Université Abdelmalek Essaadi
Faculté des Sciences et Techniques de Tanger
Département Génie Informatique
1/2 Module : IDAI-54 Programmation Orientée Objet en C++
A.U. : 2024/2025

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

rapport de projet

PRÉPARÉ PAR :

Afailal Trebak Ghizlane
Elhouaoui Hafsa
Ahadri Botaina
Harrak Fatima Zahrae

ENCADRÉ PAR:

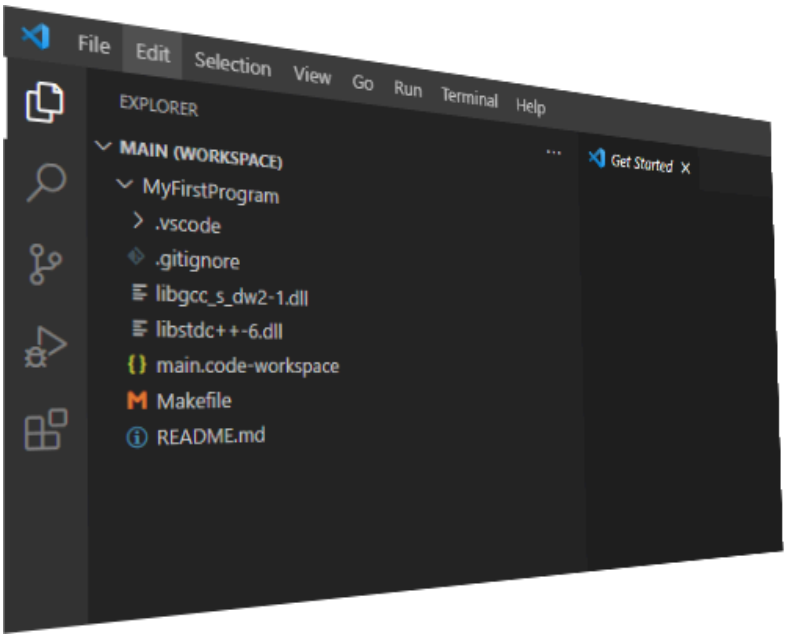
Pr. Ikram BENABDELOUAHAB

DÉVELOPPEMENT D'UN JEU DE CASSE-TÊTE DE STYLE LABYRINTHE (MAZES) EN C++ AVEC RAYLIB:

Introduction :

Ce projet est un jeu de labyrinthe développé avec la bibliothèque Raylib en C++. Le joueur commence au menu principal, puis s'immerge dans un labyrinthe généré aléatoirement, avec trois niveaux de difficulté (facile, moyen, difficile) qui influent sur la taille et la complexité. L'objectif est d'atteindre la sortie le plus rapidement possible, tout en naviguant dans un environnement captivant. Une fois le but atteint, le joueur est félicité dans l'écran de victoire. Ce jeu est conçu pour être engageant et offrir une expérience unique à chaque partie grâce à la génération procédurale des labyrinthes.

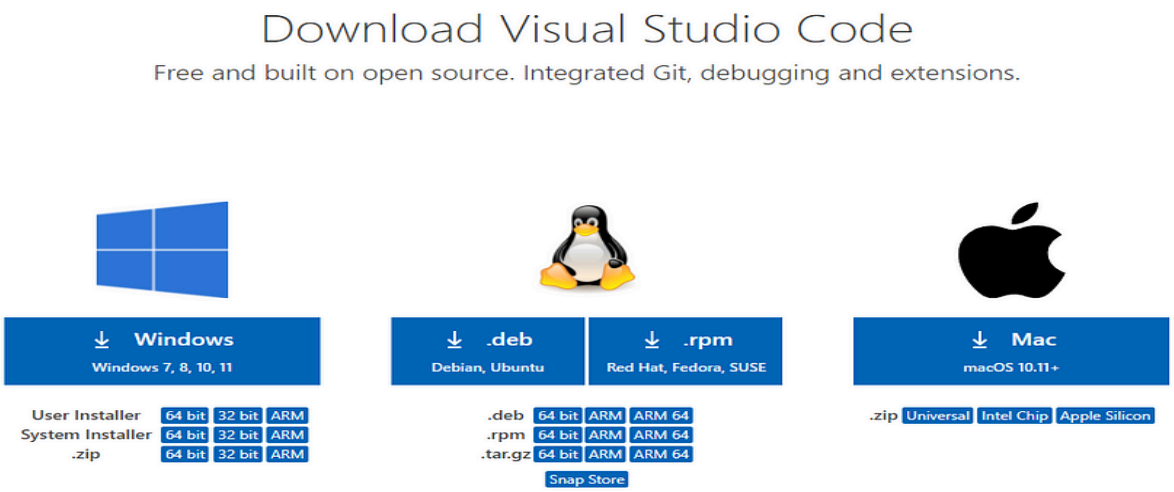
Téléchargement et Installation



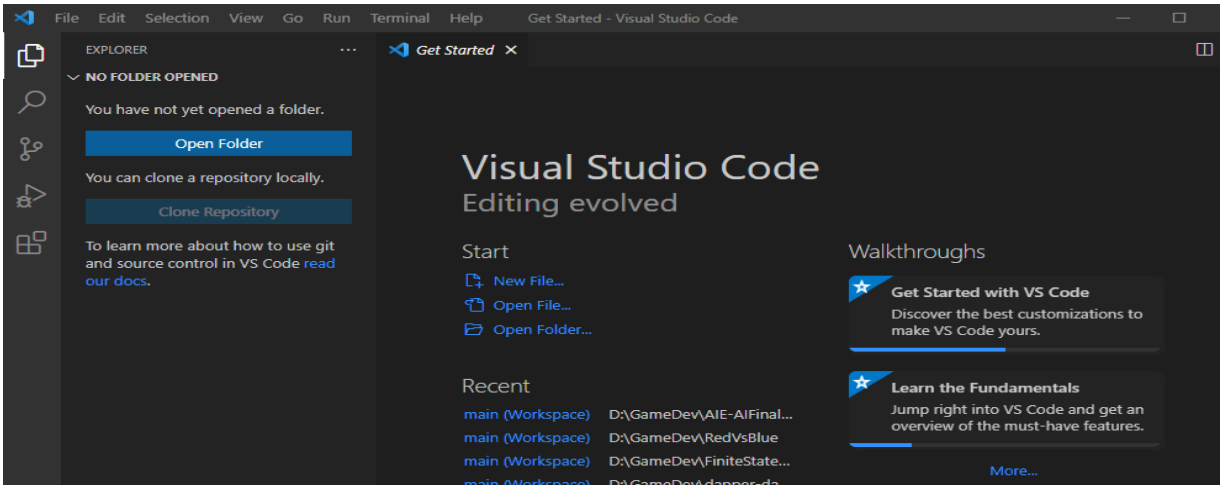
Cet article marque le début d'une série consacrée au développement de jeux en utilisant la bibliothèque raylib en C++ et Visual Studio Code.

Télécharger et Installer Visual Studio Code

Rendez-vous sur le site https://code.visualstudio.com/download et téléchargez Visual Studio Code.

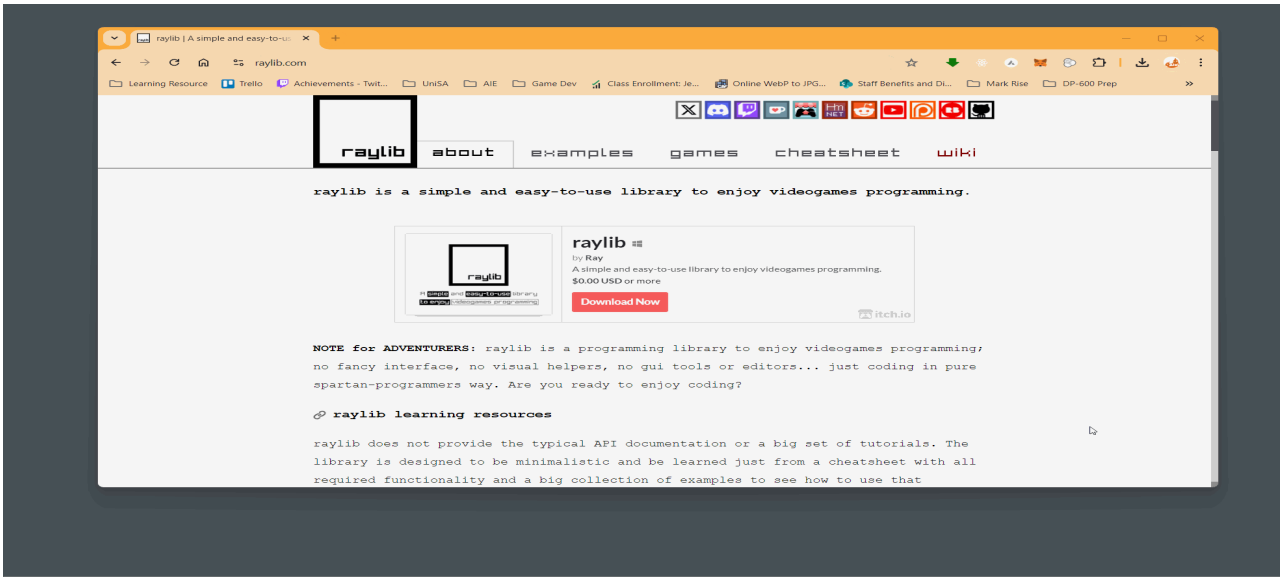


Après l'installation, cliquez sur le bouton des extensions et installez C/C++ pour Visual Studio Code de Microsoft. Tapez C/C++ dans la barre de recherche. Assurez-vous de choisir celui qui est indiqué ci-dessous avec l'étoile bleue, puis cliquez sur le bouton Installer.

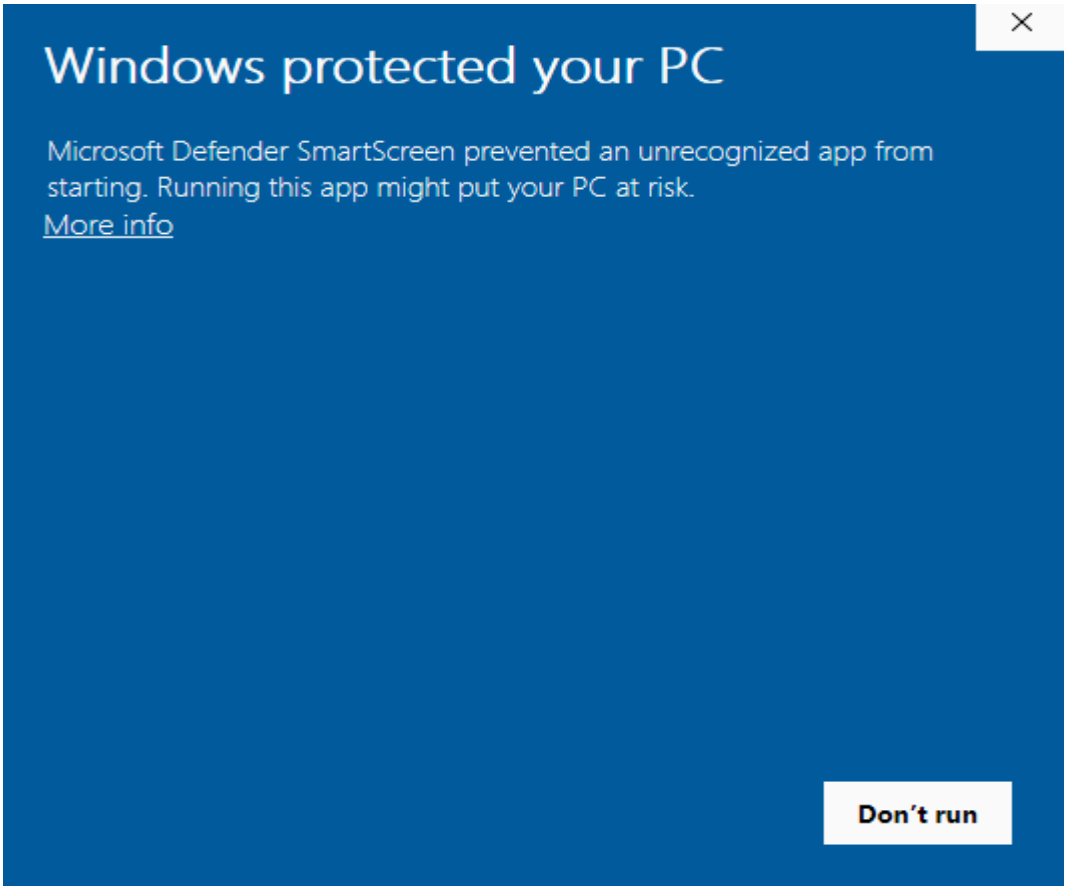


Télécharger et Installer raylib

Allez sur le site https://www.raylib.com/ et cliquez sur le bouton Télécharger maintenant. Vous pouvez faire un don, mais ce n'est pas obligatoire, cliquez sur Non merci. Choisissez raylib 5.0 Windows Installer (avec le compilateur MinGW). Si vous lisez ceci dans le futur, ce n'est pas grave si les numéros de version sont différents, assurez-vous simplement que c'est le compilateur MinGW.



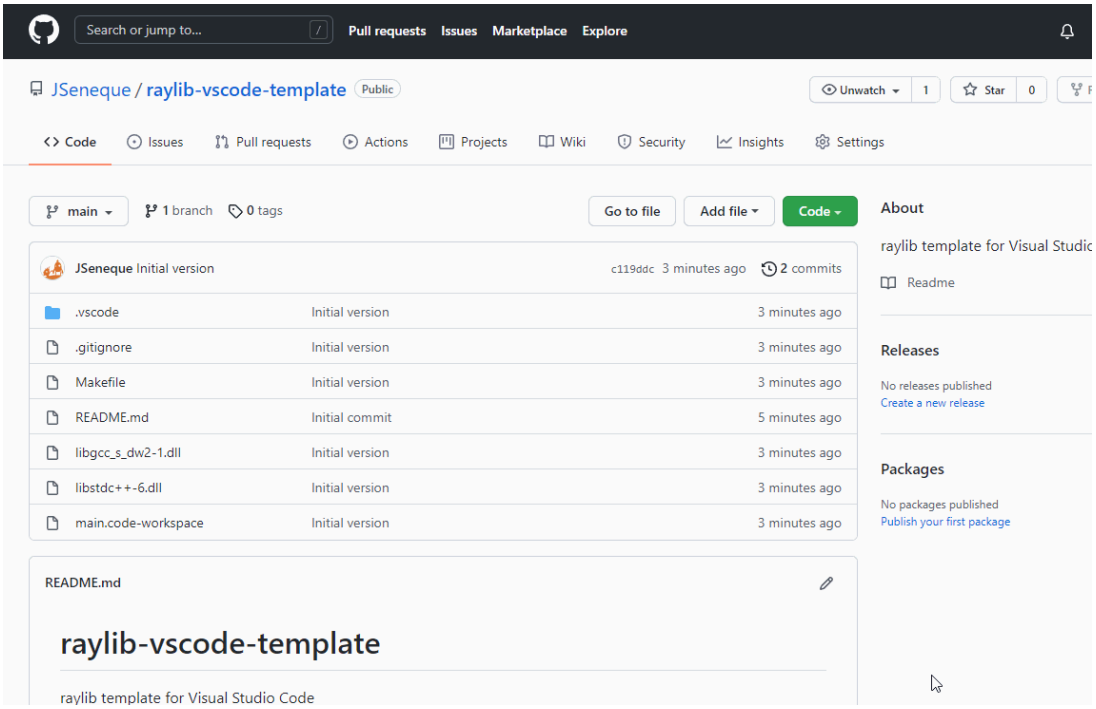
Exécutez le fichier téléchargé et si vous recevez un avertissement, cliquez sur Plus d'infos puis sur Exécuter quand même.



Suivez le processus d'installation en acceptant les paramètres par défaut. Cela créera une installation portable dans le dossier `c:\raylib`.

Télécharger le modèle raylib pour Visual Studio Code

Rendez-vous sur mon dépôt pour télécharger le zip du modèle à https://github.com/JSeneque/raylib-vscode-template.



Extrayez ce fichier zip et conservez-le quelque part, car c'est le point de départ pour chaque projet que vous réalisez. Il vous suffit de faire une copie du dossier et de le renommer.

Entrez dans le dossier nouvellement renommé et double-cliquez sur le fichier main.code-workspace. Vous recevrez un message pop-up, cliquez sur "Oui, je fais confiance aux auteurs".

Alors, nous avons maintenant tout ce dont nous avons besoin pour créer des jeux C++ en utilisant Raylib et Visual Studio Code (VSCode).

Statistiques du code:

- **Nombre de lignes de code : 307 .**
- **Nombre de classes :** 3 classes (Cell , Maze , Player) .
- **Nombre total de fonctions :32**
    1. **dans la classe `Cell` : 3 =>** `Draw(), GetCell(int x, int y ), RemoveWalls(Cell* next) .`
    2. **dans la classe `Maze` : 1=>** `GenerateMaze()`
    3. **dans la classe `Player` :4=>** `Draw(), Draw2() ,Move(int dx, int dy),GameStart(Player& goal, Player& player)`
    4. **Fonctions `globales` : 6 .**
    5. **Nombre de fonctions Raylib utilisées : 18 .**

Bibliothèques Utilisée:

- **`#include "raylib.h" :`** C'est la bibliothèque graphique centrale utilisée pour concevoir et animer le jeu. Elle fournit les fonctionnalités principales pour le développement de jeux : création de fenêtres, gestion des entrées utilisateur, dessin d'éléments graphiques, gestion des sons, etc.
- **`#include <vector> :`** Permet l'utilisation de la classe `vector`, un conteneur dynamique fourni par la STL (Standard Template Library).
- **`#include <stack>`** : Fournit la classe `stack`, un conteneur basé sur le principe **LIFO** (Last In, First Out).
- **`#include <cstdlib>` et `#include <ctime> :`**

`-cstdlib` : Contient des fonctions utilitaires comme `rand()` et `srand()` pour générer des nombres aléatoires.

`-ctime` : Fournit la fonction `time()` pour initialiser le générateur aléatoire avec l'heure actuelle, garantissant une génération différente à chaque exécution.

Les classes utilisée :

Pour gérer un labyrinthe 2D, deux classes principales sont utilisées : **Maze** et **Cell**. La classe Cell représente chaque cellule individuelle du labyrinthe, en définissant ses propriétés et ses comportements. La classe Maze, quant à elle, gère la structure globale du labyrinthe et orchestre sa génération en manipulant les instances de Cell. Il y a une relation complémentaire entre les deux classes : Maze utilise Cell pour former un réseau connecté de passages en supprimant dynamiquement les murs entre les cellules. Cette séparation permet une conception modulaire où Cell encapsule les détails d'une cellule individuelle, tandis que Maze se concentre sur l'ensemble du labyrinthe et son algorithme de génération.

◄ Class Cell :

Elle est utilisée pour gérer les propriétés et comportements de chaque cellule dans une grille 2D.

❖ les attributs :

- On a l'utilisation de trois attributs privee :
  - **x, y** : Coordonnées de la cellule (colonne et ligne).
  - **visited** : Indique si la cellule a déjà été visitée.
  - **walls** : Tableau de quatre booléens représentant les murs de la cellule (haut, droite, bas, gauche).

```
private :
    int x, y;
    bool visited;
    bool walls[4];
```

❖ les Méthodes :

- **constructeur** qui initialise les coordonnées, marque la cellule comme non visitée et active tous les murs par défaut.

```
public :
    Cell(int x, int y) : x(x), y(y), visited(false) {
        walls[0] = walls[1] = walls[2] = walls[3] = true;
    }
```

- **Draw()** : Dessine les murs de la cellule sur l'écran en utilisant la fonction DrawLine de Raylib.
  le syntaxe de DrawLine : void DrawLine(int startX, int startY,

int endX, int endY, Color color);

On appelle cette fonction quatre fois pour dessiner tous les murs. Après l'exécution de cette fonction pour chaque cellule, on obtient une grille avec tous les murs dessinés.

```
Void Draw() {
        if (walls[0]) DrawLine(x * cellSize, y * cellSize, (x + 1) * cellSize,
y * cellSize, BLACK);
        if (walls[1]) DrawLine((x + 1) * cellSize, y * cellSize, (x + 1) * cellSize,    (y +
1) * cellSize, BLACK);
        if (walls[2]) DrawLine((x + 1) * cellSize, (y + 1) * cellSize, x cellSize,     (y +
1) * cellSize, BLACK);
        if (walls[3]) DrawLine(x * cellSize, (y + 1) * cellSize, x * cellSize,          y *
cellSize, BLACK);
    }
```

- **GetCell(int x, int y) :** Récupère un pointeur vers une cellule spécifique dans une grille 2D (représentée en 1D) , Voici une explication de son fonctionnement :

Tout d'abord, la fonction vérifie si les coordonnées (x, y) ne sont pas en dehors des limites de la grille. Si elles sont valides, elle retourne un index de localisation donné par la formule suivante : index = y * cols + x;Où :

- y * cols : calcule l'offset pour atteindre la ligne y.
- x : ajoute le décalage pour atteindre la colonne x dans cette ligne.

Si les coordonnées (x, y) sont en dehors des limites de la grille, la fonction retourne nullptr, indiquant que la cellule demandée est invalide.

```
Cell* GetCell(int x, int y) {

        if (x < 0 || y < 0 || x >= cols || y >= rows) return nullptr;

        return &grid[y * cols + x]; }
```

- **RemoveWalls(Cell* next) :**

La fonction RemoveWalls(Cell* next) est utilisée pour supprimer les murs entre la cellule courante et une cellule voisine (appelée next), en fonction de leurs positions relatives. Voici une explication détaillée de son fonctionnement :

-dx est la différence entre les positions en x de la cellule courante et de la cellule voisine. Cela permet de déterminer si les cellules sont adjacentes horizontalement (gauche/droite).

De même, dy mais il est permet de savoir si les cellules sont adjacentes verticalement (haut/bas).

- Si dx == 1 : cela signifie que la cellule voisine est à gauche de la cellule courante. Le mur de gauche de la cellule courante (indice 3) et le mur de droite de la cellule voisine (indice 1) doivent être supprimés.
- Si dx == -1 : cela signifie que la cellule voisine est à droite de la cellule courante. Le mur de droite de la cellule courante (indice 1) et le mur de gauche de la cellule voisine (indice 3) doivent être supprimés.
- Si dy == 1 : cela signifie que la cellule voisine est en bas de la cellule courante. Le mur supérieur de la cellule courante (indice 0) et le mur inférieur de la cellule voisine (indice 2) doivent être supprimés.
- Si dy == -1 : cela signifie que la cellule voisine est en haut de la cellule courante. Le mur inférieur de la cellule courante (indice 2) et le mur supérieur de la cellule voisine (indice 0) doivent être supprimés.

```
void RemoveWalls(Cell* next) {

        int dx = x - next->x;

        int dy = y - next->y;

        if (dx == 1) {

            walls[3] = false; next->walls[1] = false;

        } else if (dx == -1) {

            walls[1] = false;  next->walls[3] = false;

        }

        if (dy == 1) {

            walls[0] = false; next->walls[2] = false;

        } else if (dy == -1) {

            walls[2] = false;  next->walls[0] = false;

        }

    }
```

◄ Class Maze :

La classe Maze est utilisée pour générer un labyrinthe solvable et fini de manière dynamique.Cette classe sera principalement utilisée pour manipuler les attributs et les méthodes de la classe Cell. C'est pourquoi on déclare Maze comme amie de la classe Cell.
il n'y a pas d'attributs dans cette classe juste les méthodes .

❖ les Méthodes :

- **GenerateMaze() :**Cette fonction génère un labyrinthe

solvable en utilisant un algorithme basé sur la méthode de recherche en profondeur (Depth-First Search, DFS). Elle explore dynamiquement les cellules du labyrinthe, marquant chaque cellule visitée et supprimant les murs pour créer un chemin continu. Elle utilise une pile (stack) pour gérer l'exploration des cellules, et choisit aléatoirement les cellules voisines non visitées comme prochaines étapes. Ce processus continue jusqu'à ce que toutes les cellules aient été visitées et un labyrinthe complet et solvable ait été généré.

- On initialise la cellule de départ comme étant la première cellule de la grille (grid[0]), on la marque comme visitée (visited = true) et on l'ajoute à la pile pour commencer l'exploration.

```
Cell* current = &grid[0];

current->visited = true;

stack.push(current);
```

- Tant que la pile n'est pas vide, on continue d'explorer le labyrinthe.
- On récupère la cellule au sommet de la pile (current).

```
Cell* top = current->GetCell(current->x, current->y - 1);
            Cell* right = current->GetCell(current->x + 1, current->y);
            Cell* bottom = current->GetCell(current->x, current->y +1);
        Cell* left = current->GetCell(current->x - 1, current->y);
```

Ces quatre lignes servent à obtenir les cellules voisines de la cellule courante dans les directions haut, droite, bas et gauche, respectivement.

```
 if (top && !top->visited) neighbors.push_back(top);
            if (right && !right->visited) neighbors.push_back(right);
            if (bottom && !bottom->visited)
neighbors.push_back(bottom);
            if (left && !left->visited) neighbors.push_back(left);
```

et ici, on stocke dans le vecteur neighbors uniquement les cellules qui existent et qui n'ont pas encore été visitées.
- Alors que cette partie du code gère le choix et l'exploration d'une cellule voisine parmi celles qui n'ont pas encore été visitées.
  premièrement on vérifie si le vecteur neighbors contient des cellules voisines non visitées. Si neighbors n'est pas vide, cela signifie qu'il y a au moins une cellule voisine à explorer.

```
if (!neighbors.empty()) {
                Cell* next = neighbors[rand() % neighbors.size()];
                next->visited = true;
                current->RemoveWalls(next);
                stack.push(next);
            } else {
                stack.pop();
            }
```

Si il y a des voisins, on en choisit un aléatoirement en utilisant la fonction **rand() % neighbors.size()** Cela génère un index aléatoire dans la plage des indices valides du vecteur neighbors, ce qui permet de créer un nouveau labyrinthe à chaque fois que le jeu est réinitialisé. La cellule sélectionnée est stockée dans le pointeur next,la marque comme visitée pour empêcher cette cellule d'être revisitée ou ajoutée à nouveau au vecteur neighbors dans le futur, ce qui évite les boucles infinies et assure un labyrinthe bien formé. après `current->RemoveWalls(next);` supprime les murs entre la cellule courante (current) et la cellule voisine choisie (next) où les cellules adjacentes doivent être connectées.
 La cellule voisine choisie (next) est ajoutée à la pile (stack) pour être explorée dans les étapes suivantes.

◄ Class Player :

La classe ***Player*** représente un joueur dans le labyrinthe et gère ses propriétés,ses déplacements et son interaction avec l'environnement.

❖ les attributs :

- les coordonnées du joueur (x, y)

```
private:
    int x, y;
```

❖ les Méthodes :

- **Draw(),Draw2() :**

Les méthodes Draw et Draw2 servent à afficher les joueurs sur l'écran dans le labyrinthe, en utilisant des textures spécifiques pour leur représentation visuelle , voici les détailles des deux fc:

-Vector2 est une structure ou un type de données utilisé pour représenter des **vecteurs en deux dimensions** ou des **positions dans un espace 2D**. Il est couramment utilisé dans des bibliothèques graphiques comme Raylib .

```
Vector2 position = {(float)(x * cellSize + 1), (float)(y * cellSize + 1)};
```

on retourne la position du joueur et la calculée en fonction de ses coordonnées (x, y) dans la grille.

-DrawTextureEx: est une fonction de Raylib utilisée pour dessiner une **texture** (image) sur l'écran avec des options avancées, telles que la position, la rotation, l'échelle, et une couleur.

```
DrawTextureEx(player2Texture, position, 0.0f, (float)playerHeight / playerTexture.height, WHITE);
```

- **Move() :**

cette méthode permet de modifier les coordonnées du joueur (x, y) en fonction des directions données par les valeurs dx et dy pour effectuer le déplacement.

```
void Move(int dx, int dy) {
        x += dx;
        y += dy; }
```

- **GameStart () :**

gère les interactions du joueur avec le labyrinthe :

Elle vérifie les entrées clavier pour déplacer le joueur dans une direction donnée (haut, bas, gauche, droite). et s'assure que le déplacement respecte les murs du labyrinthe. et elle détecte la

condition de victoire lorsque le joueur atteint la position de l'objectif (goal).

```
void GameStart (Player& goal, Player& player){

        if (IsKeyPressed(KEY_RIGHT) && !grid[player.y * cols +
player.x].walls[1]) player.Move(1, 0);

           if (IsKeyPressed(KEY_LEFT) && !grid[player.y * cols +
player.x].walls[3]) player.Move(-1, 0);

           if (IsKeyPressed(KEY_DOWN) && !grid[player.y * cols +
player.x].walls[2]) player.Move(0, 1);

           if (IsKeyPressed(KEY_UP) && !grid[player.y * cols +
player.x].walls[0]) player.Move(0, -1);

           if (player.x == goal.x && player.y == goal.y) {

               endTime = GetTime();

               //SaveScore(endTime - startTime);

               gameState = VICTORY;        } }
```

=> Cette classe est essentielle pour permettre une interaction dynamique entre le joueur et le labyrinthe, assurant ainsi une expérience de jeu engageante.

Les fonctions utilisée :

Dans le but de gérer l'affichage des éléments du jeu, les interactions avec le joueur, et l'enregistrement des scores pour offrir une expérience complète et interactive, les fonctions suivantes ont été utilisées :

◄ DrawStartScreen() :

Cette fonction est responsable de dessiner l'écran d'accueil du jeu, en affichant un bouton "START" permettant de démarrer la partie. Voici le détail des étapes :

◆ **ClearBackground(BLACK):** Efface l'écran en le remplissant de noir pour préparer un nouvel affichage propre.

◆ **DrawTexture(startScreenImage, 0, 0, WHITE);** : Affiche une image de fond sur tout l'écran, à partir de la position (0, 0).La couleur WHITE conserve la texture originale sans aucune modification

◆ **DrawText("Welcome to the Maze Game!", screenWidth / 2 - 200, screenHeight / 2 - 60, 30, WHITE);** : Dessine un texte d'accueil centré sur l'écran pour souhaiter la bienvenue au joueur. Le texte est de taille 30 pixels et de couleur blanche.
(screenWidth / 2 - 200) et (screenHeight / 2 - 60) calculer la position pour centrer le texte.

◆ **DrawRectangleRec(startButton, GRAY):** Dessine un rectangle représentant un bouton, avec une couleur grise (GRAY).

◆ **DrawText("START", startButton.x + 37, startButton.y + 10, 26, WHITE); :** Ajoute le texte "START" centré à l'intérieur du bouton, avec une taille de 26 pixels et une couleur blanche.
Les décalages startButton.x + 37 et startButton.y + 10 positionnent le texte correctement dans le rectangle.
=> Ces cinq fonctions utilisées proviennent de la bibliothèque **Raylib**.

```
Texture2D startScreenImage ;
void DrawStartScreen(Rectangle startButton) {
    ClearBackground(BLACK);
    DrawTexture(startScreenImage, 0, 0, WHITE);
    DrawText("Welcome to the Maze Game!", screenWidth / 2 - 200,
screenHeight / 2 - 60, 30, WHITE);
    DrawRectangleRec(startButton, GRAY);
```

```
    DrawText("START", startButton.x + 37, startButton.y + 10, 26,
WHITE);
}
```

◄ SelectDifficultyLevel :

La fonction SelectDifficultyLevel permet à l'utilisateur de choisir le niveau de difficulté (Facile, Moyenne, Difficile) dans un jeu, via une interface graphique. La fonction reste en boucle jusqu'à ce que l'utilisateur appuie sur une touche pour choisir un niveau de difficulté, puis retourne la valeur correspondante.

```
int SelectDifficultyLevel() {
```

➢ la boucle s'exécute tant que la fenêtre du jeu n'est pas fermée. La fonction **WindowShouldClose()** renvoie **true** si l'utilisateur ferme la fenêtre ou si une autre action force la fermeture de la fenêtre (comme un événement système).

```
 while (!WindowShouldClose()){
```

➢ **BeginDrawing() :**Commence à dessiner dans la fenêtre du jeu. Tout ce qui est dessiné après cet appel sera visible à l'écran.

```
BeginDrawing();
```

➢ **ClearBackground(BLACK):** Efface l'écran avec un fond noir. Cela permet de "réinitialiser" la fenêtre avant de dessiner les nouveaux éléments.

```
 ClearBackground(BLACK);
```

➢ **DrawTexture(startScreenImage, 0, 0, WHITE):** Affiche une image de l'écran de démarrage (stockée dans la variable startScreenImage) à la position (0, 0) avec la couleur de tint WHITE (c'est-à-dire sans modification de couleur).

```
 DrawTexture(startScreenImage, 0, 0, WHITE);
```

➢ **DrawText():** Cette fonction est utilisée pour afficher du texte à l'écran (Easy,Medium,Hard).

```
DrawText("   Select Difficulty Level:", 150, 230, 40, WHITE);
      DrawText("       1 - Easy", 210, 300, 27, YELLOW);
      DrawText("       2 - Medium",210, 340, 27, ORANGE);
      DrawText("       3 - Hard", 210, 385, 29, RED)
```

➢ **IsKeyPressed():** Les trois blocs (`KEY_ONE`, `KEY_two`, `KEY_three`) sont des conditions pour vérifier si l'utilisateur appuie sur les touches 1, 2 ou 3 pour sélectionner un niveau de difficulté.

```
if (IsKeyPressed(KEY_ONE)) {
          level = 1;
          break;
      }
      if (IsKeyPressed(KEY_TWO)) {
          level = 2;
          break; }
      if (IsKeyPressed(KEY_THREE)) {
          level = 3;
          break; }
```

➢ **EndDrawing()** : Cette fonction marque la fin du dessin pour cette boucle de rendu. Tout ce qui a été dessiné après **BeginDrawing()** et avant **EndDrawing()** sera visible à l'écran.

```
EndDrawing();
```

RÉSULTAT:

Cette fonction est typiquement utilisée dans les menus de jeux pour permettre au joueur de choisir la difficulté avant de commencer la partie.

◄ StartGame :

La fonction StartGame(int level) initialise un jeu de type labyrinthe avec des paramètres adaptés au niveau de difficulté sélectionné.

```
void StartGame(int level)
```

➢ **gameState = GAME** : Indique que le jeu passe à l'état actif, probablement en changeant un état global.

```
gameState = GAME;
```

➢ **startTime = GetTime()** :Enregistre l'heure de démarrage pour mesurer la durée du jeu.

```
startTime = GetTime();
```

➢ **Maze m** : Crée une instance de Maze, qui semble être une classe ou structure utilisée pour générer et gérer le labyrinthe.

```
Maze m ;
```

➢ Le paramètre level détermine la difficulté et ajuste les paramètres du labyrinthe :

```
switch (level){
```

❖ **Cas 1 : Niveau Facile**

- **Colonnes (cols) et rangées (rows)** : 11x8.
- **Dimensions du joueur** : Hauteur 50px, largeur 52px. Ces valeurs sont grandes pour faciliter la navigation.

❖ **Cas 2 : Niveau Moyen**

- **Colonnes et rangées** : 19x14 (plus dense que le niveau facile).
- **Dimensions du joueur** : 40x40, réduites pour s'adapter à des cellules plus petites.

❖ **Cas 3 : Niveau Difficile**

- **Colonnes et rangées** : 40x30, un labyrinthe beaucoup plus dense.
- **Dimensions du joueur** : 16x17, encore plus petites pour correspondre à une taille de cellule plus petite.

```
case 1:
            cols = 11;
            rows = 8;
            playerHeight = 50;
            playerWidth = 52 ; // Adjust cell size for easy level
            break;
        case 2:
            cols = 19;
            rows = 14;
            playerHeight =40;
            playerWidth = 40 ;  // Adjust cell size for medium level
            break;
        case 3:
            cols = 40;
            rows = 30;
            playerHeight = 16;
            playerWidth = 17 ;  // Adjust cell size for hard level
            break;}
```

➢ **cellSize = screenWidth / cols** : Calcule la taille des cellules du labyrinthe pour s'ajuster dynamiquement à la largeur de l'écran. Cela garantit que le labyrinthe occupe tout l'espace horizontal.

```
cellSize = screenWidth / cols;
```

➢ **srand(time(0))** : Initialise le générateur de nombres aléatoires pour que chaque partie ait un labyrinthe différent.

```
srand(time(0));
```

➢ **grid.clear()** : Efface toute donnée précédemment stockée dans la grille du labyrinthe.

- **Boucles imbriquées pour générer les cellules** :
  - Parcourt toutes les rangées (`rows`) et colonnes (`cols`).
  - Crée une instance de `Cell` pour chaque position `(x, y)` et l'ajoute à la liste `grid`.

```
grid.clear();

   for (int y = 0; y < rows; y++) {

       for (int x = 0; x < cols; x++) {

           grid.push_back(Cell(x, y));

       }

   }
```

➢ **m.GenerateMaze()** : Appelle une méthode pour générer le labyrinthe. Cette méthode appartient probablement à la classe `Maze`. Elle crée la structure logique du labyrinthe (murs, chemins, etc.).

```
 m.GenerateMaze();
```

RÉSULTAT:

C'est une méthode typique pour des jeux de labyrinthe dynamiques, où les niveaux sont générés en fonction de paramètres d'échelle.

◄ DrawTimer :

La fonction `DrawTimer` affiche un chronomètre dans le jeu, indiquant le temps écoulé depuis le début.

- ❖ **Calcul** : Elle calcule le temps écoulé en minutes et secondes.
- ❖ **Affichage** : Elle affiche ce temps sous la forme `Time: MM:SS` en bas à gauche de l'écran.

```
=>void DrawTimer(double startTime, double endTime, bool hasWon)
```

- ➢ **startTime** : Le moment où le jeu a commencé, exprimé en secondes (probablement retourné par la fonction `GetTime()`).
- ➢ **endTime** : Le moment où le jeu s'est terminé, utilisé uniquement si le joueur a gagné.
- ➢ **hasWon** : Indique si le joueur a gagné ou non. Si `true`, on calcule le temps total écoulé entre le début et la fin du jeu. Sinon, on calcule le temps actuel écoulé depuis le début.

```
=>double elapsed = hasWon ? endTime - startTime : GetTime() - startTime;
```

- ➢ Si **hasWon** est vrai, le temps écoulé correspond à **endTime - startTime** (temps final - temps de début).
- ➢ Sinon, le temps écoulé correspond à **GetTime() - startTime**, où `GetTime()` retourne le temps actuel (probablement en secondes depuis le lancement du programme).

```
=> int minutes = static_cast<int>(elapsed) / 60;

int seconds = static_cast<int>(elapsed) % 60;
```

On convertit le temps écoulé (en secondes) en minutes et secondes :

- ➢ Les minutes sont obtenues par une division entière.
- ➢ Les secondes restantes sont obtenues avec le reste de la division par 60.

```
=> char timerText[20];

sprintf(timerText, "Time: %02d:%02d", minutes, seconds);
```

- ➢ Le texte du chronomètre est formaté dans une chaîne de caractères (timerText).
- ➢ Le format "%02d:%02d" garantit que les minutes et les secondes sont toujours affichées avec deux chiffres (ex. : 05:07).

```
=> int timerY = screenHeight - 40;

DrawText(timerText, 10, timerY, 20, RED);
```

- ➢ La position verticale du texte (timerY) est calculée pour l'afficher en bas à gauche, juste au-dessus du bouton "Restart".
- ➢ La fonction **DrawText** de Raylib dessine le texte :
- ➔ **Position** : x = 10 , y = timerY.
- ➔ **Taille** : 20 pixels.
- ➔ **Couleur** : Rouge (RED).

RÉSULTAT:

Le chronomètre est affiché dans le coin inférieur gauche de la fenêtre, indiquant le temps écoulé depuis le début du jeu ou le temps total si le joueur a gagné.

◄ DrawRestarticon() :

Cette fonction permet d'afficher un bouton interactif à l'écran, facilement identifiable par l'utilisateur.

◆ **DrawRectangleRec(restartButton, RED);** : Dessine un rectangle de couleur rouge (RED) en utilisant les dimensions et la position définies par l'objet restartButton.

◆ **DrawText("RESTART", restartButton.x + 23, restartButton.y + 10, 15, WHITE);** :
Affiche le texte "RESTART" centré à l'intérieur du rectangle. Le décalage calculé (restartButton.x + 23 et restartButton.y + 10) garantit que le texte est bien positionné dans le bouton.

=> Lorsque le joueur démarre, ce bouton est placé pour permettre, en cas de besoin, de redémarrer la partie ou de changer le niveau.

```
void  DrawRestarticon(Rectangle restartButton) {
      DrawRectangleRec(restartButton, RED);
      DrawText("RESTART", restartButton.x + 23, restartButton.y + 10,
15, WHITE);}

GenerateMaze();

}
```

◄ DrawVictoryScreen :

La fonction DrawVictoryScreen(Rectangle restartButton, double elapsedTime) est utilisée pour afficher un écran de victoire après que le joueur ait terminé un niveau ou atteint l'objectif. Voici une explication détaillée de chaque étape :

```
void DrawVictoryScreen(Rectangle restartButton, double elapsedTime) {
```

➢ **ClearBackground(BLACK)** : Efface l'écran en le remplissant avec un fond noir. Cela prépare une toile propre pour dessiner les éléments de l'écran de victoire.

```
ClearBackground(BLACK);
```

➢ **DrawTexture(startScreenImage, 0, 0, WHITE)** : Dessine une image à la position (0, 0). La couleur WHITE est utilisée comme multiplicateur, laissant l'image dans ses couleurs originales.

```
 DrawTexture(startScreenImage, 0, 0, WHITE);
```

➢ **DrawText("Congratulations! You reached the goal!", ...)** :

- Affiche un texte au centre de l'écran.
- La position est calculée dynamiquement :
  - **screenWidth / 2 - 180** : Centre horizontal avec un ajustement pour aligner le texte.
  - **screenHeight / 2 - 60** : Centre vertical avec un décalage vers le haut.
- La taille de la police est **20** et la couleur est **YELLOW**, attirant l'attention du joueur.

```
 DrawText("Congratulations! You reached the goal!", screenWidth / 2 - 180, screenHeight / 2 - 60, 20, YELLOW);
```

- **char scoreText[50];** : Crée un tableau de caractères pour stocker le texte du temps final.

```
char scoreText[50];
```

- **sprintf(scoreText, "Final Time: %.2f seconds", elapsedTime);** :
  - Formate la chaîne pour inclure le temps écoulé (elapsedTime) avec deux décimales.
  - Par exemple : *"Final Time: 45.67 seconds"*.
- Dessine le texte au centre, légèrement en dessous du message principal.
- La couleur est **GREEN**, pour indiquer un résultat positif.

```
sprintf(scoreText, "Final Time: %.2f seconds", elapsedTime);
```

```
    DrawText(scoreText, screenWidth / 2 - 100, screenHeight / 2, 20,
GREEN);
```

- **DrawRectangleRec(restartButton, WHITE)** : Dessine un rectangle blanc à la position et aux dimensions définies par restartButton.

```
DrawRectangleRec(restartButton, WHITE);
```

- **DrawText("RESTART", ...)** :
  - Affiche le mot "RESTART" au centre du bouton.
  - Les calculs de position (**restartButton.x + 18** et **restartButton.y + 10**) garantissent que le texte est bien aligné à l'intérieur du rectangle.
  - Taille de la police : **25**.
  - Couleur du texte : **BLACK**, pour créer un contraste avec le fond blanc du bouton.

```
DrawText("RESTART", restartButton.x + 18, restartButton.y + 10, 25,
BLACK);
```

RÉSULTAT:

Ce type d'écran est typique dans les jeux pour informer le joueur de sa réussite et lui offrir des options de continuer ou redémarrer.

La fonction Principale:

◄ main :

Le code de la fonction main sert de point d'entrée pour un jeu de labyrinthe avec un écran de victoire. Voici une explication détaillée de ses composants et de son fonctionnement :

➢ InitWindow(screenWidth, screenHeight, "Maze Game with Victory Screen") :

Cette fonction crée une fenêtre avec une largeur et hauteur spécifiées et un titre ("**JEU DE CASSE-TÊTE DE STYLE LABYRINTHE**").

```
int main() {

    InitWindow(screenWidth  , screenHeight  , "jeu de casse-tête de style labyrinthe");
```

➢ SetTargetFPS(60) :

Définit la fréquence d'images cible à 60 FPS pour une animation fluide.

```
SetTargetFPS(60);
```

➢ playerTexture, player2Texture, startScreenImage :

Charge les images nécessaires pour le joueur, le but et l'écran de démarrage.

```
playerTexture = LoadTexture("player/img2.png");

player2Texture = LoadTexture("player/img3.png");

startScreenImage = LoadTexture("player/img1.png");
```

➢ startButton :

Position et taille du bouton de démarrage dans le menu principal.

```
Rectangle startButton = {screenWidth / 2 - 75, screenHeight / 2 - 25, 150, 50};
```

➢ **restartButton** :

Bouton pour redémarrer depuis l'écran de victoire.

```
Rectangle restartButton = {screenWidth / 2 - 75, screenHeight / 2 + 50, 150, 50};
```

➢ **restartButton2** :

Bouton pour retourner au menu depuis l'écran de jeu.

```
Rectangle restartButton2 = { 600 ,585, 110, 30};
```

➢ **Player player(0, 0)** :

Le joueur commence à la position (0, 0) dans le labyrinthe.

```
Player player(0, 0);
```

➢ **Player goal(cols - 1, rows - 1)** :

L'objectif est placé en bas à droite du labyrinthe.

```
Player goal(cols - 1, rows - 1);
```

➢ Le jeu fonctionne dans une boucle principale

while(!WindowShouldClose()), qui continue jusqu'à ce que l'utilisateur ferme la fenêtre.

```
while (!WindowShouldClose())
```

La logique du jeu est divisée en plusieurs états :

❖ **MENU** :

- ➢ L'écran de démarrage s'affiche.
- ➢ **DrawStartScreen(startButton)** :
  - ■ Dessine le menu principal avec le bouton de démarrage.
- ➢ Si l'utilisateur clique sur le bouton de démarrage :
  - ■ La difficulté est sélectionnée via **SelectDifficultyLevel()**.
  - ■ La partie est initiée avec **StartGame(difficultyLevel)**.

- Les positions du joueur et de l'objectif sont réinitialisées.

```
 if (gameState == MENU) {

            BeginDrawing();

            DrawStartScreen(startButton);

          if (CheckCollisionPointRec(GetMousePosition(), startButton)
&& IsMouseButtonPressed(MOUSE_LEFT_BUTTON)) {

                difficultyLevel = SelectDifficultyLevel();

                StartGame(difficultyLevel);

                player = Player( 0 ,  0);

                goal = Player (cols - 1, rows -1 );

            }

            EndDrawing();

        } else if (gameState == GAME) {

            BeginDrawing();

            ClearBackground(WHITE);
```

❖ **GAME :**

➢ Le jeu est actif, et les éléments du labyrinthe sont affichés.

- Les cellules du labyrinthe sont dessinées via **`for (auto& cell : grid) cell.Draw()`**.
- Le joueur et l'objectif sont affichés avec **`player.Draw()`** et **`goal.Draw2()`**.
- **`player.GameStart(goal, player)`** : Vérifie les interactions entre le joueur et l'objectif, probablement pour détecter une victoire.
- Si l'utilisateur clique sur le bouton **`restartButton2`**, l'état passe à `MENU`.

- Le temps écoulé est affiché via **DrawTimer(startTime, endTime, gameState == VICTORY)**.

```
        for (auto& cell : grid) cell.Draw();

            player.Draw();

            goal.Draw2() ;

            player.GameStart(goal, player);

          DrawStarticon(restartButton2);

          if (CheckCollisionPointRec(GetMousePosition(),
restartButton2) && IsMouseButtonPressed(MOUSE_LEFT_BUTTON)) {

              gameState = MENU;}

          DrawTimer(startTime, endTime, gameState == VICTORY);

          EndDrawing();

       }
```

❖ **VICTORY :**

➢ L'écran de victoire s'affiche via **DrawVictoryScreen(restartButton, endTime - startTime)**.

➢ Si l'utilisateur clique sur le bouton **restartButton**, l'état passe à MENU.

Avant de quitter le programme :

➢ **UnloadTexture(playerTexture)** : Libère la mémoire utilisée par les textures.

➢ **CloseWindow()** : Fermer la fenêtre.

```
else if (gameState == VICTORY) {

          BeginDrawing();

          DrawVictoryScreen(restartButton, endTime - startTime);
```

```
          if (CheckCollisionPointRec(GetMousePosition(), restartButton)
&& IsMouseButtonPressed(MOUSE_LEFT_BUTTON)) {

                gameState = MENU;}

    EndDrawing();}}

    UnloadTexture(playerTexture);

    CloseWindow();
```

Conclusion :

En résumé, ce projet de jeu de labyrinthe, développé avec la bibliothèque Raylib en C++, illustre une excellente maîtrise des concepts fondamentaux de la programmation orientée objet et de la conception de jeux vidéo. Grâce à une structure bien organisée et une expérience utilisateur engageante, le jeu propose une immersion unique à travers des labyrinthes générés procéduralement et adaptés à différents niveaux de difficulté.

L'intégration de fonctionnalités telles que la gestion des états (MENU, GAME, VICTORY), un chronomètre pour mesurer les performances du joueur, et des interfaces intuitives enrichissent l'expérience globale.