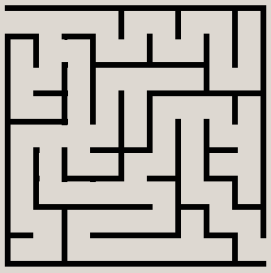


MAZE GAME USING C++

Rapport de Projet

USING RAYLIB

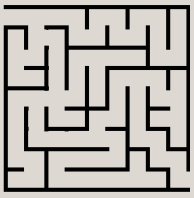
RÉALISÉ PAR :

AMAMI Yousra

ZARQI Ezzoubair

AZZAOUI Mohamed Omar

ENCADRÉ PAR : M.IKRAM BEN ABDEL OUAHAB



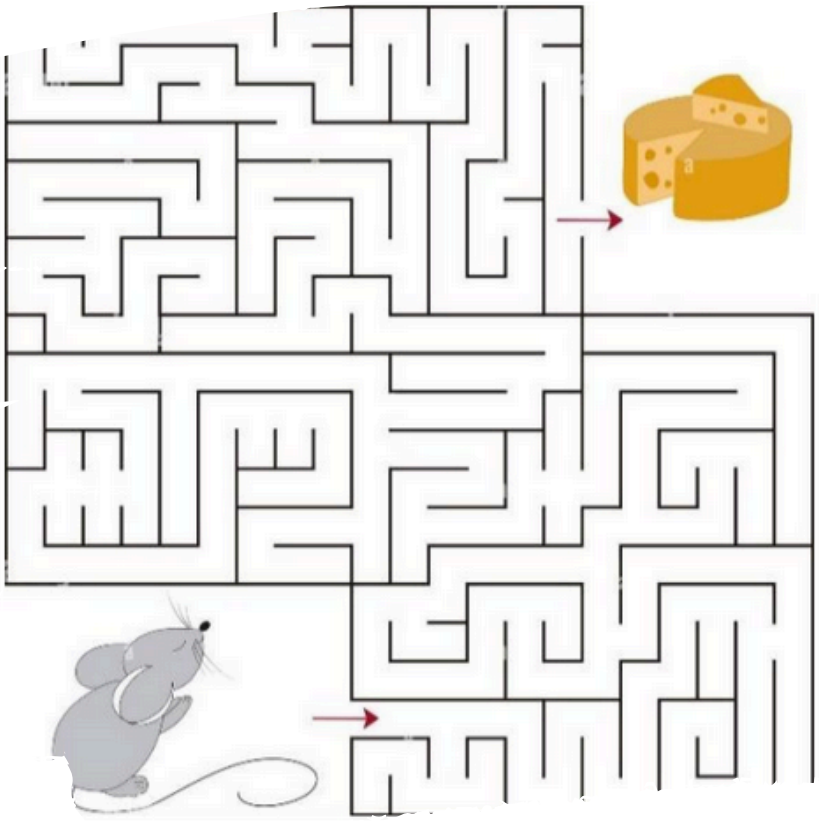
SOMMAIRE

1. Introduction générale

Le jeu Labyrinthe 2D est un jeu où le joueur doit naviguer à travers un labyrinthe généré aléatoirement. L'objectif est de trouver la sortie tout en évitant les obstacles et en utilisant les contrôles pour se déplacer dans un environnement 2D. À chaque partie, un nouveau labyrinthe est créé, offrant ainsi un défi unique à chaque fois.

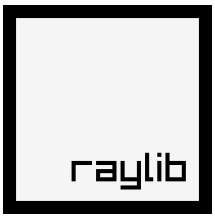
Le jeu utilise l'algorithme de génération de labyrinthes, ainsi qu'un système de détection de collisions pour garantir que le joueur ne puisse pas traverser les murs. De plus, un effet visuel 2.5D a été ajouté pour donner l'illusion de profondeur et rendre le jeu visuellement plus intéressant.

Le joueur peut choisir parmi plusieurs niveaux de difficulté, ce qui permet d'adapter l'expérience en fonction de ses compétences. Le but du jeu est simple, mais le défi réside dans la résolution du labyrinthe et la gestion des différents obstacles qui se présentent à lui.



1. Introduction générale

Les Outils utilisé :



Raylib est une bibliothèque graphique open-source en C, conçue pour créer des jeux 2D et 3D facilement. Elle est simple à utiliser, légère et idéale pour les débutants, tout en offrant des fonctionnalités puissantes pour les développeurs expérimentés.

CMake est un outil multiplateforme qui génère des fichiers de build (Makefiles, Visual Studio, etc.) à partir de fichiers CMakeLists.txt, simplifiant la configuration et la compilation des projets C/C++.

Visual Studio Code est un éditeur de code extensible développé par Microsoft pour Windows, Linux et macOS.

Zed Editor est un éditeur de code moderne et collaboratif, conçu pour être rapide, minimaliste et efficace. Il met l'accent sur la collaboration en temps réel et une expérience utilisateur fluide, idéal pour les développeurs travaillant en équipe.

Git est un logiciel de gestion de versions décentralisé. C'est un logiciel libre et gratuit, créé en 2005 par **Linus Torvalds**, auteur du noyau Linux .

GitHub est une plateforme de gestion de code basée sur Git, permettant de collaborer, versionner et héberger des projets. Elle est idéale pour les développeurs et équipes pour partager, réviser et déployer du code.

1. Introduction générale

Objectif

Le jeu s'articule autour de l'idée d'explorer un labyrinthe généré aléatoirement. Le joueur doit naviguer à travers des couloirs, éviter les murs et atteindre la sortie. L'ajout de niveaux de difficulté, d'effets visuels et de fonctionnalités interactives enrichit l'expérience utilisateur.

Langage de programmation

Le jeu est développé en C++ pour ses performances et sa flexibilité.

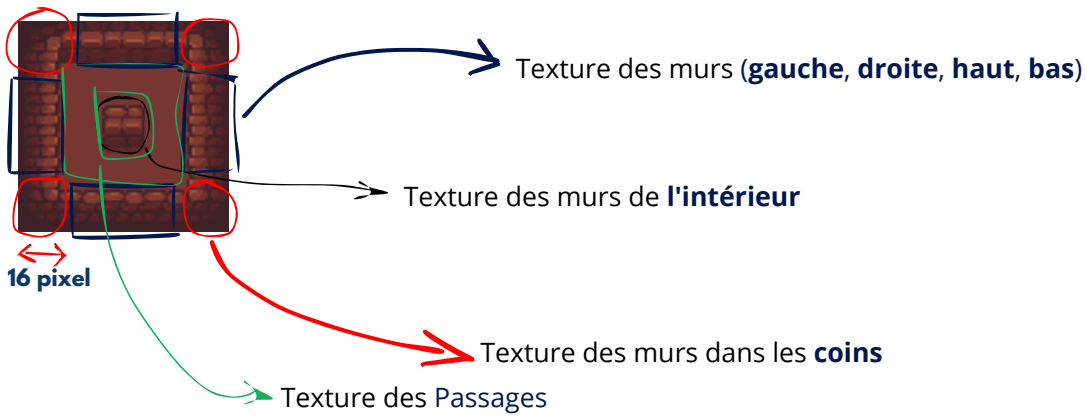
Textures et Effets

Des textures spécifiques ont été conçues pour les murs et le joueur, avec une perspective 2.5D pour simuler de la profondeur

Paradigm Utilisé

Dans ce jeu on a utilisé la Programmation Orienté Objet pour assuré la réutilisation du code, l'évolutivité et l'efficacité.

Les Texture Utilisé Pour le Labyrinthe



Remarque : il y a d'autre Texture utilisé pour les bouton ... etc qui n'est pas mentionné ici.

2. Conception et Développement

Structure de Projet :

- **build** → Fichiers de construction générés automatiquement par Cmake .
- **libs** → Répertoire contient la bibliothèque RAYLib .
- **Resources** → Les Textures utilisé et les fichiers audio etc...
- **src** → Répertoire contient les modules et le code de jeu.
- CMakeLists.txt → Le fichier de Configuration CMake .
- data.ini → Le fichier pour enregistrer les paramètre de jeu.
- LICENSE → Licence pour les droits d'auteur .
- Readme.md → Fichier réservé pour Github

On sépare le jeu en plusieurs classes :

Game : gérer la boucle principale et la logique du jeu, ainsi que les graphismes à l'écran.

Player : Gère les déplacements et animations du joueur et la collision avec les murs.

Maze : Génère et stocke la structure du labyrinthe et la logique de la rendre dans l'écran

Manager : gérer les variables globales et l'état du jeu.

Menus : Contient la logique pour dessiner l'interface graphique du jeu.

Window : gérer l'objet fenêtre, la taille, le titre etc... .

Timer : le minuteur à l'intérieur du jeu.

Button : Une sous-classe utilisée dans la classe Menus pour créer une interface graphique.

Remarque :

Les classes sont séparer en fichier entête (.h) et fichier implémentation (.cpp).

2. Conception et Développement

Implémentation :

L'algorithme du génération automatique (DFS) :

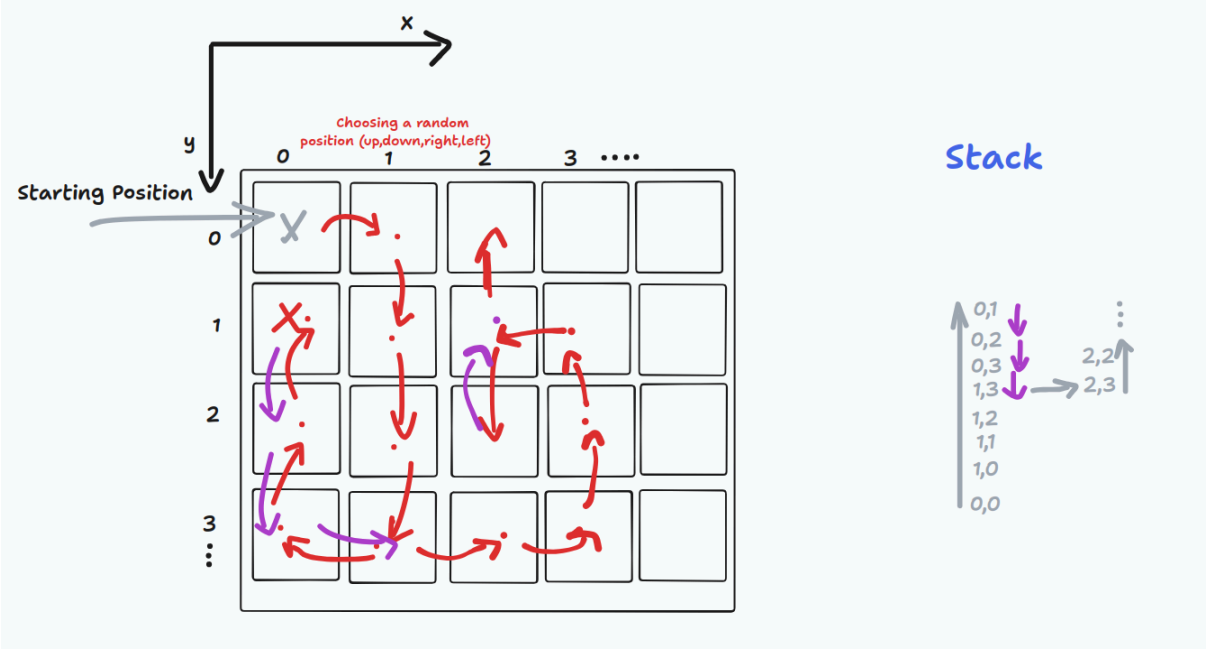
On commence d'abord par L'algorithme utilisé pour la génération du labyrinthe , ce qui le FDS (First Depth Search).

L'algorithme DFS est une méthode de parcours en profondeur utilisée dans les graphes. Lorsqu'il est appliqué à une grille bidimensionnelle représentant un labyrinthe, l'algorithme :

1. Part d'une cellule initiale choisie aléatoirement.
2. Explore récursivement les cellules voisines en creusant un chemin.
3. Revient en arrière (backtracking) lorsque toutes les cellules voisines ont été explorées.

Avantages du DFS pour un Labyrinthe :

- Produit des labyrinthes parfaits (un seul chemin entre deux points).
- Crée des labyrinthes variés grâce à l'exploration aléatoire des directions.
- Simple à implémenter et performant même sur de grandes grilles.



2. Conception et Développement

Représentation du Labyrinthe :

Le labyrinthe est représenté par une grille bidimensionnelle où :

- Chaque cellule peut être soit un mur (1), soit un chemin (0) ou (2) pour indiquer la cellule de sortie .
- Initialement, toutes les cellules sont considérées comme des murs.

Principe de Base :

L'algorithme DFS repose sur une exploration en profondeur. Cela signifie qu'il privilégie l'exploration complète d'une branche avant de revenir en arrière pour explorer d'autres possibilités.

Étapes de Génération :

A) Initialisation :

1. Choisir une cellule de départ aléatoire dans la grille.
2. La marquer comme "chemin" (valeur 0).
3. Utiliser une pile (stack) pour stocker les cellules à explorer.

B) Exploration des Cellules :

1. Tant qu'il reste des cellules dans la pile :
   - Récupérer la cellule située au sommet de la pile (cellule courante).
   - Identifier les cellules voisines de la cellule courante situées à deux cases de distance et encore marquées comme "mur".
   - Mélanger les directions aléatoirement pour garantir la diversité.
2. Si des voisins valides existent :
   - Choisir un voisin au hasard.
   - Creuser un chemin entre la cellule courante et le voisin en supprimant le mur intermédiaire.
   - Ajouter le voisin à la pile pour l'explorer plus tard.
3. Si aucun voisin valide n'existe :
   - Retirer la cellule courante de la pile (backtracking) et revenir à la cellule précédente.

2. Conception et Développement

C) Terminaison :

- L'algorithme se termine lorsque la pile est vide, indiquant que toutes les cellules accessibles ont été explorées et que le labyrinthe est complet.

Conclusion:

L'algorithme DFS est une solution simple et efficace pour la génération de labyrinthes, adaptée à des environnements interactifs comme les jeux vidéo. Sa logique d'exploration en profondeur assure des labyrinthes à la fois fonctionnels et imprévisibles, offrant une expérience immersive pour les joueurs.

Gestion des Niveaux de Difficulté :

La gestion des niveaux de difficulté dans un jeu de labyrinthe est essentielle pour garantir une expérience adaptée à différents profils de joueurs. L'objectif est d'offrir des labyrinthes simples pour les débutants (niveau Facile) tout en proposant des défis plus complexes pour les joueurs expérimentés (Moyen et Difficile).

En complément, l'ajout d'une clé de sortie augmente la profondeur du gameplay. Ce système oblige le joueur à explorer le labyrinthe pour trouver une clé, nécessaire pour déverrouiller la sortie et terminer le niveau.

Concept et Définition des Niveaux :

Niveaux Facile :

- Moins de cellules explorées .
- Plus de chemins alternatifs, donc moins de sections fermées.
- Une distance courte entre l'entrée et la sortie.
- La clé est placée près de l'entrée ou sur un chemin direct

Niveaux moyen :

- Une exploration plus dense, augmentant le nombre de chemins fermés.
- Une distance modérée entre l'entrée et la sortie.
- La clé est placée à une distance modérée, nécessitant une exploration.

2. Conception et Développement

Niveaux difficile :

- Un labyrinthe complexe avec un nombre maximal de chemins fermés.
- De longs parcours entre l'entrée et la sortie.
- La clé est éloignée dans une zone complexe, forçant le joueur à naviguer plusieurs chemins.

Le placement de la sortie : La sortie peut être déverrouillée uniquement après avoir récupéré la clé.

l'implémentation des Niveaux :

Paramétrage des Niveaux

Les niveaux de difficulté sont définis par des paramètres spécifiques appliqués à l'algorithme de génération de labyrinthe. Ces paramètres influencent :

1. La densité de murs.
2. Le nombre de cellules accessibles.
3. La distance entre l'entrée et la sortie.
4. La clé de sortie est placée stratégiquement pour forcer le joueur à s'aventurer loin dans le labyrinthe avant de pouvoir accéder à la sortie

Partie Technique :

Pour la partie technique expliquant le fonctionnement du classe **Maze**, ce classe faire la gestion des difficultés et la génération du labyrinthe de manière automatique.

Attributs :

- Dimensions de la grille (__width, __height).
- Stockage des textures (wallTexture, floorTexture, etc.).
- Grille de la carte (maze), un vecteur 2D:
  - 1 représente un mur.
  - 0 représente un chemin.
  - 2 représente la sortie.
  - 3 représente la position de la clé.
- Données des textures importées depuis des fichiers externes.

Génération du Labyrinthe :

- La grille initialement est entièrement remplie de murs (1).
- génération du labyrinthe.
- Marquage d'une cellule aléatoire avec la clé (3).

2. Conception et Développement

Personnalisation de la Difficulté :

- Trois niveaux de difficulté (EASY_DIFF, MEDIUM_DIFF, HARD_DIFF) :
  - Ajustement des dimensions de la grille.
  - Les dimensions sont forcées en nombres impairs pour garantir des chemins valides.
- Réinitialisation complète (resetMaze) en fonction de la difficulté choisie.

Gestion des Événements et Propriétés :

- Méthode unlockMaze() pour générer une sortie aléatoire (gauche, droite ou bas).
- Vérification si une cellule est un mur avec isWall().
- Affichage conditionnel (en fonction de l'état de showKey).

Débogage et Visualisation :

- Méthode printMazeToConsole() :
  - Représentation textuelle de la grille dans la console pour déboguer.
  - Symboles utilisés : # pour les murs, * pour la clé, et espaces pour les chemins.

Gestion Du joueur :

La classe Player est un élément central dans votre jeu de labyrinthe. Elle représente le personnage contrôlé par l'utilisateur et gère ses interactions avec l'environnement. Cela inclut les déplacements du joueur, la gestion des états d'animation (comme marcher ou rester immobile), le suivi des collisions avec les murs du labyrinthe, et le rendu graphique de la texture du joueur à l'écran.

État immobile STATE_IDLE | État déplacé à gauche STATE_MOVING_LEFT | État déplacé en haut STATE_MOVING_UP | État déplacé à droite STATE_MOVING_RIGHT | État déplacé en bas STATE_MOVING_DOWN

2. Conception et Développement

Vue Théorique générale :

La classe Player repose sur plusieurs concepts clés :

1. Gestion des États :

Le joueur peut être dans différents états, tels qu'immobile (idle) ou en mouvement dans une direction spécifique. Ces états sont représentés par des constantes (STATE_IDLE, STATE_MOVING_DOWN, etc.) et déterminent quelle texture doit être affichée à l'écran.

2. Interaction Utilisateur :

Le joueur répond aux entrées clavier, telles que les flèches directionnelles ou les touches W, A, S, et D. Chaque touche correspond à une direction, et un appui sur une touche met à jour la position et l'état du joueur.

3. Détection de Collisions :

Le joueur ne peut pas traverser les murs du labyrinthe. Cela est assuré par des fonctions de détection de collision, qui vérifient si les coordonnées du joueur croisent celles d'une cellule murale dans la grille du labyrinthe.

4. Rendu Graphique :

Le joueur est représenté à l'écran par une texture, dont la position est calculée en fonction de sa position sur la grille et d'un facteur d'échelle. Le rendu utilise la bibliothèque **Raylib**, notamment la fonction DrawTexturePro().

5. Gestion de la Caméra :

Pour centrer l'attention sur le joueur, la caméra suit sa position dans le labyrinthe. Cela crée un effet de suivi fluide et améliore l'immersion.

2. Conception et Développement

Partie Technique :

1. Attributs :

- Position et Mouvement :
  - posX et posY : Coordonnées en pixels du joueur.
  - speed : Vitesse de déplacement en pixels.
- Textures :
  - Différentes textures (playerIdle, playerMovingUp, etc.) sont chargées pour représenter visuellement les mouvements ou l'état d'immobilité.
- État :
  - state : Variable indiquant l'état actuel du joueur (immobile ou en mouvement dans une direction).

2. Constructeur et Destructeur :

- Initialise les attributs par défaut, comme la position, la vitesse et l'échelle, et aussi le chargement des textures.
- Assure un nettoyage éventuel des ressources (non implémenté ici, mais pourrait inclure UnloadTexture()).

3. Déplacements et Contrôles :

- Les fonctions updatePlayer() et updatePlayerControls() gèrent le déplacement du joueur :
  - Identifient la direction du déplacement selon l'entrée clavier.
  - Vérifient les collisions avant de valider le déplacement.
- Fonctions de Déplacement : Les méthodes comme moveUp(), moveDown(), etc., ajustent les coordonnées du joueur en fonction de la direction.

4. Détection des Collisions :

- Les fonctions comme isCollidingTop(), isCollidingBottom(), etc., détectent si le joueur entre en contact avec un mur.
- Elles utilisent les coordonnées du joueur pour vérifier les cellules du labyrinthe correspondant à sa position.

2. Conception et Développement

5. Rendu Graphique et Gestion de la Caméra :

- La méthode renderPlayer() :
  - Sélectionne la texture appropriée en fonction de l'état du joueur.
  - Calcule la position et la taille du sprite à l'écran.
  - Utilise DrawTexturePro() pour dessiner le sprite avec précision.
- La position de la caméra est mise à jour pour suivre le joueur.

Gestion de l'état de jeu :

La classe Manager est conçue pour centraliser la gestion des états et des écrans dans le jeu. Elle joue un rôle de chef d'orchestre, coordonnant les transitions entre les différentes phases (comme le menu principal, le jeu, ou les écrans de victoire/défaite), tout en gérant des paramètres globaux comme la pause, les options de difficulté, et la gestion des fichiers de configuration.
Rôles Principaux :

- Gestion des Écrans : Suivre et modifier l'écran actif à tout moment, que ce soit le menu principal, l'écran du jeu, ou d'autres écrans.
- Gestion des États du Jeu : Gérer les états globaux comme la pause (isPaused), la victoire (wining), et la demande de sortie (exitGame).
- Interactions Globales : Agir comme un point d'accès centralisé pour vérifier ou modifier des états globaux, comme la difficulté ou si la fenêtre doit être fermée.

Partie Technique :

1. Attributs :

- Gestion des Écrans :
  - currentScreen : Identifie l'écran actif (exemple : menu principal ou jeu).
  - Défini avec des constantes (MAIN_MENU_SCREEN et GAME_SCREEN) pour plus de lisibilité et d'évolutivité.

2. Conception et Développement

- Gestion des Écrans :
    - currentScreen : Identifie l'écran actif (exemple : menu principal ou jeu).
    - Défini avec des constantes (MAIN_MENU_SCREEN et GAME_SCREEN) pour plus de lisibilité et d'évolutivité.

- États Globaux :
    - exitGame : Indique si le jeu doit se fermer.
    - isPaused : État du jeu (1 pour pause, 0 pour en cours).
    - showDifficlttyMenu : Indique si le menu de difficulté doit être affiché.
    - wining : État de victoire du joueur.
    - windowExitRequested : Indique si l'utilisateur a demandé la fermeture de la fenêtre via l'interface (exemple : clic sur le bouton "ESC").

2. Methodes :

- Gestion des Écrans :
    - setScreen(int screen) : Permet de définir l'écran actif, utile pour basculer entre différentes parties du jeu (menu, gameplay, etc.)
    - getScreen() : Retourne l'écran actuellement actif pour que d'autres parties du code puissent adapter leur comportement.

- Gestion de l'État de Victoire :
    - getWinState() : Retourne si le joueur a gagné (1) ou non (0).
    - changeToWinState() :Passe l'état de victoire à 1. Cela peut déclencher des changements d'écran ou des animations.
    - resetWinState() :Réinitialise l'état de victoire à 0, utile lors du redémarrage d'une partie.

2. Conception et Développement

Gestion de jeu :

La classe Game représente le cœur du jeu, combinant tous les éléments essentiels comme la fenêtre, le joueur, le labyrinthe, les menus et la gestion des états pour offrir une expérience interactive. Elle orchestre le déroulement du jeu à travers des boucles principales, des dessins, et des interactions utilisateur, tout en gérant la caméra et l'échelle de rendu.
La classe Game agit comme un conteneur et un coordinateur pour les différents modules du jeu. Elle contient les principales composantes nécessaires au fonctionnement du jeu, notamment :

1. Rendu Graphique :
   - La méthode DrawGame() s'occupe du rendu des différents éléments du jeu (joueur, labyrinthe, etc.).
2. Coordination des Modules :
   - Centralise la gestion des objets clés : Window, Maze, Player, Manager, et Menu.
   - Le Manager gère les états globaux (menu, pause, écran de victoire) et les transitions.
3. Gestion des Ressources :
   - Les pointeurs vers des objets (Window, Maze, etc.) permettent une gestion centralisée, simplifiant les interactions entre modules.
4. Boucle Principale :
   - La méthode Loop() est la boucle de jeu principale qui orchestre les mises à jour des éléments, les interactions, et les rendus graphiques.

Autres Classes

Le projet comprend plusieurs classes complémentaires pour enrichir l'expérience utilisateur :

- Classe Menu : Gère l'interface graphique des menus (menu principal, pause, options). Elle facilite la navigation entre les différents écrans interactifs du jeu.

- Classe Button : Représente des boutons interactifs utilisés dans les menus pour permettre les sélections ou interactions avec l'utilisateur.

2. Conception et Développement

- Classe Timer : Ajoute un compteur dans le jeu pour suivre la durée d'une session ou mesurer un temps limite.

- Classe Window : Responsable de la gestion de la fenêtre du jeu, notamment la création, le redimensionnement, et la gestion des paramètres visuels.

Ces classes travaillent ensemble pour offrir une interface utilisateur intuitive et fonctionnelle, tout en assurant une gestion fluide de l'affichage et des interactions.

En Résumé :

La conception et le développement de ce projet ont été guidés par une approche modulaire et orientée objet, favorisant la clarté, la réutilisabilité, et l'évolutivité du code. Chaque classe joue un rôle spécifique et bien défini dans l'architecture globale, garantissant une séparation des responsabilités :

- Gameplay et interaction sont assurés par les classes telles que Player, Maze, et Manager.
- Interface utilisateur est prise en charge par les classes Menu et Button.
- Gestion des ressources et des systèmes de base, comme la fenêtre, le temps, ou les transitions, est confiée aux classes Window, Timer, et Game.

Grâce à cette organisation, le projet est facile à maintenir, à modifier, et à enrichir avec de nouvelles fonctionnalités. Les principes de conception adoptés, combinés à l'utilisation de **Raylib** et de bonnes pratiques de programmation, garantissent un jeu performant et engageant, prêt à offrir une expérience utilisateur fluide et agréable.

Ce développement met également en avant la puissance d'une architecture bien pensée, qui permet une collaboration efficace entre les différentes composantes du jeu, tout en restant flexible pour des évolutions futures.

3. Fonctionnalités principales

Pour implémenter les fonctionnalités principales du jeu, nous avons structuré le développement en plusieurs étapes clés. Voici comment chaque fonctionnalité a été conçue et intégrée :

1. Génération dynamique du labyrinthe

- Utiliser un algorithme procédural pour générer un labyrinthe unique à chaque partie. Cela assure que le joueur affronte un défi différent à chaque session.
- Les labyrinthes devront respecter des règles de connexité : chaque position dans le labyrinthe doit être atteignable et une solution doit exister.
- Une clé est générée à un emplacement aléatoire dans le labyrinthe.
- La clé est essentielle pour débloquer la sortie. Le joueur doit la récupérer avant de pouvoir terminer le niveau.

2. Niveaux de Difficulté Ajustables

- Facile :
  - La clé est placée à proximité du chemin principal, facilement visible.
  - Peu d'impasses pour éviter que le joueur se perde.
- Moyen :
  - La clé est placée dans une zone légèrement éloignée du chemin principal.
  - Plusieurs impasses et bifurcations rallongent le temps de recherche.
- Difficile :
  - La clé est cachée dans une zone complexe avec de multiples chemins trompeurs.
  - Une exploration approfondie est nécessaire pour la trouver.

3. Mécanique de Jeu Mise à Jour avec la Clé

- Départ et Objectif :
  - Le joueur commence à l'entrée et doit récupérer la clé avant d'atteindre la sortie.
- Fonctionnement de la clé :
  - Lorsqu'il atteint la position de la clé, un message ou une animation indique que la clé a été collectée.
  - La sortie devient alors accessible. Avant cela, le joueur ne peut pas terminer le niveau.

3. Fonctionnalités principales

- Réinitialisation de la Partie :
  - Si le joueur réinitialise, la clé et la sortie sont déplacées à de nouvelles positions aléatoires dans le labyrinthe.

4. Interface Graphique et Feedback Visuel

- Affichage de la clé :
  - La clé est représentée par une icône distincte visible sur la carte (par exemple, une petite clé dorée).
  - Lorsqu'elle est collectée, elle disparaît de l'écran et un indicateur visuel (ou texte) confirme sa récupération.
- État de la sortie :
  - Avant la collecte de la clé, la sortie est bloquée (par exemple, avec un symbole de porte fermée).
  - Après la collecte, la sortie s'ouvre et devient accessible.

L'ajout de la contrainte de clé enrichit la mécanique de jeu en augmentant la difficulté et en introduisant une étape supplémentaire. Cela encourage l'exploration et la réflexion stratégique, particulièrement dans les niveaux moyen et difficile. La clé devient un élément central du gameplay, modifiant à la fois la génération du labyrinthe et l'expérience utilisateur.

4. Capture d’écran du jeu

Menu Principale

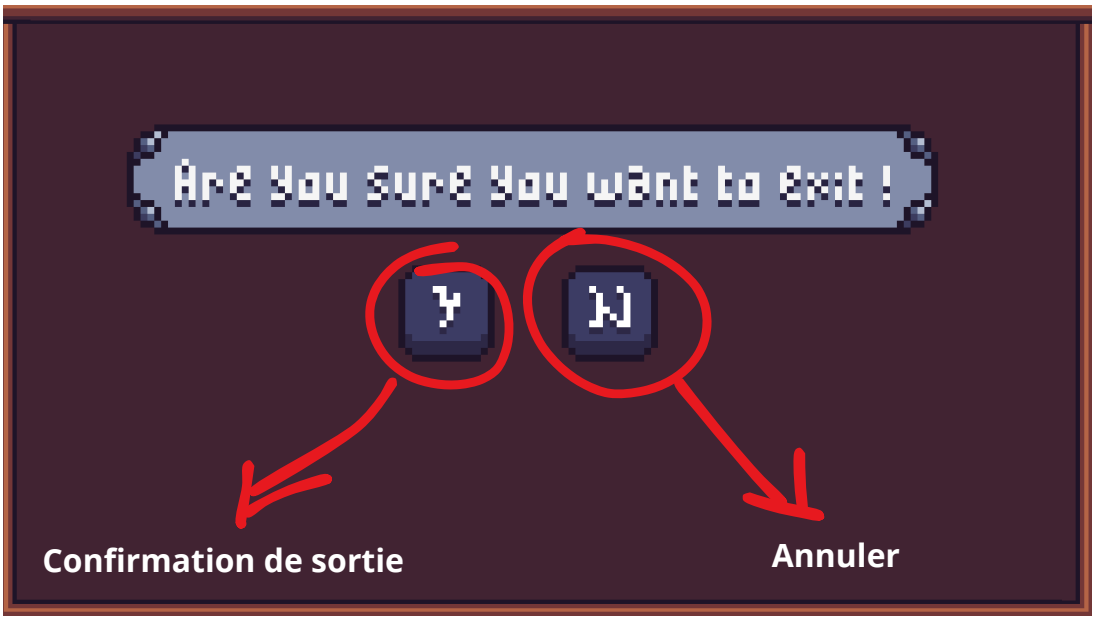
Menu des Difficultés

4. Capture d'écran du jeu

Écran de jeu

Écran de Victoire

4. Capture d'écran du jeu



Écran de confirmation de Sortie

5. Conclusion

Le projet du jeu de labyrinthe représente une application complète et pratique des concepts fondamentaux de la programmation orientée objet (POO) et de la conception logicielle en C++. En combinant des éléments techniques et créatifs, ce projet a permis de concevoir un jeu captivant et interactif, tout en mettant en œuvre des fonctionnalités robustes et optimisées.

Grâce à l'utilisation de la bibliothèque Raylib, une interface graphique conviviale et immersive a été développée, permettant au joueur d'explorer un environnement en 2D agréable et engageant. Les principales fonctionnalités, telles que la génération procédurale des labyrinthes, les niveaux de difficulté adaptatifs, la contrainte de clé pour débloquer la sortie, et l'intégration d'un chronomètre, enrichissent l'expérience utilisateur en offrant un gameplay stimulant et varié.

> “

Le projet a également mis en évidence l'importance de la modularité et de la réutilisabilité dans la conception logicielle. Chaque classe, comme celles pour le joueur, le labyrinthe, l'interface graphique ou encore le gestionnaire d'états, a été conçue de manière indépendante, favorisant la lisibilité, la maintenance et l'évolutivité du code. Cette approche garantit la possibilité d'ajouter facilement de nouvelles fonctionnalités ou d'adapter le jeu à d'autres contextes à l'avenir.

En conclusion, ce projet est une réussite à la fois pédagogique et technique. Il démontre la capacité à allier des compétences en programmation, en conception d'algorithmes et en développement graphique pour produire une application ludique et fonctionnelle. Ce jeu de labyrinthe peut être utilisé comme base pour des projets futurs, comme l'ajout d'intelligence artificielle, de modes multijoueurs, ou encore de thèmes et graphismes personnalisés, offrant ainsi un potentiel de développement et d'amélioration infini.

6. Référence

Raylib Website :	https://www.raylib.com
Raylib example :	https://www.raylib.com/examples.html
Raylib example :	https://www.raylib.com/examples.html
Raylib Cheatsheet :	https://www.raylib.com/cheatsheet/cheatsheet.html
Raylib github repo :	https://github.com/raysan5/raylib
Raylib github wiki :	https://github.com/raysan5/raylib/wiki
C++ Réfrence :	https://en.cppreference.com/w/