



Rapport Technique Exhaustif : Analyse Architecturale du Jeu de Labyrinthe

Réalisé par : Ibrahim El Hafdaoui
OUssama Allouch
Elmehdi Elkhaldi

Encadré par : Ikram Benabdelouahab



1. Structure Globale du Code

1.1 Organisation des Composants

- Utilisation de la Programmation Orientée Objet (POO)
- 4 classes principales : `button`, `Maze`, `Player`
- Séparation claire des responsabilités
- Conception modulaire et extensible

2. Analyse Détaillée des Classes

2.1 Classe Button

Objectif

- Gestion des éléments interactifs et graphiques
- Création de boutons avec des textures personnalisées

Attributs Privés

- Texture : Stockage de la texture du bouton
- Position : Position du bouton à l'écran
- Facteur d'échelle : Mise à l'échelle de l'image
- État de survol : Détection de l'interaction souris

Méthodes Clés

- Constructeur : Chargement et redimensionnement dynamique de l'image
- draw() : Rendu du bouton avec effet de survol
- ispressed() : Détection des interactions utilisateur
- updateHoverState() : Mise à jour de l'état de survol

2.2 Classe Maze

Algorithme de Génération

- Type : Génération de labyrinthe par Backtracking récursif
- Technique : Création de chemins par exploration en profondeur

```
void generateMaze(int x, int y) {
    maze[x][y] = 0;
    int directions[] = {0,1, 2, 3};
    std::random_shuffle(std::begin(directions), std::end(directions));
    for (int i = 0; i < 4; ++i) {
        // Exploration des directions adjacentes
    }
}
```

Caractéristiques

- Matrice 2D représentant le labyrinthe
- 1 = Mur
- 0 = Chemin
- Génération aléatoire à chaque exécution

2.3 Classe Player

Gestion des Mouvements

- Contrôles directionnels par touches fléchées
- Vérification des collisions avant déplacement
- Limitation aux chemins valides

Méthodes Importantes

- Déplacement : Vérification de la validité du mouvement
- Condition de victoire : Détection de l'arrivée
- Ajustement de position : Prévention des sorties de bordure

Gestion des Mouvements

- `move()` : Déplacement avec vérification de validité
- `hasWon()` : Condition de victoire
- `adjustPosition()` : Prévention des sorties de bordure

3. Algorithme de Génération de Labyrinthe

3.1 Principes Fondamentaux

- Algorithme : Backtracking récursif
- Approche : Exploration et création de chemins
- Randomisation : Utilisation de `std::random_shuffle()`

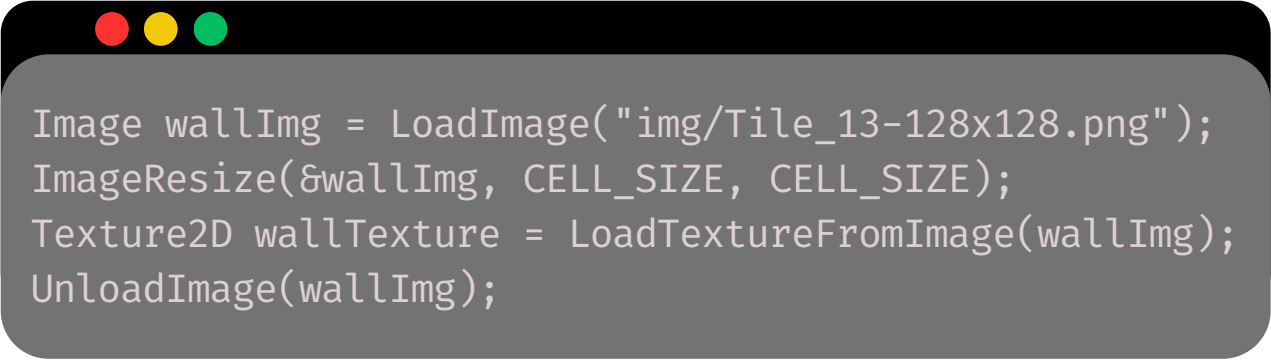
3.2 Étapes de Génération

1. Initialisation du labyrinthe avec des murs
2. Sélection aléatoire des directions
3. Création de chemins par élimination des murs
4. Exploration récursive complète

4. Gestion des Ressources Graphiques

4.1 Chargement des Textures

- Utilisation de Raylib
- Redimensionnement dynamique
- Libération explicite de la mémoire



```
Image wallImg = LoadImage("img/Tile_13-128x128.png");
ImageResize(&wallImg, CELL_SIZE, CELL_SIZE);
Texture2D wallTexture = LoadTextureFromImage(wallImg);
UnloadImage(wallImg);
```

5. Boucle de Jeu Principale

5.1 Structure

- Initialisation de la fenêtre
- Chargement des ressources
- Boucle de rendu principale
- Gestion des événements

5.2 Séquence d'Exécution

1. Initialisation du labyrinthe
2. Création du joueur
3. Boucle de rendu
 - Effacement de l'écran
 - Dessin du labyrinthe
 - Mise à jour du joueur
 - Vérification de la victoire

6. Analyses Avancées

6.1 Complexités

- Génération de Labyrinthe : Complexité quadratique
- Rendu Graphique : Proportionnel à la taille du labyrinthe
- Mouvements du Joueur : Opérations constantes

6.2 Points Potentiels d'Optimisation

- Utiliser `std::mt19937` au lieu de `std::rand()`
- Implémenter un système de pré-génération de labyrinthe
- Optimiser l'algorithme de génération

7. Recommandations d'Amélioration

7.1 Évolutions Possibles

1. Système de niveaux dynamiques
2. Génération de labyrinthe plus complexe
3. Ajout d'éléments de gameplay
4. Système de score et de chronométrage
5. Gestion des erreurs plus robuste

8. Dépendances Techniques

8.1 Bibliothèques

- Raylib (Graphismes et interactions)
- Bibliothèques standard C++
- Standard C++ : C++11/14

