
27  shouldMove = true; // Indique que le joueur doit bouger
28  }
29  // Vérifie si la touche bas est pressée
30  else if (IsKeyPressed(KEY_DOWN)) {
31  newPosition.y += CELL_SIZE; // Déplace la position vers le bas
32  shouldMove = true; // Indique que le joueur doit bouger
33  }
34  // Vérifie si la touche haut est pressée
35  else if (IsKeyPressed(KEY_UP)) {
36  newPosition.y -= CELL_SIZE; // Déplace la position vers le haut
```

```
37      shouldMove = true; // Indique que le joueur doit bouger
38      }
```

Note : Une seule direction peut être prise en compte à chaque frame.

5. **Vérification de la validité de la nouvelle position :**

```
40      // Vérifie si la nouvelle position est valide (pas dans un mur)
41      if (shouldMove) {
42      int newCellX = static_cast<int>(newPosition.x / CELL_SIZE); // Calcule
        la nouvelle cellule en X
43      int newCellY = static_cast<int>(newPosition.y / CELL_SIZE); // Calcule
        la nouvelle cellule en Y
```

- La méthode maze.isWall(newCellX, newCellY) est utilisée pour vérifier si la nouvelle position est dans une cellule contenant un mur :

```
45      // Vérifie si la nouvelle cellule n'est pas un mur
46      if (!maze.isWall(newCellX, newCellY)) {
47      position = newPosition; // Déplace instantanément à la nouvelle position
48      moved = true; // Indique que le joueur a bougé
49      }
50      }
```

6. **Retourner si le joueur a bougé ou non :**

```
53      return moved; // Retourne si le joueur a bougé ou non
54      }
```

- Cela permet d'informer d'autres parties du code si un déplacement a eu lieu.

7. **La méthode void Player::Draw() const**

- Dessine le joueur à l'écran, avec un corps circulaire, deux yeux, et un sourire.

```
56      // Méthode pour dessiner le joueur à l'écran
57      void Player::Draw() const {
```

8. **Dessiner le corps du joueur**

Utilise la fonction DrawCircleV pour dessiner un cercle représentant le joueur :

```
58    DrawCircleV(position, radius, PLAYER_COLOR); // Dessine le corps du joueur
```

9. Dessiner les yeux :

```
60    // Dessine les yeux
61    float eyeSize = radius / 4; // Taille des yeux
```

Les yeux sont deux petits cercles positionnés à l'intérieur du corps du joueur, légèrement au-dessus du centre :

```
62    Vector2 leftEye = {position.x - radius / 3, position.y - radius / 3}; // Position de l'œil gauche
63    Vector2 rightEye = {position.x + radius / 3, position.y - radius / 3}; // Position de l'œil droit
64    DrawCircleV(leftEye, eyeSize, WHITE); // Dessine l'œil gauche
65    DrawCircleV(rightEye, eyeSize, WHITE); // Dessine l'œil droit
```

10. Dessiner le sourire :

Une ligne horizontale en bas du cercle pour représenter un sourire :

```
67    // Dessine le sourire
68    Vector2 smileStart = {position.x - radius / 2, position.y + radius / 3}; // Début du sourire
69    Vector2 smileEnd = {position.x + radius / 2, position.y + radius / 3}; // Fin du sourire
70    DrawLineEx(smileStart, smileEnd, 2.0f, WHITE); // Dessine le sourire
```

11. La méthode Vector2 Player::GetPosition() const

Retourne la position actuelle du joueur sous forme de vecteur Vector2. Cela permet à d'autres objets ou systèmes (comme une caméra ou un ennemi) de connaître où se trouve le joueur.

```
73    // Méthode pour obtenir la position actuelle du joueur
74    Vector2 Player::GetPosition() const {
```

```
75      return position; // Retourne la position du joueur
76      }
```

Points importants à noter

- ***Gestion du mouvement :***

  Le joueur ne bouge que si une touche est pressée ET que la nouvelle position est valide.

  La taille des déplacements est basée sur CELL_SIZE, ce qui signifie que le joueur se déplace d'une cellule à la fois dans le labyrinthe.

- ***Validation des collisions :***

  maze.isWall(newCellX, newCellY) est essentiel pour empêcher le joueur de traverser les murs du labyrinthe.

- ***Apparence du joueur :***

  Le design simple (cercle, yeux, sourire) est efficace pour un prototype ou un jeu minimaliste.

- ***Utilisation des bibliothèques :***

  raylib et raymath facilitent la gestion graphique et les calculs vectoriels, simplifiant le code.

Maze.h

Dans cette partie du projet, nous avons mis en place une fonctionnalité de génération

automatique de labyrinthes dans cette jeu. L'objectif était de créer un algorithme qui génère un labyrinthe de manière procédurale, garantissant une solution valide et un chemin entre le point de départ et le point d'arrivée. Pour ce faire, nous avons utilisé une approche basée sur
l'algorithme de parcours en profondeur (Depth-First Search - DFS), qui explore le

labyrinthe en créant des passages tout en respectant les contraintes
de structure d'un labyrinthe.

Génération du Labyrinthe

La classe Maze gère la génération du labyrinthe. Elle utilise un tableau 2D pour stocker les murs et l'état des cellules, et implémente un algorithme de génération basé sur le parcours en
profondeur (DFS).

Constructeur de la classe Maze:

Le constructeur de la classe Maze initialise les murs du labyrinthe avec des valeurs true (ce qui signifie qu'il y a un mur), puis appelle la fonction generateMaze pour générer le labyrinthe à
partir de la position (1, 1). Il s'assure également que le point de départ et le point d'arrivée ne sont pas bloqués par des murs.

```
5   // Constructeur de la classe Maze qui initialise la taille du labyrinthe et génère le labyrinthe
6   Maze::Maze(int mazeSize) : size(mazeSize) {
7       // Initialisation des murs du labyrinthe avec des valeurs true (mur présent)
8       walls = std::vector<std::vector<bool>>(size, std::vector<bool>(size, true));
9       generateMaze(1, 1); // Appel de la fonction pour générer le labyrinthe à partir de la position (1, 1)
10      walls[1][1] = false; // Suppression du mur à la position de départ
11      walls[size-2][size-2] = false; // Suppression du mur à la position d'arrivée
12      ensurePath(); // Appel de la fonction pour s'assurer qu'il y a un chemin entre le début et la fin
13  }
14
```

Initialisation des murs du labyrinthe :

- On utilise un vecteur 2D (tableau de tableaux) de type bool pour représenter les murs du labyrinthe. La valeur true représente un mur et false représente un espace vide.
- La taille de ce vecteur est déterminée par la variable size, passée en argument au constructeur.
- Exemple : pour un labyrinthe de taille 5, walls sera une matrice de 5 x 5 initialisée à true (tous les murs sont présents).

1. ***Appel à generateMaze(1, 1) :***

- Cette ligne commence la génération du labyrinthe à partir de la position (1, 1) (les bords sont généralement laissés pour des murs). Cette fonction utilise un algorithme de parcours en profondeur pour créer des chemins dans le labyrinthe.

2. ***Suppression du mur au départ et à l'arrivée :***

- Le mur à la position de départ (1, 1) est supprimé (false).
- Le mur à la position de l'arrivée (size-2, size-2) est également supprimé pour créer un passage vers la sortie.

Appel de ensurePath() :

- Cette fonction est appelée pour s'assurer qu'il existe un chemin valide entre le départ et l'arrivée. Elle intervient après la génération du labyrinthe pour rendre le parcours entre ces deux points possible.

Fonction generateMaze:

La fonction generateMaze est la clé de la génération du labyrinthe. Elle utilise une approche récursive et explore chaque cellule en marquant les murs à supprimer, créant ainsi un chemin dans le labyrinthe. Elle s'appuie sur un algorithme DFS et un ensemble de directions possibles (haut, bas, gauche, droite) pour explorer le labyrinthe.

```
// Fonction récursive pour générer le labyrinthe en utilisant l'algorithme de parcours en profondeur
void Maze::generateMaze(int x, int y) {
    walls[x][y] = false; // Suppression du mur à la position actuelle
    // Liste des directions possibles pour le déplacement (haut, bas, gauche, droite)
    std::vector<std::pair<int, int>> directions = {{0,2}, {2,0}, {0,-2}, {-2,0}};
    static std::random_device rd; // Dispositif de génération de nombres aléatoires
    static std::mt19937 gen(rd()); // Générateur de nombres aléatoires basé sur Mersenne Twister
    std::shuffle(directions.begin(),directions.end(),gen);// Mélange des directions pour un parcours aléatoire
    // Boucle sur chaque direction possible
    for (const auto& dir : directions) {
        int nextX = x + dir.first; // Calcul de la prochaine position en x
        int nextY = y + dir.second; // Calcul de la prochaine position en y
        // Vérification que la prochaine position est valide et que le mur est présent
        if (nextX>0 && nextX<size-1 && nextY> 0 && nextY<size-1 && walls[nextX][nextY]) {
            walls[x + dir.first/2][y + dir.second/2] = false; // Suppression du mur intermédiaire
            generateMaze(nextX, nextY); // Appel récursif pour continuer la génération du labyrinthe
        }}}
```

3. ***Suppression du mur à la position actuelle :***

- La position (x, y) actuelle est marquée comme un chemin (false) dans le vecteur walls. Cette opération est répétée à chaque étape du parcours récursif, construisant progressivement le labyrinthe.

4. ***Liste des directions possibles :***

- directions contient les déplacements possibles pour la génération : vers le haut, vers le bas, vers la gauche et vers la droite, chacun ayant une distance de deux cellules (pour s'assurer que nous sautons un mur pour créer un passage).

- Exemple : {0,2} signifie un déplacement de 2 cellules vers le bas, {2,0} signifie un déplacement de 2 cellules vers la droite, et ainsi de suite.

5. ***Mélange des directions :***

- std::shuffle(directions.begin(), directions.end(), gen) mélange les directions de manière aléatoire. Ce mélange est essentiel pour éviter la création de labyrinthes prévisibles et garantir un caractère aléatoire dans la structure du labyrinthe.

6. ***Exploration des directions :***

- Pour chaque direction dans directions, le programme calcule la position (nextX, nextY) et vérifie si cette position est valide (dans les limites du labyrinthe et si la cellule est encore un mur).

- Si la cellule est valide, le programme supprime le mur intermédiaire (entre la cellule actuelle et la cellule suivante) et continue l'exploration en appelant récursivement generateMaze(nextX, nextY).

Explication de la génération :

- La fonction commence à la position (x, y) et marque la cellule comme un passage (false).
- Une liste de directions possibles est définie, et ces directions sont mélangées pour introduire de l'aléatoire dans le processus de génération.
- Pour chaque direction (haut, bas, gauche, droite), la fonction vérifie si le déplacement est valide (c'est-à-dire s'il ne sort pas des limites du labyrinthe et s'il n'y a pas encore de passage à cet endroit).
- Si un déplacement est valide, un mur intermédiaire est supprimé, et la fonction s'appelle récursivement pour explorer ce nouvel endroit.

7. *Fonction ensurePath:*

Une fois que la génération du labyrinthe est terminée, la fonction ensurePath() garantit qu'il y a bien un chemin entre le point de départ (en haut à gauche) et le point d'arrivée (en bas à droite). Cette fonction supprime les murs autour du point de départ et du point d'arrivée pour faciliter la traversée du labyrinthe.

```
//Fonction pour s'assurer qu'il y a un chemin entre le début et la fin du labyrinthe
void Maze::ensurePath() {
    for (int i=0;i<3;i++) { //Boucle pour créer un chemin de 3 cellules
        walls[1][i+1]=false; //Suppression des murs pour le chemin de départ
        walls[size-2][size-2-i]=false; //Suppression des murs pour le chemin d'arrivée
    }
}
```

- Cette fonction crée manuellement un chemin entre le départ (1, 1) et l'arrivée (size-2, size-2) en supprimant les murs qui se trouvent directement sur le chemin entre ces deux points. Cela permet de s'assurer que, quelle que soit la structure générée, il existe un moyen d'aller du début à la fin.

Cette méthode modifie les cellules près des bords du labyrinthe pour garantir qu'un chemin de départ et d'arrivée existe. Elle supprime une série de murs à partir du coin supérieur gauche et du coin inférieur droit.

8. *Fonction isWall:*

La fonction isWall() permet de vérifier si une position donnée dans le labyrinthe

correspond à un mur. Elle est utilisée pour la détection des collisions et pour déterminer si une position est traversable.

```
// Fonction pour vérifier si une position donnée est un mur
bool Maze::isWall(int x,int y)const{
    if (x<0||x>=size||y<0||y>=size) return true; // Retourne true si la position est en dehors des limites
    return walls[x][y]; // Retourne la valeur du mur à la position donnée
}
```