

RAPPORT : PROJET NE FIN NE MONNIE C++

Un Jeu De Style Labyrinthe (Maze) en C++ avec Raylib

Licence
IDAI

Encadrée Par :

Dr. Ikram Benabdelouahab

Réalisé Par :

Lina El Barrout

Nouhaila Ayad

Khaoula El Bakkali





Introduction :

Ce projet vise à concevoir un jeu de casse-tête interactif de type labyrinthe en utilisant le langage **C++** et la bibliothèque graphique Raylib. Le jeu sera construit selon les principes de la programmation orientée objet (**POO**) et offrira une expérience immersive grâce à la génération aléatoire de labyrinthes et une interface graphique intuitive. Les joueurs devront naviguer dans un labyrinthe et résoudre ses défis en fonction de différents niveaux de difficulté.



Objectif :

- Génération aléatoire de labyrinthes respectant des règles de connexité pour garantir qu'une solution existe.
- Une interface graphique **2D** fluide, gérée par la bibliothèque **Raylib**, permettant au joueur d'explorer le labyrinthe.
- Trois niveaux de difficulté (**facile**, **moyen**, **difficile**), influençant la taille et la complexité des labyrinthes.



Définition de C++ :

C++ est un langage de programmation généraliste orienté objet qui offre des performances élevées et une grande flexibilité. Connu pour sa gestion fine des ressources et sa compatibilité avec les systèmes embarqués, il est couramment utilisé dans le développement de jeux vidéo, d'applications complexes et de logiciels nécessitant un haut degré de performance.

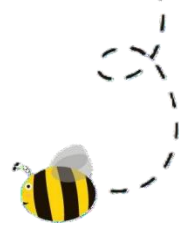


Définition de Raylib :

Raylib est une bibliothèque graphique open-source simple et facile à utiliser, destinée à la création de jeux vidéo en **2D** et **3D**. Conçue pour être intuitive, elle permet de gérer les graphiques, les entrées utilisateur, l'audio, et bien d'autres aspects nécessaires au développement de jeux.



Les classes :



la classe Chronomètre :



La classe Chronomètre est conçue pour gérer et afficher un chronomètre interactif dans une application graphique utilisant la bibliothèque " Raylib" . Elle permet de mesurer le temps écoulé, de contrôler son fonctionnement (démarrage, pause, mettre à jour, reprise, réinitialisation), et d'afficher ce temps de manière visuelle et esthétique avec des éléments graphiques .

✧ La structure de la Classe :

La classe Chronomètre est composée de plusieurs attributs et méthodes qui gèrent l'état et le comportement du chronomètre.

✓ les Attributs :

temps_départ : type(float) : enregistre le temps de départ du chronomètre.

temps_actuel : type(float) : Enregistre le temps écoulé depuis le départ.

actif : type(bool) : Indique si le chronomètre est en cours d'exécution (true) ou non (false).

temps_arrêt : type(float) : Enregistre le temps au moment de l'arrêt

textureheur : type(texture) : Texture utilisée pour afficher l'image de chronometre.

✓ les methodes :

Chronometre() : c'est un Constructeur par défaut qui Initialise le chronomètre avec des valeurs par défaut.

démarrer() : type(void) : Cette méthode démarre le chronomètre en enregistrant l'heure actuelle à l'aide de **GetTime()**. Elle définit également l'attribut actif à true, indiquant que le chronomètre est en cours d'exécution.

arrêter() : type(void) : Cette méthode permet d'arrêter le chronomètre en mettant actif à false. Elle enregistre également le temps actuel dans temps_arret, permettant de reprendre le chronomètre à partir de ce point si nécessaire.





continuer() :type(void) : Cette méthode reprend le chronomètre après l'avoir arrêté. Elle définit `actif` à `true` et ajuste `temps_depart` pour qu'il prenne en compte le temps déjà écoulé avant l'arrêt.

mettre_a_jour() :type(void) : Cette méthode qui met à jour `temps_actuel` si le chronomètre est actif. Elle calcule le temps écoulé en soustrayant `temps_depart` de l'heure actuelle obtenue via `GetTime()`. Cela permet de suivre le temps en temps réel.

reinitialiser() :type(void) : Cette méthode réinitialise le chronomètre en remettant `temps_depart` et `temps_actuel` à 0 et en définissant `actif` à `false`. Cela permet de redémarrer le chronomètre à partir de zéro.

afficher() :type(void) : Cette méthode est responsable de l'affichage du chronomètre à l'écran. Elle :

- ✓ Calcule le nombre de minutes et de secondes à partir de `temps_actuel`.
- ✓ Change la couleur du texte en fonction du temps écoulé (par exemple, si le temps dépasse 10 secondes).
- ✓ Affiche le temps au format `MM:SS`.
- ✓ Affiche une texture (la photo du chrono) à côté du temps.

setTexture(Texture2D texture):type(void) : Cette méthode permet de définir la texture à utiliser pour l'affichage. Elle prend en paramètre une texture de type `Texture2D` et l'assigne à l'attribut `textureheur`.

la classe **TableauScores** :

La classe **TableauScores** permet de présenter visuellement les scores clés du jeu, en affichant le meilleur score et le score final de manière graphique et intuitive utilisant la bibliothèque Raylib.

✧ Structure de la Classe :

La classe **TableauScores** contient une méthode principale s'appelle **afficher**, Cette méthode est responsable de l'affichage des scores à l'écran. Elle prend plusieurs paramètres pour personnaliser l'affichage :

✓ les Attributs :

meilleurScore (float) : Le meilleur score enregistré

scoreFinal (float): Le score final du joueur.

x (int)et y(int) : Les coordonnées de position (en pixels) de scores



✓ la methode :

La classe TableauScores contient une méthode principale s'appelle : **afficher()** : Cette méthode affiche deux scores (le meilleur score et le dernier score) dans une interface graphique. Les scores sont convertis en secondes (en prenant le reste de la division par 60) et affichés sur l'écran avec des images.

La Classe Position :

Cette classe représente une position 2D avec des coordonnées **x** et **y** ; la classe contient un constructeur qui initialise les coordonner en (0,0) et revoie avec les méthodes **getX()** et **getY()** .

✓ les Attributs :

X :type(int)

Y:type(int)

✓ les methodes :

Le constructeur Position : initialiser les coordonnes X et Y

getX():type (int) :**accesseur** pour renvois la valeur de X

getY():type(int) :**accesseur** pour renvois la valeur de X

La Classe Cell :



✓ les Attributs :

Tout les attributs de type booléen :

Visited : si la cellule visiter ou non

Topwall: le mur en haut

Bottomwall:le mur en bas

Leftwall:le mur a gauche

Rightwall:le mur a droite

✓ les methodes :

Tout les methodes de type booléen :

le constructeur cell () : initialise les variable ; **visited** par **false** pour traduire que la cellule n'a pas été visiter ,(topwall,bottomwall,leftwall,rightwall)par **true** pour que la cellule au début soit fermé avec des mure de partout .

lesméthodes(hasTopWall(),hasBottomwall(),hasleftwall(),hasrightwall()) : renvoie **true** si la cellule a un murs du coter définit sinon renvoie **false** (exemple hasTopWall() renvoie **true** si la cellule a un mur supérieur)

La Classe MAZE(Labyrinthe) :

Parlant du fichier labyrinthe.h qui comporte la classe du labyrinthe qui constitue la partie principale du jeu est nommée « Maze » cette classe contient des membres et des méthodes publique (pour pouvoir les utiliser plus tard en dehors de la classe ;

✓ les Attributs :

Cell maze [MazeConfig::grid_width][MazeConfig::grid_height]:Une Matrice 2D pour les cellules du labyrinthe, chaque cellule contient des informations sur les murs et l'état de visite.

WallTexture: type(Texture2D) :Qui représente la texture pour les murs du labyrinthe.

BackgroundTexture:type(Texture2D):Qui représente la texture pour le fond des cellules du labyrinthe (pour le niveau difficile).

✓ les methodes :

le constructeur :Comme toute autre classe, la classe « Maze » contient un constructeur nommé «**maze ()**» . Cette méthode initialise le labyrinthe en appelant la méthode "**initiaizeMaze ()** " et change les textures des murs du fond à partir de fichiers (png) en utilisant la fonction **LoadTexture()**.

✧ Les méthodes principales pour générer un labyrinthe :

Void initiaizeMaze () :Réinitialise chaque cellule avec des valeurs par défauts (contient tous ces murs et gardant les cellules marquer non visiter), en les parcourant une par une avec 2 boucles for limiter par leur taille de grille.

Void generatemaze (int x, int y) :Après avoir eu des cellules avec tous leurs murs il est temps de supprimer certains, tout en implémentant un algorithme de **backtracking** pour générer un labyrinthe parfait (sans cycles) en commençant par les paramètre (x,y) choisie et le marquer comme visiter, cette méthode récursive mélange ensuite les directions possible à l'aide de **GetRandomValue()** de façon aléatoire pour que le labyrinthe soit imprévisible et calcule la position de la cellule voisine en vérifiant bien sûr que la cellule est bien dans la grille et qu'elle n'est pas visiter puis supprimer le mure entre la cellule actuelle et le voisin (exemple : si en va en haut on enlève le mur du haut de la cellule actuelle et le mur du bas de la cellule voisine) puis appelle mais cette fois ci pour la cellule voisine. Une fois que tous les voisins ont été visités la méthode revient on arrière et termine son appel récursif pour explorer tout le labyrinthe sans oublier aucune cellule. cette méthode est faites à l'aide de 2 boucles for et indépendant avec des structure if else.

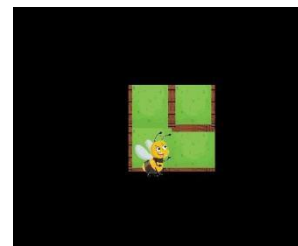
Void addCycles (int count) :Étant donné que notre labyrinthe utilise l'Algorithme de **backtracking (Recursive Backtracker)** qui génère une seul solution unique, cette méthode est là pour rendre le labyrinthe plus complexe on ajoutant le nombre de cycles choisie en paramètre ceux qui implique des chemins supplémentaires. Pour ajouter un cycle, **addCycles ()** choisie aléatoirement sa position avec la fonction **GetRandomValue ()** puis vérifie si cette cellule a été visitée ; si une cellule aléatoire sélectionnée n'a pas été visitée, la boucle passe simplement à une autre tentative sans casser de mur. Puis choisie un mur à casser aléatoirement (droite ou bas) :

- ✓ Si le mur de droite est choisi, supprimer le mur de droite de la cellule actuelle et le mur de gauche de la cellule voisine.
- ✓ Si le mur du bas est choisi, supprimer le mur du bas de la cellule actuelle et le mur du haut de la cellule voisine

Ce qui crée un nouveau chemin on les rendant plus difficile à résoudre.

drawCompleteMaze () : Cette méthode est utilisée pour dessiner tout le labyrinthe en parcourant toutes les cellules et affiche ces murs en texture (**walltexture**) avec la fonction **DrawTexturePro ()** en vérifiant quels murs doivent être dessinés pour chaque cellule.

void drawNearMaze (const Position& position) :Cette méthode a pour but d'afficher une petite zone du labyrinthe autour de la position du joueur. On commence d'abord par couvrir l'entièreté de la page en noire puis, parcourir toute les cellules visibles en regardant autour de la position du joueur ensuite dessiner l'arrière-plan des cellules et enfin dessiner les murs si il existe dans la cellule.



void addDeceptivePaths (int count, int maxLength) : Cette méthode permet de générer des chemins trompeurs (des chemins dans le labyrinthe qui ne mènent à rien) ; on commence par choisir un point et la longueur du chemin aléatoirement avec **GetRandomValue ()** puis il crée un chemin en prenant une direction aléatoire et brise les murs entre les cases ; si la nouvelle case est valide ce si en le répétons (**int count**) fois donner en paramètre.

Le destructeur : Et enfin la classe dispose d'un destructeur "**~Maze ()**" qui est appelée automatiquement lorsque l'objet du labyrinthe est détruit, on déchargeant les textures "**wallTexture**" et "**BackgroundTexture**" par la fonction **UnloadTexture ()** pour empêcher les fuites de mémoire quand l'objet n'est plus utilisé.

La Classe Obstacle :

La classe Obstacle permet de gérer les obstacles mobiles dans un labyrinthe. Elle permet de définir leur apparence, leur position, leur déplacement, ainsi que leur gestion en cas de collision ou de sortie des limites. Cette classe enrichit le jeu en ajoutant des éléments dynamiques et interactifs.

✓ les Attributs :

currentPosition :type(Position) : pour la position actuelle

Texture :type(Texture2D) : la texture de l'obstacle (image)

Direction :type(Vector2) : la direction de déplacement

Speed : type(float) : la vitesse

Scale: type(float) : la taille de l'obstacle

✓ les methodes :



Le Constructeur Obstacle (): Initialise l'obstacle avec une texture, une position initiale, une vitesse et un facteur d'échelle.

resetPosition(): permet de réinitialiser la position actuelle de l'obstacle à sa position initiale.

move(Maze &maze):

- ✓ Change la direction de l'obstacle.
- ✓ Calcule la prochaine position en fonction de la direction.



- ✓ Vérifie les murs du labyrinthe pour déterminer si l'obstacle peut se déplacer.
- ✓ Réinitialise la position si l'obstacle dépasse les limites du labyrinthe.

La méthode `draw()`: Affiche l'obstacle à sa position actuelle sur l'écran en tenant compte de l'échelle.

La méthode `getCollisionRectangle()`: Retourne un rectangle de collision basé sur la position actuelle de l'obstacle, utilisé pour détecter les collisions avec d'autres objets.

Destructeur `~Obstacle` : Libère la mémoire allouée pour la texture de l'obstacle.

La Classe Niveau :

Cette classe encapsule et simplifie la gestion du labyrinthe à différents niveaux de difficulté. La classe regroupe les fonctionnalités nécessaires pour configurer le labyrinthe en fonction de différents niveaux (facile, moyen, difficile).

✓ les Attributs :

`maze : type(Maze)` : représente le labyrinthe et fournit toutes les fonctionnalités liées à sa création

`std::vector<Obstacle> obstacles` : Liste des obstacles présents dans le niveau.

`int currentDifficultyLevel` : Indicateur du niveau de difficulté actuel (1 = facile, 2 = moyen, 3 = difficile).

✓ les methodes :

Constructeur `Niveau` : Le constructeur initialise un nouvel objet `Maze` en générant un labyrinthe à partir de la cellule (0, 0).

`void genererObstacles(int nombreObstacles)` (privée)

Son rôle : Génère un nombre donné d'obstacles dans des positions aléatoires du labyrinthe.

`void easyNiveau()` Rôle : Configure le niveau "facile" avec 2 obstacles et affiche le labyrinthe complet.

void mediumNiveau() Rôle : Configure le niveau "moyen" avec 2 obstacles et affiche le labyrinthe complet.

void difficultNiveau(const Position& position) Rôle : Configure le niveau "difficile" avec 4 obstacles , Réinitialise le labyrinthe , l'ajoute des chemins trompeurs,et affiche une portion du labyrinthe autour du joueur.

void miseAJourNiveau() Rôle : Réinitialise le labyrinthe et rétablit le niveau de difficulté à 0.

std::vector<Obstacle>& getObstacles() Rôle : Retourne la liste des obstacles présents dunniveau actuel.

Le son :

✧ Gestion des Effets Sonores

Les effets sonores jouent un rôle crucial dans la création d'une expérience immersive pour les joueurs. Ils permettent de renforcer l'engagement et d'ajouter une dimension auditive au jeu. Chaque son est soigneusement associé à un événement spécifique, on utilisons **InitAudioDevice()** pour initialiser le système audio et **CloseAudioDevice()** pour fermer le système ce qui aide à contextualiser les actions du joueur. Voici un aperçu des différents effets sonores gérés dans le code :

✓ Son d'introduction :

En a utiliser la fonction (**introSound**) lors du lancement du jeu, établissant l'ambiance et préparant le joueur à l'expérience à venir.

✓ Son de jeu :

Concernant Le son de jeu (**jeuSound**) :Se déclenche pendant le déroulement du jeu, créant une atmosphère dynamique et immersive qui accompagne les actions du joueur.

✓ Son collision :

Et aussi le son de collision (**collisionSound**) :Émis lorsqu'une collision avec un obstacle se produit, signalant au joueur qu'une interaction a eu lieu et ajoutant une réactivité au jeu.

✓ Son victoire :

Finalement le son de victoire (**victoireSound**) qui Joue lorsque le joueur atteint la fleur ou remporte le jeu, renforçant le sentiment d'accomplissement et de satisfaction.

La Classe Bouton:

la classe bouton represente un bouton dans une interface utilisateur graphique

✓ les Attributs :

Texture : type(Texture2D) :utilisé pour l'image de bouton

Position :type(Vector2) : pour positionner le bouton

Rect :type(Rectangle) :utilisé pour définir la position et les dimensions du bouton dans l'espace 2D,

isClick: type(bool) : utilisé pour le logique d'utilisation de bouton

✓ les methodes :

Constructeur Button() : C'est le constructeur de la classe. Il initialise les variables **texture** , **position** ,**rect** et **isClick** .

Constructeur 2 Button():C'est un constructeur Surchargé.il prend en paramètres un chemin d'image ,une position et un facteur d'échelle .il charge l'image et définit les variables **texture**,**position** et **rect**.

isClicked (): cette méthode vérifie si le bouton est cliqué. Elle retourne **true** si la souris est cliqué sur le bouton , et **false** si non

Draw():cette méthode dessine le bouton a l'écran . elle utilise les variables **texture** et **position** pour déterminer la position et la taille du bouton.

La Classe Player(Abeille):



La classe Player représente un joueur dans un labyrinthe. Elle gère la position, le mouvement, et l'affichage du joueur à l'écran, ainsi que l'objectif à atteindre pour gagner.

✓ les Attributs :

➤ **Attributs privés :**

Texture :`type(texture2D)` : Stocke la texture (image) utilisée pour représenter visuellement le joueur.

Position :`type(position)` : Stocke les coordonnées actuelles du joueur dans le labyrinthe.

Goal:`type(position)` : Stocke les coordonnées de l'objectif final que le joueur doit atteindre pour gagner.

maze:`type(&Maze)` : Référence vers le labyrinthe dans lequel évolue le joueur. Il est utilisé pour vérifier les murs et les limites du labyrinthe.

Scale :`type(float)` : Définit l'échelle de la texture pour ajuster sa taille lorsqu'elle est affichée à l'écran.

➤ **Attribut public :**

hasWon:`type(Bool)` : Indique si le joueur a atteint l'objectif (**true** si le joueur gagne).


✓ **les methodes :**

Constructeur Player() : Initialise un joueur avec une position de départ, un labyrinthe de référence, une texture, et initialise **hasWon** à **false**.


Constructeur 2 Player(): Constructeur par défaut .

movePlayer(): `type(void)` : La méthode **moveplayer** permet de déplacer le joueur dans le labyrinthe en fonction des paramètres **dx** et **dy**, qui représentent les variations des coordonnées **x** et **y** . Elle vérifie d'abord si le déplacement respecte les limites du labyrinthe pour éviter toute sortie de la grille. Ensuite, elle s'assure qu'il n'y a pas de murs bloquant le passage en vérifiant les murs adjacents dans la structure du labyrinthe. Si les conditions sont remplies, la position du joueur est mise à jour. Enfin, elle détermine si le joueur a atteint l'objectif (goal), en mettant la variable **hasWon** à **true** lorsque la position actuelle correspond à celle de l'objectif.

Draw(): `type(void)` : Affiche le joueur sur l'écran à sa position actuelle. Elle ne prend aucun paramètre. La position du joueur dans la grille est convertie en coordonnées en pixels à l'aide des configurations du labyrinthe (comme *MazeConfig::maze_offset_x* et *MazeConfig::cell_size*). Ensuite, la texture associée au joueur est dessinée à l'écran avec l'échelle définie (par défaut **0.10f**). La fonction **DrawTextureEx()** est utilisée pour gérer le rendu visuel du joueur avec la texture correspondante.



drawGoal():type(void) :La méthode **drawGoal** affiche l'objectif (position finale) sur l'écran. Elle ne prend aucun paramètre. elle calcule la position en pixels en fonction des coordonnées de l'objectif goal dans la grille et des configurations du labyrinthe. La texture est ensuite dessinée à cette position avec une échelle différente (*0.24f*), afin de mettre en valeur visuellement la zone d'arrivée. Cette méthode permet au joueur de visualiser clairement l'objectif à atteindre dans le labyrinthe.



Possitionabb():type(void) :Réinitialise la position du joueur au point de départ défini par (*0, MazeConfig::grid_height - 1*). Elle remet également la variable **hasWon** à **false** pour indiquer que le joueur n'a pas encore atteint l'objectif. Cette méthode ne prend aucun paramètre et est utile pour redémarrer une partie ou réinitialiser l'état du joueur après un échec.

getPosition():type(position) :La méthode **getPosition** est un accesseur qui retourne la position actuelle du joueur dans le labyrinthe. Elle ne prend aucun paramètre et renvoie un objet **Position** contenant les coordonnées **x** et **y** du joueur. Cette méthode est utile pour obtenir la position du joueur à des fins de logique de jeu ou d'affichage.

getCollisionRectangle():type(Rectangle) :La méthode **getCollisionRectangle** calcule et retourne un rectangle représentant la zone de collision du joueur. Elle ne prend aucun paramètre. La position du joueur est convertie en coordonnées d'écran, et la largeur ainsi que la hauteur du rectangle sont calculées en multipliant les dimensions de la texture par l'échelle appliquée (*0.10f*). Cette méthode est particulièrement utile pour gérer les interactions et les détections de collision avec d'autres éléments du jeu.

getTexture():type(Texture2D) :La méthode **getTexture** retourne la texture utilisée pour représenter le joueur. Elle ne prend aucun paramètre et renvoie un objet de type **Texture2D**. Cette méthode permet d'accéder directement à la texture pour des besoins d'affichage ou de traitement.

getScale():type(float) :La méthode **getScale** retourne l'échelle appliquée à la texture du joueur. Elle ne prend aucun paramètre et renvoie une valeur de type float. Cette échelle est utilisée pour ajuster visuellement la taille de la texture affichée à l'écran.

~Player():Le destructeur **~Player** est appelé automatiquement lorsque l'objet **Player** est détruit. Il libère les ressources utilisées par la texture du joueur en appelant la fonction **UnloadTexture**.



La fonction principale main :

Les ressources :

● Les audios :

- ✧ Introduction : un audio pour la démarche de jeu
- ✧ Intermédiaire : un audio pendant le jeu
- ✧ Collisions (entre l'abeille et l'obstacle)
- ✧ Victoire : lorsque l'abeille arrive à la fleur

● Les arrière-plans :

- ✧ L'arrière-plan pour le menu principal
- ✧ L'arrière-plan dans le jeu

● Les images :

- ✧ Une image d'une horloge
- ✧ Une image pour l'abeille
- ✧ Une image pour la fleur
- ✧ Une image pour l'obstacle
- ✧ Une image pour afficher la victoire
- ✧ Une image pour afficher lorsque le jeu est arrêté



● Les objets :

- ✧ Obstacles :
 - ✓ Une plante : lorsque l'abeille entre en collision avec la plante, elle revient au départ
- ✧ Boutons :
 - ✓ Play / Exit : pour jouer ou quitter le jeu
 - ✓ Next / Replay : pour continuer ou réinitialiser la partie
 - ✓ Song / No song : pour activer ou désactiver les audios
 - ✓ Stop / Continue : pour arrêter le jeu ou continuer (arrêter le chronomètre)
 - ✓ Home : pour revenir au menu principal





- ✓ Easy / Medium / Hard : pour les niveaux
- ✓ Level : pour afficher un message (Choisir un niveau)

- ✧ Chronomètre : afficher la durée de la partie
- ✧ Score : afficher le score actuel et le meilleur score (le score minimum)
- ✧ Joueur : Abeille
- ✧ Labyrinthe : Maze
- ✧ L'objectif : Fleur



● Les variables booléennes :

Pour organiser le code et manipuler la logique du jeu

● Le menu principal :

- ✧ Deux boutons :
 - ✓ PLAY
 - ✓ EXIT
- ✧ Arrière-plan

● Le deuxième menu :

- ✧ Trois boutons distinguant les niveaux (**facile**, **moyen**, **difficile**)
- ✧ Bouton Home pour revenir au menu principal

● Décharger les ressources :

Supprimer toutes les ressources (les audios, les images)

La boucle principale de jeu :

● Afficher le menu principal :

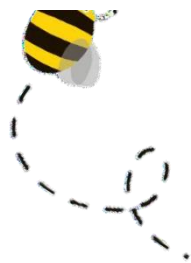
- ✧ Gestion des boutons :

Si on clique sur le bouton **PLAY**, il affiche le deuxième menu (les niveaux de jeu).

Si on clique sur le bouton **EXIT**, il ferme la fenêtre de jeu.

● Afficher le deuxième menu :

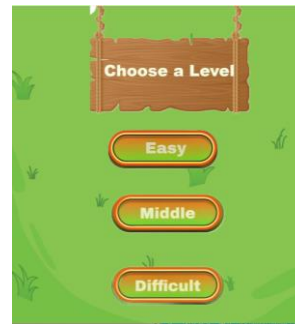




Afficher l'arrière-plan correspondant au jeu.

✧ Gestion des boutons :

✓ Si vous cliquez sur le bouton EASY :



Active l'audio correspondant (de jeu) et arrête l'audio de l'introduction, définit le niveau facile, appelle la fonction **drawCompleteMaze()** et ajoute l'obstacle (vent) pour ralentir le mouvement de l'abeille.

✓ Si vous cliquez sur le bouton MEDIUM :

Active l'audio correspondant (de jeu) et arrête l'audio de l'introduction, définit le niveau moyen, appelle la fonction **drawCompleteMaze()** et ajoute la fonction **addcycles()** pour ajouter de la complexité au labyrinthe. Ajoute l'obstacle (plante).

✓ Si vous cliquez sur le bouton HARD :

Active l'audio correspondant (de jeu) et arrête l'audio de l'introduction, définit le niveau difficile, appelle la fonction **drawNearMaze()** pour afficher un labyrinthe caché sous un arrière-plan noir pour augmentant la complexité.

● Au début du jeu :

- ✧ Mise à jour du chronomètre (pour chaque partie de jeu) puis affichage.
- ✧ Affichage de l'abeille, de la fleur, et des boutons.

● Lorsqu'il y a une collision :

- ✧ Avec la plante :

Si une collision entre l'abeille et l'obstacle survient, lance le son correspondant (**collisionSound**), et change la position de l'abeille au départ tout en affichant le texte "Oooops".

● Réinitialiser la partie :

Réinitialiser le chronomètre, le labyrinthe, et la position de l'abeille. Démarrer à nouveau le chronomètre.



- Arrêter et continuer le jeu :

Arrêter le chronomètre et afficher une image contenant le bouton "Continuer", ce qui permet de reprendre le chronomètre.

- Le victoire :

Si l'abeille atteint la fleur, afficher le score actuel et le meilleur score avec le son correspondant (`victoireSound`) et afficher le bouton "Next".

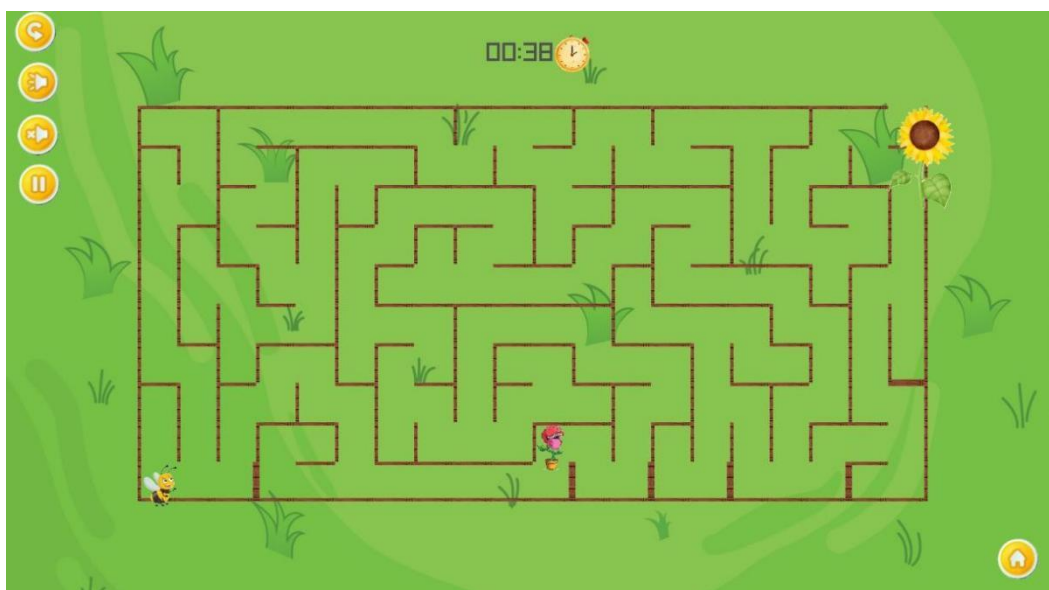
- Next :

Lorsque le joueur gagne, réinitialiser le chronomètre, la position de l'abeille, le labyrinthe et afficher le deuxième menu pour choisir le niveau ou revenir au menu principal.



CONCLUSION :

En résumé, ce projet de développement d'un jeu de labyrinthe en **C++** avec la bibliothèque **Raylib** démontre une utilisation efficace des concepts de programmation orientée objet. Grâce à la génération aléatoire de labyrinthes, une interface graphique fluide et des niveaux de difficulté progressifs, il offre une expérience ludique et immersive aux joueurs. La combinaison de la puissance de **C++** et de la flexibilité de **Raylib** permet d'assurer des performances optimales et une facilité d'utilisation. Ce projet illustre non seulement la mise en œuvre de solutions techniques avancées, mais aussi l'importance de la créativité dans le développement de jeux vidéo.





LES SOURCES :

- <https://www.remove.bg/fr/upload>
- <https://notube.net/ar/youtube-app-48>
- <https://clideo.com/editor/>
- <https://www.vecteezy.com/>
- <https://vocalremover.org/ar/cutter>

- <https://online-audio-converter.com/>
- <https://youtu.be/VLJITaFvHo4?si=ihXrogFYFGaEja8X>
- <https://youtu.be/K7vaT8bZRuk?si=eEMv-kF8DZuKi92M>
- <https://www.youtube.com/watch?v=Ah0UJTxAxg>
- <https://www.youtube.com/watch?v=YSFFB2bg40s>
- <https://www.youtube.com/watch?v=X5yyCSqr6tE&pp=ygUSU291bmQgZWZmZWNOIC0qV2lu>
- <https://www.youtube.com/watch?v=teUWsONJkk8>
- https://youtu.be/xy_NKN75Jhw?si=2820s9-PP8pUuvkJ
- <https://www.youtube.com/watch?v=rUK2pER87VE>
- <https://www.youtube.com/watch?v=rUK2pER87VE>
- <https://www.vectorstock.com/royalty-free-vector/daisy-chamomile-flowers-vector-2043470>
- <https://www.shutterstock.com/image-vector/honeycombs-background-isolated-vector-illustration-2496758481>
- <https://cloudconvert.com/avif-to-png>