



JEU DE CASSE-TÊTE LABYRINTHE



Réalisé par :

BOUKAB Fatima Zahra
BOURHDIFA Aya
EN -NAJIB Hajar

Encadré par :

Pr.Ikram Ben Abdel Ouahab

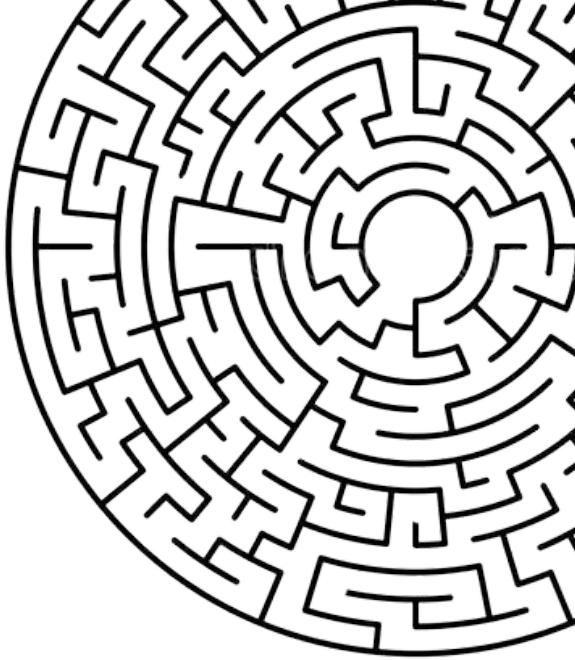


OBJECTIF DU PROJET :

L'objectif de ce projet est de développer un jeu de casse-tête de type labyrinthe en utilisant la programmation orientée objet (POO) en C++ et la bibliothèque graphique Raylib. Le jeu proposera une expérience immersive où le joueur devra naviguer dans un labyrinthe généré aléatoirement à chaque nouvelle partie. Il comprendra trois niveaux de difficulté (facile, moyen, difficile), qui affecteront la taille du labyrinthe et la complexité de la navigation.



PLAN :



1- Introduction

2-Étapes de création du jeu

3-Techniques utilisées

4-Conclusion



Introduction

Programmation Orientée Objet:

La programmation orientée objet gagne en popularité dans le monde de la programmation et de la science des données. Cette approche, centrée sur les objets ou les entités de données, présente de multiples avantages, ce qui explique son adoption croissante dans le développement logiciel et la data science. Les concepts clés de la POO sont :

- **La classe** : une classe est un ensemble de code contenant des variables et des fonctions.
- **Les objets** : un objet est un bloc de code mêlant des variables et des fonctions, appelées respectivement attributs et méthodes.
- **L'encapsulation** : l'encapsulation permet d'enfermer dans une capsule les données.
- **L'héritage** : le concept d'héritage signifie qu'une classe B va hériter des mêmes attributs et méthodes qu'une classe A.
- **Le polymorphisme** : lorsqu'une classe hérite des méthodes d'une classe parent, il est possible de surcharger une méthode, qui consiste à redéfinir la méthode de la classe parent.

Langage de programmation C++ :

C++ est un langage de programmation polyvalent basé sur C, intégrant la programmation orientée objet. Il est réputé pour son efficacité et sa performance, mais peut être complexe en raison de ses fonctionnalités avancées



La bibliothèque Raylib :

Raylib est une bibliothèque open source en C conçue pour le développement de jeux vidéo et d'applications graphiques. Elle est reconnue pour sa simplicité d'utilisation, permettant aux développeurs, y compris les débutants, de créer rapidement des jeux en 2D et 3D. Raylib offre un large éventail de fonctionnalités, comme la gestion des graphismes, des sons, des entrées utilisateur et de la physique, tout en restant légère et performante. Sa documentation claire et ses nombreux exemples facilitent l'apprentissage, ce qui en fait un outil idéal pour les projets éducatifs ou prototypes rapides.

Étapes de création du jeu

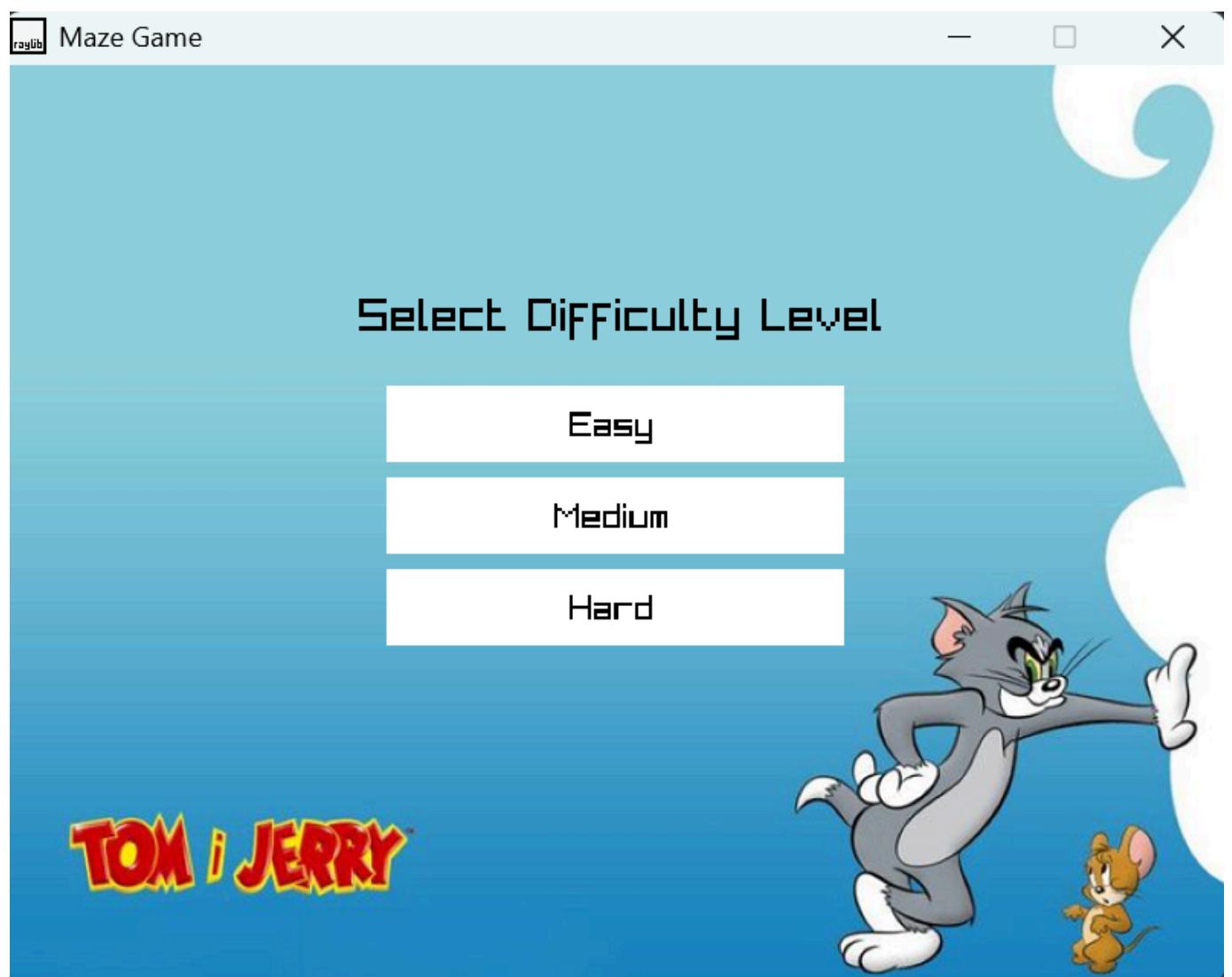
La création du jeu consiste en plusieurs étapes :

- Création de la page d'accueil du jeu , cette interface présente une image dynamique avec deux personnages emblématiques Tom&Jerry , et une musique pour plonger le joueur dans l'ambiance , et un bouton "PLAY NOW" bien visible qui lance le jeu.

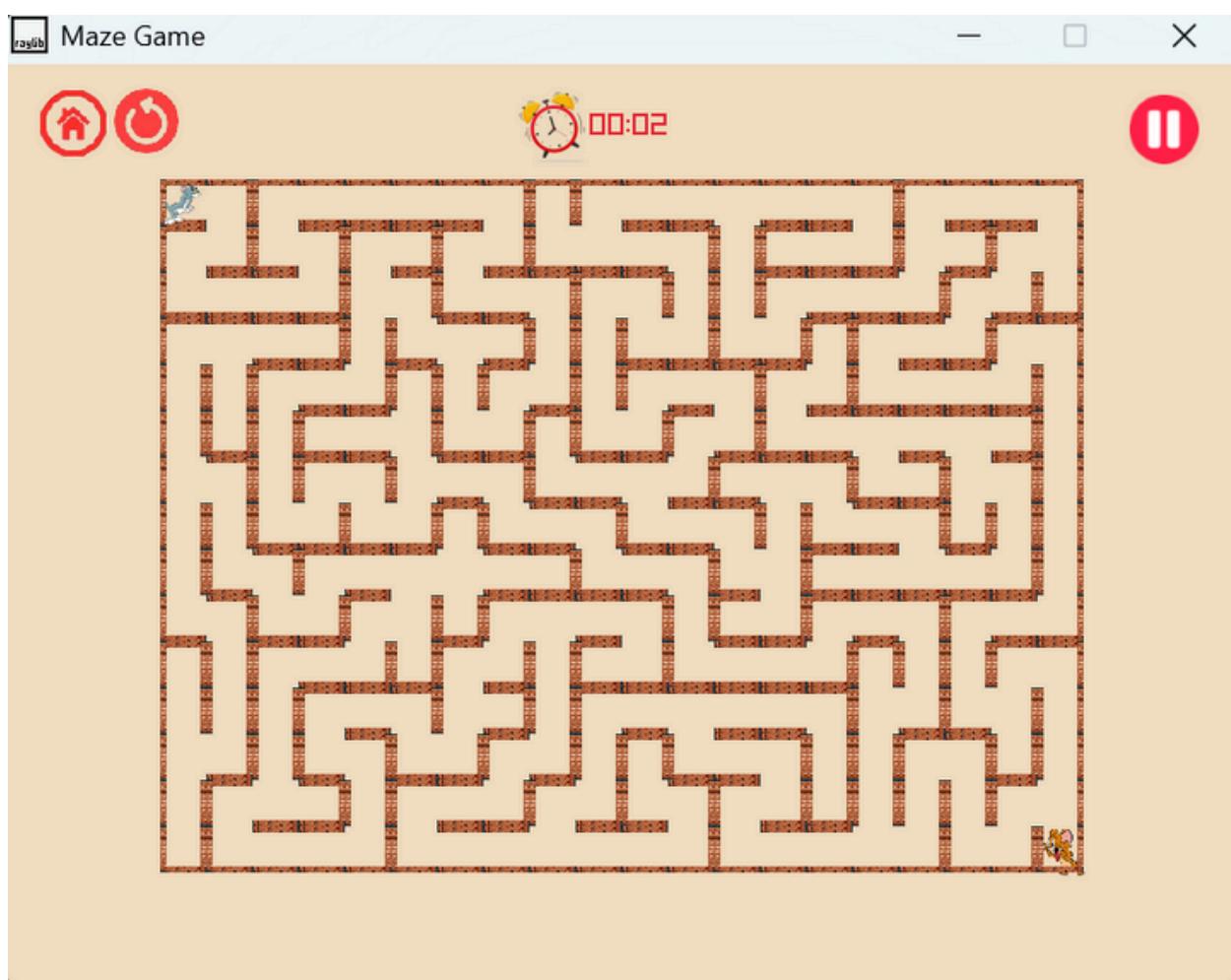


- Après que l'utilisateur a cliqué sur le bouton "PLAY NOW", la deuxième interface s'affiche avec le texte "Select Difficulty Level" en haut de l'écran. Elle comprend trois boutons :
 - Le premier bouton, "EASY"
 - Le deuxième bouton, "MEDIUM"
 - Le troisième bouton, "HARD"

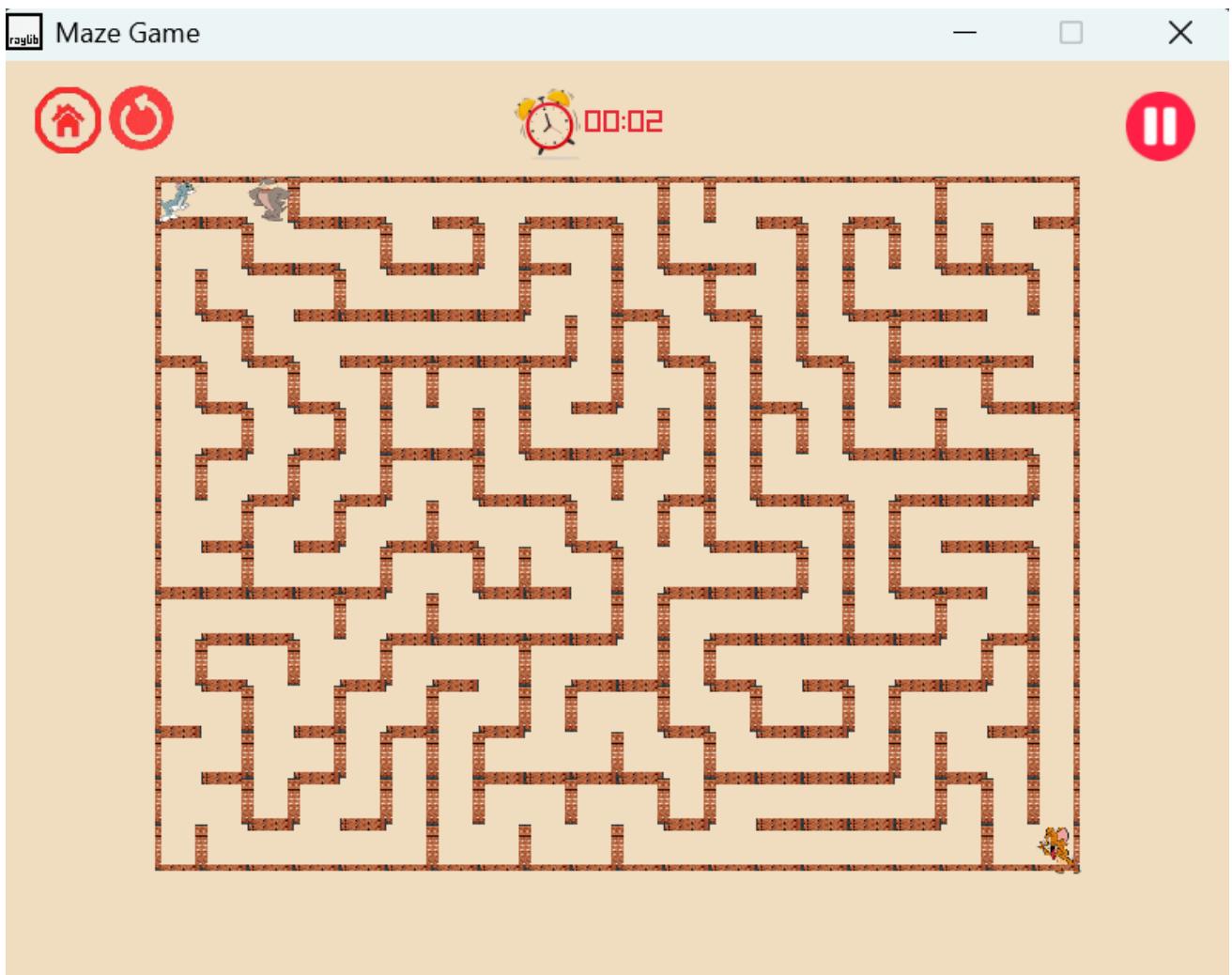
Ces boutons permettent à l'utilisateur de choisir le niveau de difficulté du jeu



- Le jeu représente un labyrinthe avec un personnage joueur (Tom) situé à l'entrée et une icône (Jerry) marquant la sortie. L'interface inclut un chronomètre qui indique le temps écoulé. Elle comporte également trois boutons :
 - Un bouton "RESET", représenté par une flèche en boucle, permettant de réinitialiser le jeu.
 - Un bouton "PAUSE", marqué par une icône de pause, pour interrompre temporairement la partie.
 - Un bouton "HOME", permet de revenir à la page d'accueil.
- L'objectif du jeu est de guider le personnage à travers le labyrinthe pour atteindre la sortie.



- Dans le deuxième niveau, un obstacle mobile, représenté par une icône (Spike), si le joueur entre en contact avec cet obstacle, il est automatiquement renvoyé à sa position initiale (l'entrée du labyrinthe), est ajouté pour augmenter la difficulté du jeu.



- Dans le troisième niveau, le labyrinthe change de configuration toutes les 3 secondes, rendant le jeu plus dynamique et augmentant considérablement la difficulté.

- L'interface de fin de partie s'affiche lorsque le joueur remporte la victoire. Un grand texte annonce "YOU WIN!" pour célébrer le succès. Juste en dessous, deux informations sur les performances du joueur sont présentées :

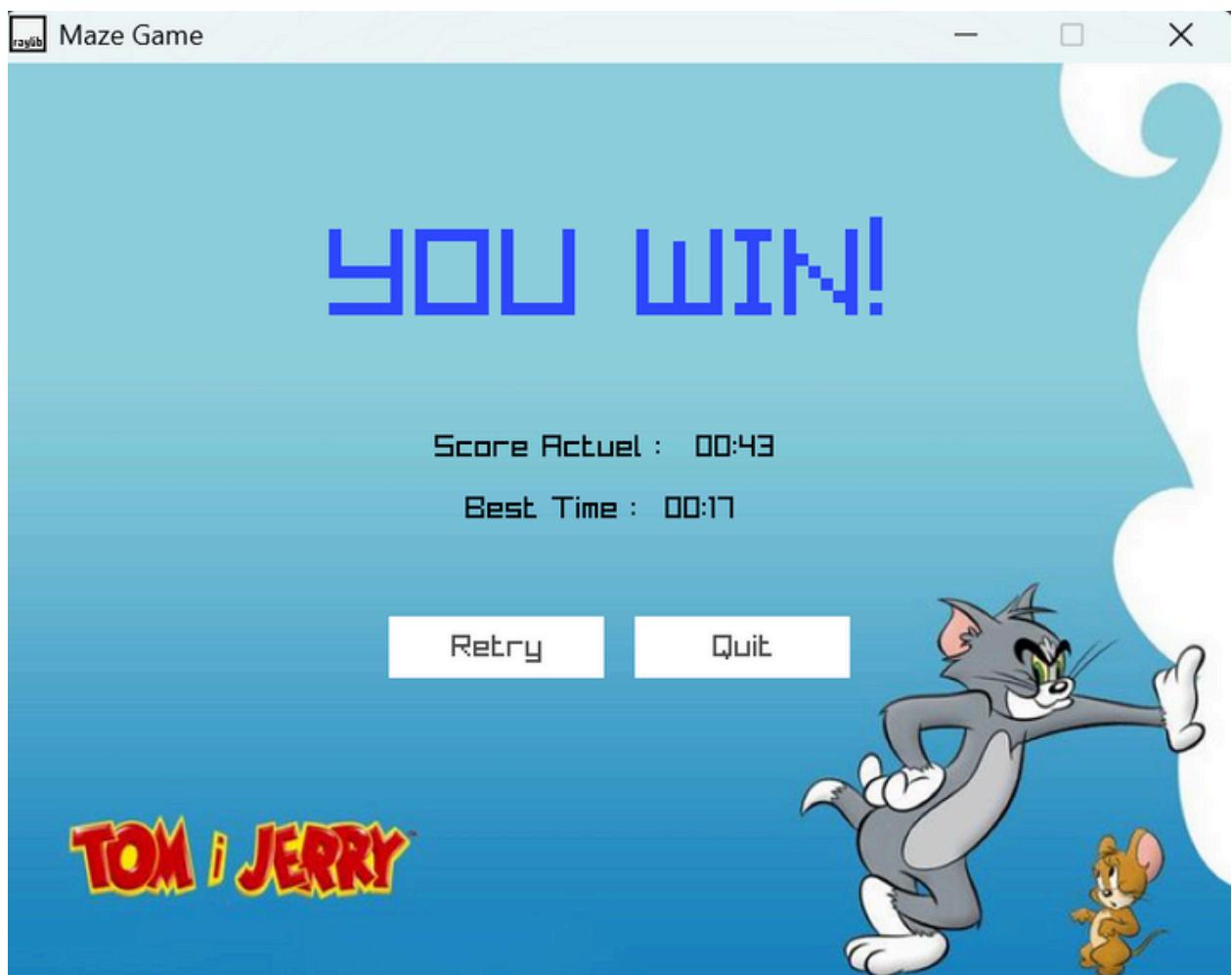
-le Score Actuel: indiquant le temps pris pour terminer le labyrinthe .

- le Best Time: qui affiche le meilleur temps réalisé jusqu'à présent .

Enfin, deux boutons sont disponibles :

-le bouton Retry: pour relancer une nouvelle partie.

-le bouton Quit : pour quitter le jeu.



Techniques utilisées

- Classe Position :

```
class Position { // Définition d'une classe représentant une position avec des coordonnées x et y
public:
    int x, y; // Déclaration des variables membres x et y représentant la position

    // Constructeur qui initialise les valeurs x et y à 0 par défaut, mais peut être modifié avec des valeurs spécifiques
    Position(int x = 0, int y = 0) : x(x), y(y) {}

};
```

La classe **Position** représente les coordonnées d'un point dans l'espace du labyrinthe. Elle est utilisée pour localiser des éléments clés comme l'entrée, la sortie ou le joueur. Elle permet également de suivre les déplacements du joueur en mettant à jour ses coordonnées. De plus, elle facilite les calculs lors des algorithmes, comme ceux de génération ou de résolution du labyrinthe. Grâce à sa simplicité, cette classe offre une structure claire pour manipuler les positions et gérer efficacement les éléments du jeu.

- Classe Cell :

```
class Cell {
public:
    bool visited; // Variable indiquant si la cellule a été visitée ou non (utile pour l'algorithme de génération de labyrinthes)
    bool topWall, bottomWall, leftWall, rightWall; // Variables pour les murs de la cellule (chaque mur est représenté par un booléen)

    // Constructeur par défaut
    Cell() : visited(false), topWall(true), bottomWall(true), leftWall(true), rightWall(true) {}

    // Le constructeur initialise les valeurs des membres de la classe :
    // - visited est initialisé à false (la cellule n'a pas été visitée)
    // - tous les murs (haut, bas, gauche, droite) sont initialisés à true (les murs sont présents)
};

};
```

La classe **Cell** joue un rôle central dans la représentation de chaque unité du labyrinthe. Elle est utilisée pour construire la structure globale du labyrinthe en gérant les murs et l'état de visite des cellules lors de la génération et de l'affichage du labyrinthe.

- Classe Niveau:

```
class Niveau {
public:
    enum Level { FACILE, MOYEN, DIFFICILE }; // Définition d'un énuméré pour les trois niveaux de difficulté
    Level niveau; // Variable membre qui représente le niveau actuel

    Niveau(Level level) : niveau(level) {}

    // Fonction qui retourne la densité des obstacles en fonction du niveau choisi
    float getObstacleDensity() {
        switch (niveau) {
            case FACILE: return 0.3f; // Pour le niveau facile, la densité des obstacles est de 30%
            case MOYEN: return 0.5f; // Pour le niveau moyen, la densité des obstacles est de 50%
            case DIFFICILE: return 0.7f; // Pour le niveau difficile, la densité des obstacles est de 70%
        }
    }

    // Fonction qui indique si le niveau est dynamique (seulement le niveau difficile est dynamique)
    bool isDynamic() {
        return niveau == DIFFICILE; // Si le niveau est difficile, la fonction retourne true, sinon false
    }
};
```

La classe **Niveau** est utilisée pour gérer les niveaux de difficulté du labyrinthe. Elle définit trois niveaux (FACILE, MOYEN, DIFFICILE) grâce à un énuméré et ajuste le comportement du jeu en conséquence. Son rôle est de fournir des paramètres adaptés à chaque niveau, comme la densité des obstacles via la fonction `getObstacleDensity`, qui influence la complexité du labyrinthe. De plus, elle indique si le niveau est dynamique avec la fonction `isDynamic`, permettant de modifier les règles du jeu pour le rendre plus difficile. Ainsi, cette classe centralise la gestion des niveaux et leur impact sur le labyrinthe.

• Classe Obstacle:

```
class Obstacle {
public:
    Position position;           // La position de l'obstacle dans le labyrinthe (utilise la classe Position pour gérer les coordonnées x et y)
    float moveTimer;             // Un compteur de temps qui permet de contrôler le déplacement de l'obstacle
    float moveInterval;          // L'intervalle de temps entre chaque déplacement de l'obstacle (en secondes)
    Texture2D texture;          // Texture de l'obstacle (ex. image qui représente l'obstacle)

    // Constructeur qui initialise la position, le timer de mouvement, l'intervalle et la texture de l'obstacle
    Obstacle(int x = 0, int y = 0, float interval = 100.0f, const char* texturePath = "Spike.png") |
        : position(x, y), moveTimer(0), moveInterval(interval) {
        texture = LoadTexture(texturePath); // Charge la texture spécifiée pour l'obstacle
    }

    // Destructeur qui décharge la texture quand l'obstacle n'est plus nécessaire
    ~Obstacle() {
        UnloadTexture(texture); // Décharge la texture de la mémoire
    }

    // Fonction qui permet de changer la texture de l'obstacle
    void SetTexture(const char* texturePath) {
        UnloadTexture(texture); // Décharge la texture actuelle
        texture = LoadTexture(texturePath); // Charge la nouvelle texture
    }

    // Fonction qui fait déplacer l'obstacle dans le labyrinthe
    void Move() {
        moveTimer += GetFrameTime(); // Incrémente le timer de mouvement en fonction du temps écoulé entre deux images

        if (moveTimer >= moveInterval) { // Si l'intervalle de déplacement est atteint
            // Déplace l'obstacle dans une direction aléatoire
            position.x += GetRandomValue(-2, 2);
            position.y += GetRandomValue(-2, 2);

            // Limite les déplacements de l'obstacle pour qu'il reste dans les limites du labyrinthe
            if (position.x < 0) position.x = 0;
            if (position.x >= GRID_WIDTH) position.x = GRID_WIDTH - 1;
            if (position.y < 0) position.y = 0;
            if (position.y >= GRID_HEIGHT) position.y = GRID_HEIGHT - 1;

            moveTimer = 0; // Réinitialise le timer pour le prochain déplacement
        }
    }
}
```

La classe **Obstacle** gère l'obstacle dynamique dans le labyrinthe. Elle définit sa position avec la classe Position, contrôle leurs déplacements avec un timer moveTimer et un intervalle moveInterval, et leur apparence grâce à une texture texture. Elle inclut une fonction Move pour déplacer l'obstacle à intervalles réguliers dans des directions aléatoires, tout en restant dans les limites du labyrinthe. Cela ajoute une dynamique supplémentaire et augmente la difficulté du jeu.

```

// Fonction qui dessine l'obstacle à l'écran avec mise à l'échelle et décalage
void Draw(float scaleFactor, int offsetX, int offsetY) {
    int scaledCellSize = scaleFactor * SCREEN_WIDTH / GRID_WIDTH; // Calcul de la taille de la cellule mise à l'échelle
    int posX = offsetX + position.x * scaledCellSize; // Calcul de la position horizontale de l'obstacle
    int posY = offsetY + position.y * scaledCellSize; // Calcul de la position verticale de l'obstacle

    // Dessine la texture de l'obstacle à la position spécifiée
    DrawTexturePro(texture, {0, 0, (float)texture.width, (float)texture.height},
    {(float)posX, (float)posY, (float)scaledCellSize, (float)scaledCellSize}, {0, 0}, 0, WHITE);
}

// Fonction qui vérifie si l'obstacle est en collision avec le joueur
bool CheckCollision(Position player) {
    // Retourne true si l'obstacle se trouve à la même position que le joueur
    return (position.x == player.x && position.y == player.y);
}
];

```

La classe permet également de gérer l'affichage des obstacles avec la fonction `Draw`, qui dessine chaque obstacle à une position calculée à l'écran en fonction de l'échelle et des décalages. De plus, la fonction `CheckCollision` vérifie si l'obstacle entre en collision avec le joueur en comparant leurs positions respectives, permettant de déclencher des effets spécifiques en cas de collision.

• Classe Maze :

```
✓ class Maze {
    private:
        Cell maze[GRID_WIDTH][GRID_HEIGHT];      // Tableau de cellules représentant le labyrinthe
        Niveau niveau;                          // Niveau du jeu, définissant la difficulté
        Obstacle movingObstacle;                // Obstacle qui se déplace dans le labyrinthe
        int gridWidth, gridHeight;               // Dimensions du labyrinthe
        Texture2D wallTexture;                  // Texture des murs du labyrinthe

        // Génère un chemin dans le labyrinthe en utilisant un algorithme de backtracking
        void GeneratePath(int x, int y) {
            maze[x][y].visited = true; // Marque la cellule actuelle comme visitée
            while (true) {
                // Tableau de directions possibles : 0 = haut, 1 = droite, 2 = bas, 3 = gauche
                int directions[] = {0, 1, 2, 3};

                // Mélange aléatoirement les directions pour diversifier le parcours
                for (int i = 0; i < 4; i++) {
                    int j = GetRandomValue(i, 3);
                    int temp = directions[i];
                    directions[i] = directions[j];
                    directions[j] = temp;
                }

                bool moved = false; // Indicateur si un mouvement a été effectué
                for (int i = 0; i < 4; i++) {
                    int nx = x, ny = y;
                    // Calcul de la nouvelle position en fonction de la direction
                    if (directions[i] == 0) ny -= 1; // Haut
                    else if (directions[i] == 1) nx += 1; // Droite
                    else if (directions[i] == 2) ny += 1; // Bas
                    else if (directions[i] == 3) nx -= 1; // Gauche

                    if (nx >= 0 && nx < gridWidth && ny >= 0 && ny < gridHeight && !maze[nx][ny].isWall) {
                        if (maze[nx][ny].isWall) {
                            maze[nx][ny].isWall = false;
                            moved = true;
                        }
                    }
                }
            }
        }
}
```

La classe **Maze** est essentielle pour gérer la structure et les fonctionnalités du labyrinthe dans le jeu. Elle utilise un tableau bidimensionnel de cellules pour représenter la grille, où chaque cellule peut contenir des murs et des états comme "visité" ou "non visité". Le labyrinthe est généré dynamiquement à l'aide d'un algorithme de backtracking, qui crée un chemin en supprimant les murs entre les cellules tout en assurant la diversité des parcours grâce à un mélange aléatoire des directions. La classe intègre également un obstacle mobile, géré en fonction de la difficulté définie par l'objet **Niveau**.

En termes visuels, chaque mur du labyrinthe est dessiné avec une texture spécifique, et la méthode **DrawMaze** ajuste les dimensions et l'échelle pour s'adapter à l'écran.

La classe permet aussi de régénérer le labyrinthe à partir de la position actuelle du joueur, par exemple après un événement spécifique, en réinitialisant toutes les cellules et en générant un nouveau chemin .

Enfin, la méthode `HasWall` permet de vérifier la présence de murs dans une direction donnée, ce qui est crucial pour les déplacements du joueur. Cette conception modulaire rend `Maze` non seulement responsable de la logique structurelle mais aussi de l'interaction visuelle et dynamique du jeu.

- Classe Player:

```
class Player {  
public:  
    Position position;           // Position actuelle du joueur dans le labyrinthe  
    int gridWidth, gridHeight;   // Dimensions dynamiques du labyrinthe  
    Texture2D texture;          // Texture pour représenter le joueur  
  
    // Constructeur initialisant la position et la texture du joueur  
    Player(int x = 0, int y = 0, const char* texturePath = "Tom.png")  
        : position(x, y), gridWidth(GRID_WIDTH), gridHeight(GRID_HEIGHT) {  
            // Charger la texture depuis un fichier  
            texture = LoadTexture(texturePath);  
    }  
  
    // Destructeur pour décharger la texture lorsque l'objet est détruit  
    ~Player() {  
        UnloadTexture(texture);  
    }  
}
```

La classe **Player** représente le joueur dans le labyrinthe, gérant sa position, ses déplacements et son affichage. Elle stocke la position actuelle du joueur sous forme de coordonnées dans la grille et utilise une texture pour le représenter visuellement. Cette classe est initialisée avec une position par défaut et une texture chargée depuis un fichier.

Elle inclut une méthode **Move** pour permettre au joueur de se déplacer dans le labyrinthe, tout en vérifiant que le déplacement est valide : il doit rester à l'intérieur des limites de la grille et ne pas être bloqué par un mur. De plus, cette méthode détecte si le joueur a atteint la sortie , signalant ainsi la victoire.

Pour l'aspect visuel, la méthode **Draw** affiche le joueur à sa position actuelle en adaptant l'échelle et le positionnement à l'écran. La classe supporte également des labyrinthes de tailles différentes grâce à la méthode **SetGridSize**, qui ajuste dynamiquement les dimensions du labyrinthe. Enfin, le destructeur veille à libérer les ressources associées à la texture, garantissant une gestion efficace de la mémoire.

- Classe Game :

```
class Game {  
private:  
    Maze maze; // Le labyrinthe du jeu  
    Player player; // Le joueur, représentant Tom  
    Position goal; // La position de l'objectif ( Jerry)  
    bool gameWon; // Indicateur si le jeu est gagné  
    bool isPaused; // Indicateur si le jeu est en pause  
    float timer; // Chronomètre du jeu  
    float changeTimer; // Timer pour régénérer le labyrinthe  
    float bestTime; // Meilleur temps du joueur  
    Niveau niveau; // Niveau de difficulté  
    Rectangle resetButton; // Bouton pour réinitialiser le jeu  
    Texture2D resetButtonTexture; // Texture du bouton Reset  
    Rectangle homeButton; // Bouton pour revenir à l'écran d'accueil  
    Texture2D homeButtonTexture; // Texture du bouton d'accueil  
    Rectangle retryButton; // Bouton pour recommencer  
    Rectangle quitButton; // Bouton pour quitter  
    Rectangle pauseButton; // Bouton pour mettre en pause  
    Texture2D goalTexture; // Texture pour le point d'arrivée  
    Texture2D timerIcon; // Texture pour l'icône du timer  
    Texture2D backgroundTexture; // Texture pour l'arrière-plan du jeu  
    Texture2D pauseTexture; // Texture pour le bouton de pause  
    Texture2D resumeTexture; // Texture pour le bouton de reprise
```

La classe **Maze** représente le labyrinthe central du jeu et joue un rôle clé dans son fonctionnement. Elle est utilisée pour générer et gérer la structure du labyrinthe en fonction du niveau de difficulté choisi. Lors de l'initialisation de la classe Game, une instance de Maze est créée en passant le niveau sélectionné, ce qui détermine la taille de la grille et le placement des obstacles. La classe fournit des fonctionnalités pour régénérer dynamiquement le labyrinthe, ce qui est utile dans des niveaux où le labyrinthe évolue au fil du temps. De plus, elle intègre un obstacle mobile, particulièrement pertinent pour les niveaux intermédiaires, qui interagit directement avec le joueur en détectant les collisions. Le labyrinthe sert aussi de base pour les mouvements du joueur en validant si une position donnée est accessible. Enfin, la classe Maze gère le dessin graphique du labyrinthe et de ses éléments, comme les obstacles fixes et mobiles, permettant une représentation visuelle cohérente à l'écran, tout en s'adaptant dynamiquement à l'état du jeu et à l'échelle d'affichage.

- Fonction ShowIntroScreen :

```
void ShowIntroScreen() {
    // Charger l'image de fond
    Texture2D background = LoadTexture("img2.png"); // Remplacez par le chemin de votre image

    // Initialiser le système audio
    InitAudioDevice();

    // Charger et jouer la musique en boucle
    Music introMusic = LoadMusicStream("tom-and-jerry-ringtone (online-audio-converter.com).wav");
    PlayMusicStream(introMusic); // Jouer la musique
    SetMusicVolume(introMusic, 0.5f); // Optionnel : ajuster le volume de la musique

    // Calculer l'échelle de l'image pour s'adapter à l'écran
    float scaleX = (float)SCREEN_WIDTH / (float)background.width;
    float scaleY = (float)SCREEN_HEIGHT / (float)background.height;
    float scale = (scaleX > scaleY) ? scaleX : scaleY; // Choisir l'échelle la plus adaptée pour ne pas déformer l'image

    // Définir le bouton "PLAY NOW"
    Rectangle playButtonBase = {SCREEN_WIDTH - 220, 20, 200, 60}; // Taille de base du bouton
    float buttonScale = 1.0f; // Échelle dynamique du bouton
    float scaleSpeed = 0.5f; // Vitesse d'oscillation
    bool isGrowing = true; // Indique si le bouton est en train de grandir
```

La fonction **ShowIntroScreen** gère l'écran d'introduction du jeu. Elle charge une image de fond et une musique, créant une ambiance engageante grâce à la lecture en boucle de cette dernière avec un volume ajustable. L'image est redimensionnée dynamiquement pour s'adapter parfaitement à l'écran, tandis qu'un bouton "PLAY NOW" est affiché avec une animation d'oscillation pour capter l'attention de l'utilisateur. Ce bouton réagit au survol de la souris en changeant de couleur et permet, via un clic, de quitter l'écran d'introduction pour passer à la suite du jeu. Pendant la boucle principale de cet écran, la musique est mise à jour en continu et les animations du bouton sont actualisées pour garantir une fluidité visuelle.

- Méthode ShowLevelMenu :

```
Niveau::Level ShowLevelMenu() {
    // Charger l'image de fond
    Texture2D background = LoadTexture("img4.png");

    // Calculer l'échelle de l'image pour s'adapter à l'écran
    float scaleX = (float)SCREEN_WIDTH / (float)background.width;
    float scaleY = (float)SCREEN_HEIGHT / (float)background.height;

    // Choisir la plus petite échelle pour conserver les proportions
    float scale = scaleX;

    // Dimensions des boutons
    int buttonWidth = 300;
    int buttonHeight = 50;
    int buttonX = SCREEN_WIDTH / 2 - buttonWidth / 2;

    // Positions des boutons
    Rectangle easyButton = {buttonX, SCREEN_HEIGHT / 2 - 90, buttonWidth, buttonHeight};
    Rectangle mediumButton = {buttonX, SCREEN_HEIGHT / 2 - 30, buttonWidth, buttonHeight};
    Rectangle hardButton = {buttonX, SCREEN_HEIGHT / 2 + 30, buttonWidth, buttonHeight};

    // Couleurs des boutons
    Color easyColor = WHITE;
    Color mediumColor = WHITE;
    Color hardColor = WHITE;
```

La méthode **ShowLevelMenu** de la classe Niveau gère l'affichage du menu de sélection du niveau de difficulté. Elle charge une image de fond et l'adapte à la taille de l'écran. Trois boutons représentant les niveaux Easy, Medium et Hard sont affichés, et leur couleur change en fonction de la position de la souris. Lorsqu'un utilisateur clique sur un bouton, la fonction vérifie quel bouton a été sélectionné et retourne le niveau correspondant (Niveau::FACILE, Niveau::MOYEN, ou Niveau::DIFFICILE). Enfin, elle libère les ressources utilisées, comme l'image de fond, pour gérer efficacement la mémoire.

Conclusion

Les objectifs fixés ont été atteints : une génération procédurale de labyrinthes respectant les règles de connexité, l'intégration de trois niveaux de difficulté adaptés, et une interface graphique intuitive et fluide.

Au-delà de l'aspect technique, ce projet a également permis de relever plusieurs défis, notamment la création d'algorithmes robustes pour la génération de labyrinthes résolvables, la gestion des collisions, et l'amélioration de l'expérience utilisateur avec des mécanismes tels que le chronomètre et la réinitialisation dynamique. De plus, l'ajout d'un obstacle mobile dans les niveaux avancés a enrichi la complexité du gameplay et apporté une dimension stratégique au jeu.

Ce projet nous a permis de travailler en équipe et de partager nos connaissances, nous enseignant également la patience.

Enfin, je tiens à remercier tout particulièrement Madame Ikram Ben Abdel Ouahab pour tous ses efforts durant le cours et ses précieux conseils.

