The Dedale project

Mathis Boultoureau
mboultou@enssat.fr

Yvan Allioux
yallioux@enssat.fr

Emilie Daniel
edaniel@enssat.fr

Abstract

1. Introduction

Ce projet a été donné dans le contexte du cours « Contenus multimédias » en 2ème année de cycle ingénieur à l'ENSSAT. Il consiste à créer un jeu de labyrinthe à l'aveugle.

2. Attribution des tâches

Pour accélérez le développement, nous avons divisé en 3 tâches qui sont attribuées à chacun des membres du groupe. Voici les tâches attribuées à chacun des membres du groupe :

- Mathis : Affichage graphique du labyrinthe pour pouvoir débugger plus facilement
- Yvan : Génération aléatoire du labyrinthe
- Emilie : Gestion du son

3. Ecriture en 2D puis passage en 3D

Nous avons commencé par écrire le code pour faire le labyrinthe en 2D pour commencer simplement. Nous avons pour cela utilisé la bibliothèque SDL2. Lors du passage en 3D, nous nous sommes heurté au fait que nous ne pouvons mélanger les objets SDL2 (par exemple, les textures, rectangles) et OpenGL sur la même fenêtre. Nous avons donc réécrit le code pour utiliser uniquement OpenGL à l'aide du code fourni en exemple.

4. Affichage graphique du labyrinthe

Sur la partie 3D, nous avons repris l'objet cube présent dans le code initial et nous avons changé les dimensions pour former les murs. Nous avons ensuite positionné tous les murs suivant les positions et directions (nord, est, sud ou ouest) des murs.

5. Génération aléatoire du labyrinthe

5.1. Choix de l'algorithme

Initialement l'algorithme de génération utilisé était l'algorithme de Kruskal. J'ai décidé de passer à la génération d'algorithme DFS (Depth-First Search) pour plusieurs raisons. Tout d'abord, l'algorithme de Kruskal était très lent et complexe à coder. Il fallait casser les murs de manière aléatoire, ce qui pouvait entraîner des boucles dans le labyrinthe et ralentir considérablement le processus de génération. De plus, l'algorithme de Kruskal avait tendance à supprimer trop de murs, ce qui conduisait à des labyrinthes avec très peu de murs et des chemins trop ouverts. En passant à l'algorithme DFS, j'ai pu résoudre ces problèmes. L'algorithme DFS explore le labyrinthe en profondeur, en créant des chemins plus étroits et en conservant un nombre suffisant de murs pour rendre le labyrinthe plus complexe. De plus, l'algorithme DFS est plus simple à implémenter et génère des labyrinthes plus rapidement, ce qui en fait un choix plus efficace pour la génération de labyrinthes.

Voici une liste simplifiée des avantages de l'algorithme DFS :

1. Simplicité de mise en œuvre.
2. Génération plus rapide.
3. Contrôle du nombre de murs.
4. Flexibilité et adaptation facile.

5.2. Fonctionnement de l'algorithme de Kruskal

L'algorithme de Kruskal utilisé dans le code génère un labyrinthe en cassant progressivement les murs entre les cellules pour créer des connexions entre les zones. Cela se fait en utilisant le tableau des zones pour suivre les connexions entre les cellules et en utilisant des opérations aléatoires pour choisir les murs à casser. L'algorithme se termine lorsque toutes les cellules sont connectées.

1. Le constructeur de la classe MazeGenerator prend en compte les dimensions du labyrinthe (largeur, longueur et hauteur) et initialise la taille du labyrinthe ainsi que sa représentation sous forme de matrice tridimensionnelle appelée « m_Matrice ». Chaque cellule du labyrinthe est représentée par un entier de 4 bits initialisé à la valeur 15.
2. Une liste « tableauDesZones » est créée pour suivre les zones connexes du labyrinthe. Chaque cellule du labyrinthe est initialement dans sa propre zone.
3. L'algorithme procède ensuite à la création du labyrinthe en utilisant le principe de l'algorithme de Kruskal. L'idée est de casser progressivement les murs entre les cellules pour créer des connexions entre les zones.
4. L'algorithme choisit aléatoirement un mur entre deux cellules adjacentes et vérifie si elles appartiennent déjà à la même zone. Si c'est le cas, le mur n'est pas cassé car cela créerait une boucle dans le labyrinthe. Sinon, le mur est cassé et les deux zones sont fusionnées en mettant à jour le tableau des zones.
5. L'algorithme répète l'étape 4 jusqu'à ce que toutes les cellules du labyrinthe appartiennent à la même zone,

ce qui signifie que toutes les cellules sont connectées et qu'il n'y a pas de boucles.
6. Pendant le processus de génération du labyrinthe, la méthode « suprimeMur » est utilisée pour casser un mur entre deux cellules données dans une certaine direction (Nord, Sud, Est ou Ouest). Cette méthode met à jour les valeurs des entiers de 4 bits pour représenter la suppression du mur.
7. La méthode « memeZonePartout » est utilisée pour vérifier si toutes les cellules appartiennent à la même zone. Cela permet de savoir quand l'algorithme de Kruskal a terminé la génération du labyrinthe.
8. Enfin, la méthode « enregistrementMatriceDansFichier » est utilisée pour enregistrer la matrice du labyrinthe dans un fichier texte.

5.3. Fonctionnement de l'algorithme Random Depth First (DFS)

L'algorithme Random Depth First (RDF) est une méthode de génération de labyrinthe qui utilise une approche basée sur la recherche en profondeur.

L'algorithme DFS utilisé dans le code génère un labyrinthe en explorant récursivement les cellules et en supprimant les murs entre les cellules voisines. Il garantit que toutes les cellules sont accessibles à partir de la cellule de départ, créant ainsi un labyrinthe connecté sans boucles.

1. Le constructeur de la classe MazeGeneratorDFS prend en compte les dimensions du labyrinthe (largeur, longueur et hauteur) et initialise la taille du labyrinthe ainsi que sa représentation sous forme de matrice (vecteur de vecteurs) appelée « maze ». Chaque cellule du labyrinthe est représentée par un entier de 4 bits initialisé à la valeur 15.
2. L'algorithme DFS (implémenté dans la méthode « dfs ») est utilisé pour générer le labyrinthe en explorant les cellules du labyrinthe de manière récursive.
3. L'algorithme commence par une cellule de départ choisie aléatoirement. Pour chaque cellule visitée, il mélange aléatoirement les directions possibles (Nord, Sud, Est, Ouest) pour décider quelles cellules voisines explorer en premier.
4. Pour chaque direction mélangée, l'algorithme vérifie si la cellule voisine correspondante est valide et non visitée. Si c'est le cas, il supprime le mur entre la cellule courante et la cellule voisine, en mettant à jour les valeurs des entiers de 4 bits de chaque cellule pour représenter la suppression du mur dans la direction appropriée.
5. L'algorithme DFS continue à explorer récursivement les cellules voisines non visitées jusqu'à ce qu'il atteigne une impasse, c'est-à-dire qu'il n'y ait plus de cellules voisines non visitées à explorer.
6. L'algorithme revient ensuite en arrière en remontant la pile des appels récursifs jusqu'à ce qu'il trouve une cellule avec des voisins non visités. Il reprend alors l'exploration à partir de cette cellule.
7. L'algorithme se termine lorsque toutes les cellules du labyrinthe ont été visitées, ce qui signifie que toutes les connexions entre les cellules ont été établies et que le labyrinthe est entièrement généré.

6. Gestion du son

7. Difficultés rencontrées

8. Améliorations possibles

Voici quelques améliorations possibles :

- Ajout d'une 3ème dimension (hauteur) pour le labyrinthe
- Amélioration des performances : à chaque déplacement on vérifie la collision avec chaque mur au lieu de seulement ceux qui sont proches du joueur