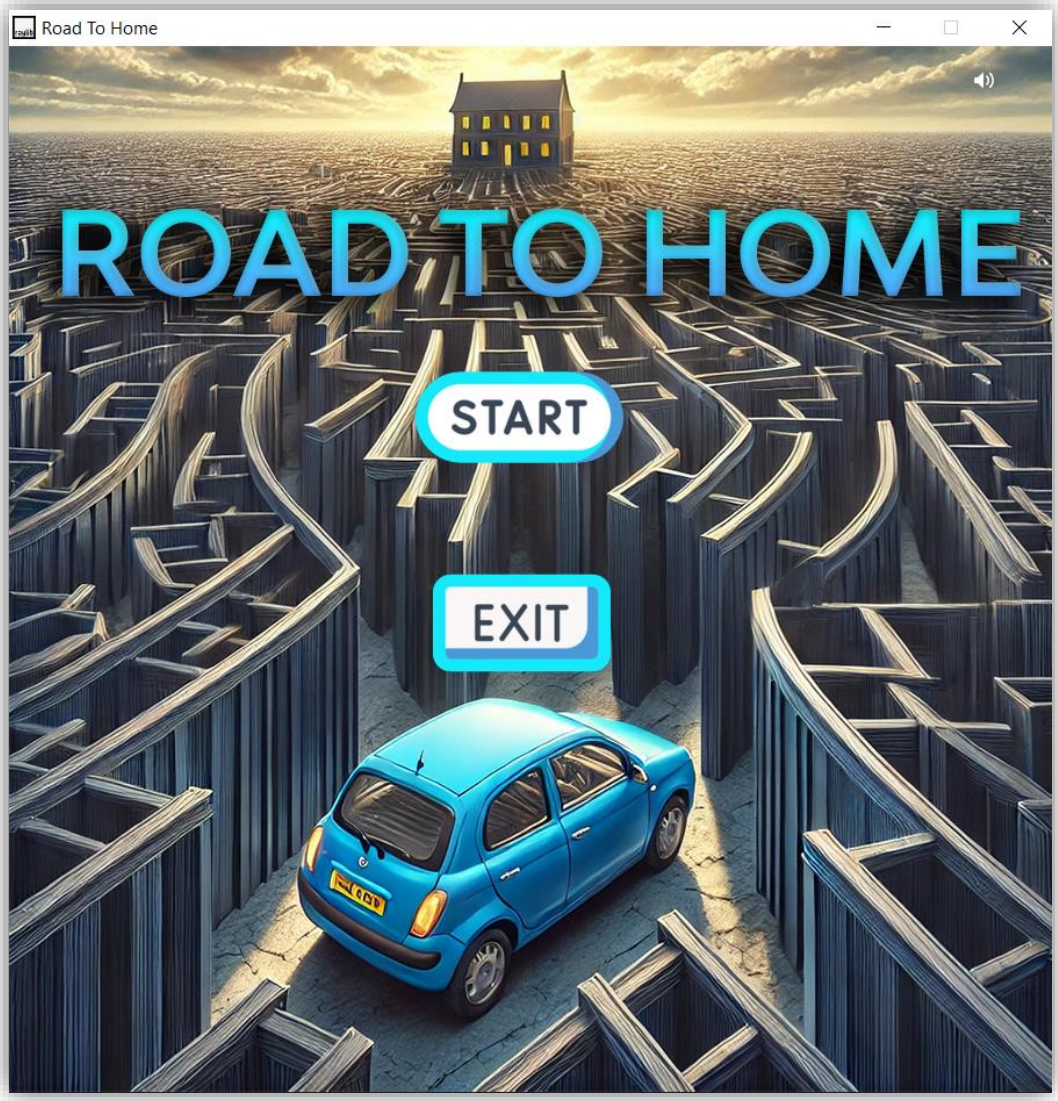
Faculté des Sciences et Techniques de Tanger

Rapport sur le projet

ROAD TO HOME GAME

C++/Raylib



Ce rapport présente le développement d'un jeu de labyrinthe en C++ utilisant la bibliothèque Raylib. Le jeu permet au joueur de naviguer dans un labyrinthe généré de manière procédurale et de se rendre à la sortie. Le joueur est chronométré et le meilleur temps est enregistré

I_ Classe labyrinthe ; Génération procédurale du labyrinthe :

La classe Labyrinthe est responsable de la génération procédurale d'un labyrinthe en utilisant un algorithme de recherche en profondeur avec backtracking. Décomposons ses composants principaux :

```
class Labyrinthe {
public:
    int largeur, hauteur; // Dimensions du labyrinthe
    vector<vector<int>> grille; // Grille du labyrinthe
```

Le labyrinthe est représenté par une grille 2D où :

- 1 représente les murs
- 0 représente les passages (chemins que le joueur peut emprunter)

Methodes principales :

1_Constructeur :

```
Labyrinthe::Labyrinthe(int l, int h) : largeur(l), hauteur(h) {
    srand(time(0)); // Initialisation du générateur aléatoire
    grille.resize(hauteur, std::vector<int>(largeur, 1)); // Initialisation : tout
est mur
    genererLabyrinthe(); // Génération du labyrinthe
}
```

Le constructeur initialise la grille du labyrinthe avec toutes les cellules comme des murs et définit une graine aléatoire pour garantir la génération unique de chaque labyrinthe.

2_Algorithme de génération de labyrinthe :

```
void Labyrinthe::genererLabyrinthe() {
    // Positions initiales
    int x = 1, y = 1; // Position de départ
```

```
    // Marquer la cellule de départ comme un passage
    grille[y][x] = 0; // Départ

    // Pile pour le backtracking (retour arrière)
    stack<pair<int, int>> pile; // Pile pour les positions
    pile.push({x, y}); // Ajouter la position de départ

    // Déplacements possibles (droite, bas, gauche, haut)
    vector<pair<int, int>> directions = {{2, 0}, {0, 2}, {-2, 0}, {0, -2}}; //
Directions possibles
```

L'algorithme fonctionne comme suit :

Initialisation :

Il commence toujours au point de coordonnées (1,1), marque cette case comn un passage, et utilise une pile pour explorer systématiquement les différentes configurations possibles.

Point important sur les directions :

Les mouvements sont effectués par pas de deux cases. Cette méthode garantit toujours un mur entre deux chemins, créant une structure de labyrinthe régulière et structurée.

```
while (!pile.empty()) { // Tant que la pile n'est pas vide
        pair<int, int> current = pile.top(); // Position actuelle
        int cx = current.first; // Coordonnée x actuelle
        int cy = current.second; // Coordonnée y actuelle
        vector<pair<int, int>> voisins; // Liste des voisins
```

Condition de la boucle While (!pile.emty()) :

La boucle continue tant que la pile n'est pas vide. Elle traite à chaque itération une position potentielle, explorant systématiquement toutes les configurations possibles de chemins.

Récupération de la position courante :

Utilisation d'un pair<int, int> pour stocker les coordonnées x et y. La méthode pile.top() permet de consulter l'élément au sommet de la pile sans le supprimer.

Préparation de la liste des voisins :

Préparation d'un vecteur pour stocker les positions adjacentes potentiellement accessibles depuis la position courante dans le labyrinthe.

Criteres de validite dun voisin :

Vérification rigoureuse des cases : rester dans les limites du labyrinthe, ne pas être déjà un passage, et maintenir une distance de deux cases.

```
// Trouver les voisins valides
        for (size_t i = 0; i < directions.size(); i++) { // Parcourir les directions
            int dx = directions[i].first; // Déplacement en x
            int dy = directions[i].second; // Déplacement en y
            int nx = cx + dx; // Nouvelle coordonnée x
            int ny = cy + dy; // Nouvelle coordonnée y
            if (nx > 0 && ny > 0 && nx < largeur - 1 && ny < hauteur - 1 &&
grille[ny][nx] == 1) { // Vérifier validité
                voisins.push_back(make_pair(nx, ny)); // Ajouter le voisin
            }
        }
```

Boucle de parcours des directions :

Parcourt systématiquement tous les voisins potentiels définis dans le tableau des directions. Utilise size_t pour éviter les warnings de conversion d'entiers.

Calcul des déplacements :

Détermine les variations horizontales (dx) et verticales (dy) pour chaque direction. Permet de générer les coordonnées des cases adjacentes avec précision.

Calcul des nouvelles coordonnées :

Calcule les nouvelles positions x et y en ajoutant les déplacements aux coordonnées courantes. Permet de définir les positions potentielles dans le labyrinthe.

```
if (nx > 0 && ny > 0 && nx < largeur - 1 && ny < hauteur - 1 && grille[ny][nx] == 1)
```

Vérification de validité :

- nx > 0 : pas trop à gauche
- ny > 0 : pas trop en haut
- nx < largeur - 1 : pas trop à droite
- ny < hauteur - 1 : pas trop en bas
- grille[ny][nx] == 1 : la case est accessible (valeur 1)

Ajout du voisin valide :

Ajoute les coordonnées du voisin qui a passé tous les tests de validation au vecteur des voisins potentiels.

```
 if (!voisins.empty()) { // S'il y a des voisins
            // Choisir un voisin aléatoire
            pair<int, int> next = voisins[rand() % voisins.size()]; // Voisin
aléatoire
            int nx = next.first; // Coordonnée x du voisin
            int ny = next.second; // Coordonnée y du voisin

            // Casser le mur entre la cellule actuelle et le voisin
            grille[cy + (ny - cy) / 2][cx + (nx - cx) / 2] = 0; // Casser le mur

            // Marquer le voisin comme un passage
            grille[ny][nx] = 0; // Marquer passage
            // Ajouter le voisin à la pile
            pile.push(make_pair(nx, ny)); // Ajouter à la pile
        } else {
            // Retour arrière
            pile.pop(); // Retirer de la pile
        }
    }
```

```
    // Départ et sortie
    grille[1][1] = 0; // Départ
    grille[hauteur - 2][largeur - 2] = 0; // Sortie
}
```

Vérification de voisins disponibles :

Teste s'il existe des voisins non explorés. Permet de décider entre continuer l'exploration ou effectuer un backtracking.

```
    pair<int, int> next = voisins[rand() % voisins.size()]; // Voisin aléatoire
```

Sélection aléatoire d'un voisin :

Utilise rand() % voisins.size() pour choisir un index aléatoire. Garantit un choix non déterministe parmi les voisins valides disponibles.

Extraction des coordonnées du voisin :

Sépare les coordonnées x et y du voisin sélectionné pour faciliter leur manipulation et leur utilisation dans l'algorithme.

Casser le mur entre cellules :

Calcule la position du mur entre la cellule courante et le voisin en utilisant une moyenne. Transforme cette case en passage (valeur 0).

Ajout du voisin à la pile :

Ajoute les coordonnées du voisin au sommet de la pile. Prépare l'exploration des voisins de cette nouvelle cellule dans le labyrinthe.

Gestion du backtracking :

Lorsque aucun voisin n'est disponible, retire l'élément courant de la pile. Permet de revenir à la dernière cellule ayant des voisins non explorés.

3_Rendu Visuel du Labyrinthe :

```
void Labyrinthe::dessiner(int tailleCase, int offsetX, int offsetY) {
    for (int i = 0; i < hauteur; i++) { // Parcourir les lignes
        for (int j = 0; j < largeur; j++) { // Parcourir les colonnes
            if (grille[i][j] == 1) {
                DrawRectangle(j * tailleCase + offsetX, i * tailleCase + offsetY,
tailleCase, tailleCase, BLACK); // Dessiner un mur
            } else {
                DrawRectangle(j * tailleCase + offsetX, i * tailleCase + offsetY,
tailleCase, tailleCase, LIGHTGRAY); // Dessiner un passage
            }
        }
    }
}
```

Structure de parcours :

Utilise une double boucle imbriquée pour parcourir l'intégralité de la grille, avec i représentant les lignes et j représentant les colonnes.

Logique de dessin :

Distingue les murs (dessinés en noir) des passages (dessinés en gris clair) en fonction de la valeur dans la grille. (grille[i][j] == 1)

Calcul des positions :

Détermine précisément la position x et y de chaque rectangle en multipliant les coordonnées par la taille de la case et en ajoutant un décalage.

Fonction DrawRectangle() :

Méthode de dessin de la bibliothèque Raylib qui positionne et dessine un rectangle en spécifiant sa position, sa taille et sa couleur.

II_ Classe Joueur :

La classe Joueur est responsable pour la gestion de l'apparence, les mouvements et les actions d'un joueur dans un jeu de labyrinthe, probablement développé avec Raylib.

1_Constructeur:

```
Joueur::Joueur(Texture2D& right, Texture2D& left, Texture2D& up, Texture2D& down,
Texture2D& house, int startX, int startY)
    : joueurTextureRight(right), joueurTextureLeft(left), joueurTextureUp(up),
joueurTextureDown(down), joueurX(startX), joueurY(startY), houseTexture(house) {
    joueurTexture = &joueurTextureRight;
}
```

Paramètres :

- right, left, up, down : Textures représentant le joueur dans différentes directions.
- house : Texture de la "maison" ou de l'objectif final.
- startX, startY : Position de départ du joueur dans la grille.

Fonctionnement :

Initialise les textures du joueur dans différentes directions et sa position de départ. Par défaut, configure la texture de direction droite comme texture active.

2_Méthode dessiner :

```
void Joueur::dessiner(int tailleCase, int offsetX, int offsetY, int largeur, int
hauteur) {
    int joueurOffsetX = (tailleCase - joueurTexture->width) / 2; // Calculer les
positions pour centrer la texture du joueur dans la case
    int joueurOffsetY = (tailleCase - joueurTexture->height) / 2;
    // Afficher le joueur avec la texture centrée
    DrawTexture(*joueurTexture, joueurX * tailleCase + offsetX + joueurOffsetX,
joueurY * tailleCase + offsetY + joueurOffsetY, WHITE);
    // Afficher l'image de la maison à la sortie
    DrawTexture(houseTexture, (largeur - 2) * tailleCase + offsetX, (hauteur - 2) *
tailleCase + offsetY, WHITE);
}
```

Objectif : Afficher le joueur et la maison sur la grille.

Paramètres :

- tailleCase : Taille d'une cellule dans la grille.
- offsetX, offsetY : Décalages pour ajuster le dessin sur l'écran.
- largeur, hauteur : Dimensions de la grille.

Fonctionnement :

- La texture du joueur est dessinée au centre de sa case.
- La texture de la maison est dessinée près de la sortie (position (largeur - 2, hauteur - 2)).

3_Méthode deplacer :

```
void Joueur::deplacer(Labyrinthe& labyrinthe) {
    static float deplacementTimer = 0.0f;
    const float deplacementInterval = 0.08f; // Intervalle de temps entre les
déplacements

    deplacementTimer += GetFrameTime();

    if (deplacementTimer >= deplacementInterval) {
        if (IsKeyDown(KEY_RIGHT) && labyrinthe.grille[joueurY][joueurX + 1] == 0) {
```

```
            joueurX++;
            joueurTexture = &joueurTextureRight;
        }
        if (IsKeyDown(KEY_LEFT) && labyrinthe.grille[joueurY][joueurX - 1] == 0) {
            joueurX--;
            joueurTexture = &joueurTextureLeft;
        }
        if (IsKeyDown(KEY_DOWN) && labyrinthe.grille[joueurY + 1][joueurX] == 0) {
            joueurY++;
            joueurTexture = &joueurTextureDown;
        }
        if (IsKeyDown(KEY_UP) && labyrinthe.grille[joueurY - 1][joueurX] == 0) {
            joueurY--;
            joueurTexture = &joueurTextureUp;
        }
        deplacementTimer = 0.0f; // Réinitialiser le timer après un déplacement
    }
}
```

Objectif : Gérer les déplacements du joueur en fonction des touches pressées tout en limitant leur fréquence à une certaine vitesse.

Paramètres :

- labyrinthe : Référence au labyrinthe. La grille du labyrinthe est utilisée pour vérifier si une case cible est libre avant d'effectuer le déplacement.

Fonctionnement :

- Temporisation des déplacements :

- Un timer (deplacementTimer) est utilisé pour limiter la fréquence des déplacements à un intervalle défini (deplacementInterval = 0.08f), ce qui garantit un déplacement fluide mais contrôlé.

- Vérification des touches directions :

Si une touche directionnelle est pressée (KEY_RIGHT, KEY_LEFT, etc.), la méthode vérifie :

- Si la case cible dans la grille du labyrinthe est libre (labyrinthe.grille[...] == 0).

- Si la case est libre :

+ La position du joueur (joueurX ou joueurY) est mise à jour en fonction de la direction de déplacement souhaitée et de la validité de la case adjacente dans la grille du labyrinthe.

+ La texture du joueur est mise à jour pour refléter la direction choisie (droite, gauche, haut, ou bas).

- Une fois un déplacement effectué, le timer est réinitialisé pour préparer le prochain cycle de mouvement.

4_Méthode aAtteintSortie :

```
bool Joueur::aAtteintSortie(int largeur, int hauteur) {
    return joueurX == largeur - 2 && joueurY == hauteur - 2;
}
```

Objectif : Vérifier si le joueur a atteint la case de sortie du labyrinthe.
Retourne : true si le joueur est à la position (largeur - 2, hauteur - 2)

5_Méthode Reinitialiser:

```
void Joueur::reinitialiser(int startX, int startY) {
    joueurX = startX;
    joueurY = startY;
    joueurTexture = &joueurTextureRight;
}
```

Objectif : Réinitialiser le joueur à sa position de départ et restaurer la texture par défaut (droite).

6_Méthode DessinerImage:

```
void Joueur::DessinerImage(const char* imagePath, Texture2D& texture, int newWidth, in
newHeight) {
    Image image = LoadImage(imagePath);
    ImageResize(&image, newWidth, newHeight);
    texture = LoadTextureFromImage(image);
    UnloadImage(image);
}
```

Objectif : Charger une image depuis un chemin, la redimensionner, et la convertir en une texture utilisable par Raylib.

Fonctionnement :

- L'image est chargée avec LoadImage, redimensionnée avec ImageResize, puis convertie en texture avec LoadTextureFromImage.
- Après usage, l'image est déchargée avec UnloadImage pour éviter les fuites de mémoire.

7_Méthode DessinerImage:

```
void Joueur::DessinerVoitureJaune(Texture2D& joueurTextureRight, Texture2D&
joueurTextureLeft, Texture2D& joueurTextureUp, Texture2D& joueurTextureDown) {
    DessinerImage("Graphics/Cars/Yellow_Car/Car_Right.png", joueurTextureRight, 35,
20);
    DessinerImage("Graphics/Cars/Yellow_Car/Car_Left.png", joueurTextureLeft, 35,
20);
    DessinerImage("Graphics/Cars/Yellow_Car/Car_Top.png", joueurTextureUp, 20, 35);
    DessinerImage("Graphics/Cars/Yellow_Car/Car_Bottom.png", joueurTextureDown, 20,
35);
}

void Joueur::DessinerVoitureBleue(Texture2D& joueurTextureRight, Texture2D&
joueurTextureLeft, Texture2D& joueurTextureUp, Texture2D& joueurTextureDown) {
```

```
    DessinerImage("Graphics/Cars/Blue_Car/Car_Right.png", joueurTextureRight, 35,
19);
    DessinerImage("Graphics/Cars/Blue_Car/Car_Left.png", joueurTextureLeft, 35, 19);
    DessinerImage("Graphics/Cars/Blue_Car/Car_Top.png", joueurTextureUp, 19, 35);
    DessinerImage("Graphics/Cars/Blue_Car/Car_Bottom.png", joueurTextureDown, 19,
35);
}

void Joueur::DessinerVoitureRouge(Texture2D& joueurTextureRight, Texture2D&
joueurTextureLeft, Texture2D& joueurTextureUp, Texture2D& joueurTextureDown) {
    DessinerImage("Graphics/Cars/Red_Car/Car_Right.png", joueurTextureRight, 35, 20);
    DessinerImage("Graphics/Cars/Red_Car/Car_Left.png", joueurTextureLeft, 35, 20);
    DessinerImage("Graphics/Cars/Red_Car/Car_Top.png", joueurTextureUp, 20, 35);
    DessinerImage("Graphics/Cars/Red_Car/Car_Bottom.png", joueurTextureDown, 20, 35);
}
```

Description :

- Ces méthodes chargent les textures correspondant aux différentes couleurs de voiture (Jaune, Bleue, Rouge).
- Chaque texture est redimensionnée avant d'être utilisée pour s'assurer qu'elle correspond aux dimensions du joueur.

III_Classe button :

```
class button {
private:
    Texture2D texture;
    Vector2 position;
    float scale;
    bool isVolumeButton;
    bool isMuted;
    Music* backgroundMusic;  // Changé en pointeur

public:
    button(const char* imagepath, Vector2 imageposition, float scale, bool isVolume = false);
    ~button();

    void Draw();
    void DrawWithScale(float newScale);
    bool isPressed(Vector2 mousePosition, bool mousePressed);
    bool isHovered(Vector2 mousePosition) const;
    void updateVolume();

    Rectangle getBounds() const;
};

#endif // BUTTON_HPP
```

La classe button est une classe crée pour représenter un bouton interactif dans une interface graphique, géré par la bibliothèque raylib. Elle permet de gérer l'affichage d'un bouton à l'écran, de vérifier si l'utilisateur a cliqué dessus ou si la souris le survole, et de gérer l'interaction de ce bouton avec l'utilisateur, comme le contrôle du volume ou l'affichage de l'état du bouton.

Méthodes principales :

1_constructeur button() :

```
button::button(const char* imagepath, Vector2 imageposition, float buttonScale, bool isVolume)
    : position(imageposition), scale(buttonScale), isVolumeButton(isVolume), isMuted(false), backgroundMusic(nullptr
    {
    Image image = LoadImage(imagepath);
    int newWidth = static_cast<int>(image.width * scale);
    int newHeight = static_cast<int>(image.height * scale);

    ImageResize(&image, newWidth, newHeight);
    texture = LoadTextureFromImage(image);
    UnloadImage(image);

    if (isVolumeButton) {
        InitAudioDevice();
        backgroundMusic = new Music;
        *backgroundMusic = LoadMusicStream("Graphics/Musics/Background_music.mp3");
        PlayMusicStream(*backgroundMusic);
        SetMusicVolume(*backgroundMusic, 1.0f);
    }
}
```

Le constructeur initialise les variables de la classe premièrement on a LoadImage(imagepath) qui Charge l'image depuis un fichier et ImageResize() qui Redimensionne l'image en fonction de l'échelle donnée (buttonScale) puis LoadTextureFromImage() qui permet de Convertir l'image en texture qui pourra être affichée à l'écran ensuite on a la Gestion de la musique si le bouton est un bouton de volume (isVolumeButton == true), la musique est chargée et lue à l'aide des fonctions raylib pour gérer l'audio (InitAudioDevice(), LoadMusicStream(), PlayMusicStream()).

2_ Destructeur ~button() :

```
button::~button() {
    UnloadTexture(texture);
    if (isVolumeButton && backgroundMusic != nullptr) {
        StopMusicStream(*backgroundMusic);
        UnloadMusicStream(*backgroundMusic);
        delete backgroundMusic;
        CloseAudioDevice();
    }
}
```

Le destructeur est responsable de la libération des ressources allouées par la classe dont il y a la méthode UnloadTexture() qui Libère la texture du bouton.

- Si le bouton est un bouton de volume et qu'il existe une musique en cours de lecture (backgroundMusic), on arrête la musique, la décharge et ferme le périphérique audio pour éviter toute fuite de mémoire.

3_méthode draw() :

```
void button::Draw() {
    DrawTextureV(texture, position, WHITE);
}
```

La méthode Draw permet d'afficher le bouton à l'écran. Elle utilise la fonction DrawTextureV de raylib pour afficher la texture à la position donnée, avec la couleur blanche (WHITE), ce qui signifie qu'il n'y a pas de filtre de couleur appliqué.

4_Méthode DrawWithScale() :

```
void button::DrawWithScale(float scale) {
    float centerX = position.x + texture.width / 2;
    float centerY = position.y + texture.height / 2;

    float newX = centerX - (texture.width * scale) / 2;
    float newY = centerY - (texture.height * scale) / 2;

    Rectangle source = {0, 0, (float)texture.width, (float)texture.height};
    Rectangle dest = {newX, newY, texture.width * scale, texture.height * scale};
    DrawTexturePro(texture, source, dest, {0, 0}, 0, WHITE);
}
```

Cette méthode permet de dessiner le bouton avec une taille redimensionnée, en ajustant l'échelle du bouton. Elle effectue plusieurs étapes premièrement le Calcule le centre du bouton en fonction de sa position deuxièmement la modification de la position de dessin pour que le bouton soit centré après avoi changé sa taille et finalement l'utilisation la fonction DrawTexturePro pour dessiner la texture avec les nouvelles dimensions calculées.

4_Méthode isPressed() :

```
bool button::isPressed(Vector2 mousePosition, bool mousePressed) {
    Rectangle rect = {position.x, position.y, static_cast<float>(texture.width), static_cast<float>(texture.height)};
    return CheckCollisionPointRec(mousePosition, rect) && mousePressed;
}
```

Cette méthode permet de vérifier si le bouton a été pressé par l'utilisateur.

- Elle crée un rectangle autour du bouton (rect) basé sur sa position et ses dimensions.
- CheckCollisionPointRec() vérifie si la position de la souris (mousePosition) se trouve à l'intérieur de ce rectangle.
- Enfin, elle vérifie si le bouton gauche de la souris a été pressé (mousePressed)

5_Méthode getBounds() :

```
Rectangle button::getBounds() const {
    return Rectangle{position.x, position.y, static_cast<float>(texture.width), static_cast<float>(texture.height)};
}
```

La méthode getBounds renvoie un rectangle qui définit les limites du bouton à l'écran.

6_Méthode isHovered() :

```
bool button::isHovered(Vector2 mousePosition) const {
    Rectangle rect = {position.x, position.y, static_cast<float>(texture.width), static_cast<float>(texture.height)};
    return CheckCollisionPointRec(mousePosition, rect);
}
```

Cette méthode permet de vérifier si la souris survole le bouton.

7_Méthode updateVolume() :

```
void button::updateVolume() {
    if (!isVolumeButton || backgroundMusic == nullptr) return;

    UpdateMusicStream(*backgroundMusic);
    Vector2 mousePos = GetMousePosition();

    if (isPressed(mousePos, IsMouseButtonPressed(MOUSE_BUTTON_LEFT))) {
        isMuted = !isMuted;
        SetMusicVolume(*backgroundMusic, isMuted ? 0.0f : 1.0f);

        UnloadTexture(texture);
        Image newImage = LoadImage(isMuted ? "Graphics/Volumes/Mute.png" : "Graphics/Volumes/volume-up.png");
        ImageResize(&newImage, static_cast<int>(newImage.width * scale), static_cast<int>(newImage.height * scale));
        texture = LoadTextureFromImage(newImage);
        UnloadImage(newImage);
    }
}
```

Cette méthode gère l'interaction avec le bouton de volume :

-Si le bouton est un bouton de volume et qu'une musique est en cours, elle met à jour la musique avec UpdateMusicStream().

-Elle vérifie si le bouton a été pressé avec isPressed().

-Elle bascule l'état muet (isMuted), puis met à jour le volume de la musique.

-Enfin, elle change l'image du bouton pour afficher l'état actuel du volume

IV_Classe niveau :

```
class Niveau {
public:
    Niveau();
    ~Niveau();

    void afficher();
    void afficherStartMaze();
    int verifierSelection(Vector2 mousePos, bool mousePres);
    int verifierStartMaze(Vector2 mousePos, bool mousePres);

private:
    Texture2D background;
    Texture2D carColorsBackground; // Nouveau background pour la sélection de couleur de voiture
    button* facileBtn;
    button* moyenBtn;
    button* difficileBtn;
    button* startMazeBtnY; // Bouton pour débuter le labyrinthe avec voiture jaune
    button* startMazeBtnB; // Bouton pour débuter le labyrinthe avec voiture bleue
    button* startMazeBtnR; // Bouton pour débuter le labyrinthe avec voiture rouge
};

#endif // NIVEAU_HPP
```

La classe Niveau représente un niveau de jeu dans une interface graphique utilisant la bibliothèque raylib. Elle gère l'affichage des différents boutons

permettant à l'utilisateur de sélectionner un niveau de difficulté (facile, moyer difficile), ainsi que le choix de la couleur de la voiture pour démarrer un labyrinthe.
Elle est responsable de l'affichage des écrans de sélection du niveau, de la gestion des interactions avec l'utilisateur (via des clics de souris) et de la gestion des ressources graphiques (comme les textures de fond et les boutons).

Les méthode principale :

1_Constructeur Niveau() :

```
Niveau::Niveau() {
    background = LoadTexture("Graphics/Backgrounds/HOME_Difficulty.png");
    carColorsBackground = LoadTexture("Graphics/Backgrounds/HOME_Car_Colors.png");
    facileBtn = new button("Graphics/Buttons/Facile.png", {358, 220}, 1.2);
    moyenBtn = new button("Graphics/Buttons/Moyenne.png", {350, 370}, 1.3);
    difficileBtn = new button("Graphics/Buttons/Difficile.png", {350, 520}, 1.3);
    startMazeBtnY = new button("Graphics/Buttons/Y_Car_But.png", {350, 200}, 1.2);
    startMazeBtnB = new button("Graphics/Buttons/B_Car_But.png", {350, 370}, 1.2);
    startMazeBtnR = new button("Graphics/Buttons/R_Car_But.png", {350, 540}, 1.2);
}
```

cette méthode permet du Chargement des textures , Le constructeur commen par charger les textures de fond (deux fonds différents, un pour la sélection de difficulté et un autre pour la sélection de la couleur de voiture) via la fonction LoadTexture().Ensuite il crée des objets button pour chaque bouton de sélectic de niveau (facile, moyen, difficile) et pour chaque bouton de sélection de couleur de voiture (jaune, bleu, rouge). Chaque bouton est initialisé avec une image, une position (Vector2), et un facteur d'échelle pour ajuster sa taille.

2_Destructeur ~Niveau() :

```
Niveau::~Niveau() {
    UnloadTexture(background);
    UnloadTexture(carColorsBackground); // Libérer la mémoire du nouveau background
    delete facileBtn;
    delete moyenBtn;
    delete difficileBtn;
    delete startMazeBtnY; // Libérer la mémoire du bouton jaune
    delete startMazeBtnB; // Libérer la mémoire du bouton bleu
    delete startMazeBtnR; // Libérer la mémoire du bouton rouge
}
```

Le destructeur est utilisé pour libérer les ressources allouées dynamiquement p
la classe :
Les textures de fond sont déchargées avec UnloadTexture() et la Suppression des boutons déjà crée.

3_Méthode afficher() :

```
void Niveau::afficher() {
    DrawTexture(background, 0, 0, WHITE);
    Vector2 mousePos = GetMousePosition();

    // Appliquer l'effet hover pour chaque bouton

    if (facileBtn->isHovered(mousePos)) {
        facileBtn->DrawWithScale(1.1f);
    } else {
        facileBtn->Draw();
    }

    if (moyenBtn->isHovered(mousePos)) {
        moyenBtn->DrawWithScale(1.1f);
    } else {
        moyenBtn->Draw();
    }

    if (difficileBtn->isHovered(mousePos)) {
        difficileBtn->DrawWithScale(1.1f);
    } else {
        difficileBtn->Draw();
    }
}
```

Cette méthode est responsable de dessiner l'écran de sélection du niveau. Elle

- Affiche l'image de fond à l'aide de DrawTexture().
- Récupère la position de la souris avec GetMousePosition().
- Pour chaque bouton de niveau (facile, moyen, difficile), elle vérifie si la souris est au-dessus du bouton (en utilisant isHovered()). Si oui, le bouton est agrandi avec DrawWithScale() pour donner un effet visuel de survol, sinon le bouton est affiché à sa taille normale avec Draw().

4_Méthode afficherStartMaze() :

```
void Niveau::afficherStartMaze() {
    DrawTexture(carColorsBackground, 0, 0, WHITE); // Utiliser le nouveau backgro
    Vector2 mousePos = GetMousePosition();

    // Appliquer l'effet hover pour chaque bouton de voiture
    if (startMazeBtnY->isHovered(mousePos)) {
        startMazeBtnY->DrawWithScale(1.1f);
    } else {
        startMazeBtnY->Draw();
    }

    if (startMazeBtnB->isHovered(mousePos)) {
        startMazeBtnB->DrawWithScale(1.1f);
    } else {
        startMazeBtnB->Draw();
    }

    if (startMazeBtnR->isHovered(mousePos)) {
        startMazeBtnR->DrawWithScale(1.1f);
    } else {
        startMazeBtnR->Draw();
    }
}
```

Cette méthode est utilisée pour afficher l'écran de sélection de la couleur de la voiture avant de commencer un labyrinthe.

Elle affiche un autre fond (carColorsBackground) et applique le même comportement de survol (hover effect) sur les boutons de voiture (jaune, bleu, rouge), en augmentant leur taille lorsqu'ils sont survolés.

5_Méthode verifierSelection() :

```
int Niveau::verifierSelection(Vector2 mousePos, bool mousePres) {
    if (facileBtn->isPressed(mousePos, mousePres)) return 13;
    if (moyenBtn->isPressed(mousePos, mousePres)) return 17;
    if (difficileBtn->isPressed(mousePos, mousePres)) return 21;
    return 0;
}
```

Cette méthode permet de vérifier quel bouton de niveau a été cliqué par l'utilisateur. Elle vérifie si l'un des boutons (facileBtn, moyenBtn, ou difficileBtn) est pressé, et retourne une valeur correspondant au niveau choisi et si aucun bouton n'est pressé, la méthode retourne 0.

V_ Chronometre :

La classe Chronometre est utilisée pour gérer un chronomètre dans un jeu ou une application interactive. Elle permet de démarrer, arrêter, mettre à jour et afficher un timer en fonction du temps écoulé. Voici une description détaillée c différentes parties de son implémentation.

1_Constructeur :

```
Chronometre::Chronometre() : timer(0.00f), timerRunning(false) {}
```

- **Objectif :**
- Initialiser un objet Chronometre avec des valeurs par défaut.
- **Attributs :**
- timer : Initialisé à 0.00f. Représente le temps écoulé en secondes.
- timerRunning : Initialisé à false. Indique si le chronomètre est en cou de fonctionnement.
- **Fonctionnement :**

Ce constructeur ne prend aucun paramètre et configure un chronomèt prêt à l'emploi avec une valeur initiale à zéro et non démarré.

2_Méthode : void startTimer()

```
void Chronometre::startTimer() {
    timerRunning = true;
}
```

- **Objectif :**

- Démarrer ou relancer le chronomètre.
- **Fonctionnement :**

- La méthode modifie l'attribut `timerRunning` pour le passer à `true`.
- Le timer commence à s'incrémenter (dans la méthode `updateTimer`) dès que cette méthode est appelée.

3_Méthode : void stopTimer()

```
void Chronometre::stopTimer() {
    timerRunning = false;
}
```

- **Objectif** :

- Arrêter temporairement le chronomètre.
- **Fonctionnement** :

- La méthode met l'attribut `timerRunning` à `false`.
- Une fois arrêté, le temps accumulé reste stocké dans `timer` et n'est plus mis à jour tant que `startTimer()` n'est pas réappelée.

4_Méthode : void updateTimer()

```
void Chronometre::updateTimer() {
    if (timerRunning) {
        timer += GetFrameTime();
    }
}
```

Objectif :

- Mettre à jour la valeur de `timer` en fonction du temps écoulé.
- **Fonctionnement** :

- La méthode vérifie d'abord si `timerRunning` est `true` :
Si oui, elle récupère le temps écoulé depuis le dernier frame
avec `GetFrameTime()` et l'ajoute à `timer`.
- `GetFrameTime()` retourne un `float` représentant le temps écoulé entredeux frames, ce qui permet d'incrémenter le chronomètre en temps réel.
- Si `timerRunning` est `false`, la méthode n'a aucun effet.

5_Méthode : void drawTimer()

```
void Chronometre::drawTimer(int screenWidth, int screenHeight, Font font) {
    DrawTextEx(font, TextFormat("Votre Score : %.2f s", timer), {12, 12}, 40, 1,
BLACK);
    DrawTextEx(font, TextFormat("Votre Score : %.2f s", timer), {10, 10}, 40, 1,
WHITE);
```

```
}
```

- **Objectif :**

Afficher la valeur actuelle du chronomètre à l'écran avec un style visuel.

- **Paramètres :**

- `screenWidth` et `screenHeight` : Dimensions de l'écran. Ici, ils ne son pas directement utilisés mais pourraient être employés pour positionner dynamiquement le texte.

`font` : Police utilisée pour dessiner le texte.

- **Fonctionnement :**

- La méthode affiche la valeur actuelle de `timer` en secondes avec deux décimales.
- `TextFormat` est utilisé pour formater la chaîne de caractères en insér la valeur de `timer`.

Deux appels à `DrawTextEx` sont effectués :

- Le premier affiche le texte en noir légèrement décalé pour créer un effe d'ombre (position `{12, 12}`).
- Le second affiche le texte en blanc à la position principale `{10, 10}`.

Cela améliore la lisibilité du texte sur des arrière-plans clairs ou complexes.

VI_ Boucle principale du jeu :

La fonction principale (int main) initialise la fenêtre et les ressources graphiques, puis affiche un menu principal avec des options de démarrage et de sortie. Le joueur choisit la difficulté et la couleur de son véhicule. Le gameplay consiste à naviguer dans un labyrinthe généré aléatoirement pour atteindre une maison, avec un chronomètre mesurant le temps. Un système de meilleurs scores enregistre les performances, et les touches ESPACE et H permettent de réinitialiser le niveau ou de retourner au menu. Le jeu inclut également un contrôle du volume et gère proprement les ressources à la fermeture.

1_Initialisations:

```
// Fonction principale
int main() {
    // Dimensions de la fenêtre
    int screenWidth = 900; // Largeur de la fenêtre
    int screenHeight = 900; // Hauteur de la fenêtre

    // Initialisation de la fenêtre Raylib
    InitWindow(screenWidth, screenHeight, "Road To Home"); // Créer la fenêtre

    SetTargetFPS(60); // Limiter à 60 images par seconde

    // Load textures once
    Texture2D background = LoadTexture("Graphics/Backgrounds/HOME.png");
    Texture2D houseTexture = LoadTexture("Graphics/House/house.png");
    Texture2D bestRecordBackground =
LoadTexture("Graphics/Backgrounds/BEST_RECORD.png");
    Texture2D mazeBackground = LoadTexture("Graphics/Backgrounds/Maze_Back.png");
    Texture2D joueurTextureRight, joueurTextureLeft, joueurTextureUp,
joueurTextureDown;

    button startButton("Graphics/Buttons/start.png", {350, 230}, 0.7); // Appel de la
classe button pour créer un bouton de démarrage
    button exitButton("Graphics/Buttons/exit.png", {365, 420}, 0.6); // Appel de la
classe button pour créer un bouton de sortie

    Font customFont = LoadFontEx("Graphics/Fonts/AfacadFlux.ttf", 80, 0, 0); //
Charger votre police personnalisée avec taille et gras

    bool exit = false; // Argument booléen initialisé à false
    bool gameStarted = false; // Indicateur de démarrage du jeu
    bool difficultyMenu = false;
    int largeur = 0; // Largeur du labyrinthe / facile = 13, moyen = 17, difficile =
21
    int hauteur = 0; // Hauteur du labyrinthe / facile = 13, moyen = 17, difficile =
21

    Niveau niveau;
    Labyrinthe* labyrinthe = nullptr;

    Joueur joueur(joueurTextureRight, joueurTextureLeft, joueurTextureUp,
joueurTextureDown, houseTexture, 1, 1);

    // Remplacer l'initialisation de l'audio et du bouton de volume par :
```

```
    button volumeButton("Graphics/Volumes/volume-up.png", {(float)(screenWidth - 70),
20}, 0.60, true);

    bool CarsColors = false; // Indicateur pour afficher les couleurs de voiture

    // Initialiser la classe Chronometre
    Chronometre chronometre;  // Remplacer Jeu jeu
```

Configuration de la fenêtre :

Création d'une fenêtre 900x900 pixels avec InitWindow(), titre "Road To Home" et FPS limité à 60.

Chargement des textures :

Chargement des ressources graphiques depuis le dossier "Graphics" : backgrounds, textures du joueur, maison et autres éléments visuels.

Création des éléments d'interface :

Création des boutons start (350,230) et exit (365,420), initialisation de la police AfacadFlux.ttf et du bouton de volume.

Initialisation des variables de contrôle :

Déclaration des booléens de contrôle (exit, gameStarted, difficultyMenu, CarsColors) et des dimensions du labyrinthe.

Création des objets de jeu :

Instanciation des classes principales : Niveau, Labyrinthe, Joueur et Chronometre.

2_Gestion des menus:

Menu Principal :

```
if (!gameStarted) { // Si le jeu n'a pas encore commencé

            if (!difficultyMenu) { // Si le menu de difficulté n'est pas affiché

                // Menu principal
                DrawTexture(background, 0, 0, WHITE);

                // Effet de hover sur le bouton start et exit
                if (startButton.isHovered(mousePos)) {
                    startButton.DrawWithScale(1.1f); // Agrandir de 10% quand survolé
                } else {
                    startButton.Draw(); // Taille normale
                }

                if (exitButton.isHovered(mousePos)) {
                    exitButton.DrawWithScale(1.1f); // Agrandir de 10% quand survolé
                } else {
                    exitButton.Draw(); // Taille normale
                }

                // Dessiner les boutons start et exit
                if (startButton.isPressed(mousePos, mousePres)) {
                    difficultyMenu = true;
                }
                if (exitButton.isPressed(mousePos, mousePres)) {
                    exit = true;
                }
            }
```

Le menu principal s'affiche lorsque le jeu n'a pas encore commencé (gameStarted = false). Il présente le fond d'écran principal (background) et deux boutons : "start" et "exit". Les boutons réagissent au survol de la souris en s'agrandissant de 10%. Si le bouton "start" est cliqué, le jeu passe au menu de difficulté (difficultyMenu = true). Si le bouton "exit" est cliqué, le jeu se ferme (exit = true).

Menu de Difficulté :

```
else { // Si le menu de difficulté est affiché
```

```
                if (!CarsColors) { // Si le menu de sélection de couleur de voiture n'est
pas affiché


                    niveau.afficher(); // Afficher le menu de difficulté
                    int Nouvelletaille = niveau.verifierSelection(mousePos, mousePres);
                    if (Nouvelletaille > 0) {
                        largeur = hauteur = Nouvelletaille;
                        labyrinthe = new Labyrinthe(largeur, hauteur);
                        CarsColors = true; // Afficher le bouton de début du labyrinthe
                    }
                }
```

Ce menu s'affiche lorsque difficultyMenu = true. Il propose trois niveaux de difficulté (facile, moyen, difficile) qui déterminent la taille du labyrinthe (largeur et hauteur). La sélection d'une difficulté initialise un nouveau labyrinthe et passe au menu de sélection des voitures (CarsColors = true).

Menu de Sélection des Voitures :

```
else { // Si le menu de sélection de couleur de voiture est affiché

                    niveau.afficherStartMaze();
                    int carSelection = niveau.verifierStartMaze(mousePos, mousePres);
                    if (carSelection > 0) {
                        if (carSelection == 1) {
                            Joueur::DessinerVoitureJaune(joueurTextureRight,
joueurTextureLeft, joueurTextureUp, joueurTextureDown);
                        } else if (carSelection == 2) {
                            Joueur::DessinerVoitureBleue(joueurTextureRight,
joueurTextureLeft, joueurTextureUp, joueurTextureDown);
                        } else if (carSelection == 3) {
                            Joueur::DessinerVoitureRouge(joueurTextureRight,
joueurTextureLeft, joueurTextureUp, joueurTextureDown);
                        }
                        gameStarted = true;
                        CarsColors = false;
                    }
                }
            }
        }
```

Ce menu s'affiche lorsque CarsColors = true. Il permet de choisir parmi trois couleurs de voitures : jaune, bleue et rouge. La sélection d'une voiture

initialise les textures du joueur et démarre le jeu (gameStarted = true).

Gestion des Transitions :

Les transitions entre les menus sont gérées par des booléens et des interactions de la souris :

- Menu Principal → Menu Difficulté : Clic sur le bouton "start"
- Menu Difficulté → Sélection Voiture : Choix de la difficulté
- Sélection Voiture → Jeu : Choix de la voiture
- Retour au menu principal : Touche H
- Quitter le jeu : Bouton "exit" ou touche Escape

3_ Phase de jeu :

Compte à rebours :

Avant de commencer le jeu, un compte à rebours de 3 secondes est affiché.

Jeu en cours :

Une fois le compte à rebours terminé, le labyrinthe est dessiné, et le joueur peut se déplacer. Le chronomètre est démarré pour suivre le temps de jeu.

Mise à jour et dessin :

Le labyrinthe et le joueur sont mis à jour et dessinés à chaque itération. Le chronomètre est également mis à jour et affiché.

Gestion des événements :

Les touches ESPACE et H permettent de réinitialiser le labyrinthe ou de retourner au menu principal, respectivement. La touche ESCAPE permet de

quitter le jeu.

4_Fin de la boucle :

La boucle se termine par EndDrawing(), qui finalise le processus de dessin pour l'itération actuelle. Si le joueur atteint la sortie du labyrinthe, le chronomètre est arrêté, et les scores sont affichés. Le meilleur score est mis à jour si le score actuel est meilleur.

5_Nettoyage des ressources :

Avant de quitter le jeu, les ressources allouées (textures, polices, objets) sont libérées pour éviter les fuites de mémoire. La fenêtre Raylib est ensuite fermée avec CloseWindow().