Projet d'Informatique n.3 : le Labyrinthe

Clément Aubin & Sarah-Emmanuelle Nehme

31 mars 2014

Table des matières

Introduction

L'objectif de ce projet était de réaliser un labyrinthe en mode console, en deux ou trois dimensions selon les données entrées par l'utilisateur, mais généré aléatoirement. Nous avons également cherché à porter ce labyrinthe en SDL (c'est à dire en interface graphique) pour un résultat esthétiquement plus agréable. Il faudra que le joueur atteigne l'arrivée à travers les dédales de pixels. On peut utiliser différents arguments pour afficher le jeu :

- `-c` pour lancer la génération et la résolution d'un labyrinthe sur le terminal
- `-menu` programme de base : affichage du labyrinthe basique
- `--pauStyle` résolution du problème comme demandé à Pau
- `--cergyStyle` lancer le jeu sur interface graphique
- `--avengers` lance le jeu avec un thème qui plaira aux fans de Marvel©
- `-gui` permet de profiter du jeu en version plus évoluée

Problématique : L'objectif initial était de générer un labyrinthe aléatoirement, de manière la plus rapide possible, quelque soit la taille demandée et sans qu'il ne se forme de cycles. Il fallait ensuite créer un programme capable de rechercher puis dévoiler le plus court chemin à l'utilisateur.

Comment jouer ? Il suffit d'utiliser les flèches du clavier pour se déplacer sur le plateau de jeu. Si vous voulez monter un étage, utilisez "`pgup`", pour descendre "`pgdown`". Vous pouvez afficher à tout moment un chemin de petits cailloux qui vous mènera vers la sortie en appuyant sur la touche "`H`".
Bon jeu !

Chapitre 1

Réflexion préliminaire à la génération du labyrinthe

Entreprenons ensemble l'explication de l'algorithme utilisé pour l'affichage du labyrinthe, corps essentiel du programme, dont s'est occupé Clément.
La première chose à faire était de créer une matrice à travers la structureTheMatrix comprenant plusieurs éléments :

- map : tableau à à trois dimensions d'entiers
- wallsList : tableau de structure *singleWallInfo* contenant trois composantes `x,y,z` correspondant aux coordonnées des murs à casser
- n : entier (utilisé dans la fonction récursive du chemin le plus court)
- globalVars : structure permettant au démarrage d'interpréter les arguments fournis par l'utilisateur pour ne modifier que ces variables

Nous utiliserons également le type adjacentValuesArray, qui est un tableau d'entiers comprenant six valeurs : celles des cases adjacentes.
Bien entendu, le labyrinthe généré doit à chaque fois être différent, d'une partie à une autre, ce qui instaure une difficulté supplémentaire, donc l'utilisation d'un *random*.

Chapitre 2

La destruction des murs de manière aléatoire

Nous travaillerons ici sur une matrice représentant le plateau de jeu, avec ses murs et ses chemins possibles. L'objectif du programme de génération du labyrinthe est de "casser" ces murs de manière à constituer les chemins, sans former de cycles. La taille du tableau est demandée à l'utilisateur dès le début de la partie, ce qui permet d'initialiser la matrice en deux temps :

La procédure `initMatrix` permet de remplir la matrice de valeurs. Nous travaillerons ici sur un tableau de $n \times m \times k$ cases (valeurs déterminés par l'utilisateur) dont chaque ligne paire sera constituée de -1 et chaque ligne impaire alternera les -1 et les entiers naturels (de 0 à j), les -1 représentants les **murs**. Par le biais de cette procédure, nous créons également un tableau WALLSLIST contenant les coordonnées des cases ayant pour valeurs -1, donc pouvant être détruites (c'est à dire ne se situant pas à la lisière du plateau de jeu).
La procédure `generateMap` parcourt ensuite cette matrice jusqu'à ce que le booléen *isGenerationOk* soit vrai. Il le sera lorsque la matrice ne sera constituée que des valeurs -1 et 0.

1. Jusque là, à chaque passage, on appelle la procédure generateNewPath qui permet de détruire un mur. On utilise ici le type *adjacentValuesArray* et le tableau précédemment créé *wallsList*. On utilise le `random` sur toutes les cases du *wallsList* pour sélectionner au hasard quel mur on va détruire.

2. Appel de la fonction getAdjacentValues qui permet d'associer aux cases adjacentes, dont on connait les coordonnées, leurs valeurs respectives, puis de les stocker dans un tableau de *adjacentValuesArray* passé en paramètres. Nous utilisons également la procédure *sortAdjValues* permettant de trier ces valeurs grâce au **tri à bulles**

3. La condition permet ici de vérifier que l'on ne forme pas de cycles. Si elle est respectée, on prend le minimum des cases alentours (autre que -1) puis on affecte ce minimum à la case correspondant au mur que l'on cherche à détruire. Ensuite, on HARMONISE les valeurs alentours (c'est à dire qu'on leur donne la valeur minimum si elles sont différentes de -1), via la procédure *harmonizeValues*

4. Même si la condition n'est pas respectée, les coordonnées de ce mur destructible sont retirées du tableau WALLSLIST via les procédures *exchangeWallsInfo* qui échange cette case avec la dernière case du tableau, et la fonction de base *SetLenght* qui permet de réduire de 1 la taille du tableau.

Chapitre 3

Transformation du tableau en labyrinthe

Cette partie se fait très aisément. On utilise la fonction *displayShellMap* qui permet le passage des chiffres aux "dessins" : si l'argument de départ était nul, le labyrinthe s'affichera uniquement en deux dimensions de manière très peu ergonomique, c'est pour cela que "`-menu`" permet un affichage plus esthétique.

Chapitre 4

L'algorithme du Plus Court Chemin

Les cases de départ et d'arrivée sont soit définies par l'utilisateur, soit, dans le cas du jeu en lui-même, assignées automatiquement à la première case en haut à gauche de l'étage du bas pour le départ, et à la case en bas à droite de l'étage le plus haut pour l'arrivée. On appelle ici la procédure *generateSolution* qui permet de trouver le plus court chemin jusqu'à la sortie, en faisant appel à deux procédures :

1. *littlePath* est une fonction récursive contenant un compteur **n** qui commence à 1. Elle s'initialise avec les coordonnées du point de départ. A chaque tour de boucle, on associe la valeur de n à la case sur laquelle on travaille, puis on "avance" d'une case et le compteur est implémenté de 1. La fonction vérifie à chaque fois que la case n'est ni un mur, ni la même case que précédemment.

2. *reversePath* : on commence à la case d'arrivée, contenant la plus grande valeur de **n**, puis on teste sur toutes les cases alentours laquelle contient la valeur **n-1**. Par récursivité, on recommence sur la case en question, etc...

Chapitre 5

L'interface graphique

Nous avons décidé de créer une interface graphique pour le jeu du labyrinthe, afin de le rendre plus ergonomique, et plus attrayant pour le joueur. Le choix des poneys s'est fait naturellement, sachant la réputation eistienne de ce point de vue là. Il existe également un thème Avengers.

Pour conclure

Ce fût un mini-projet très instructif. Il me permit personnellement de découvrir la SDL et ce qu'elle apportait à un jeu en terme d'ergonomie et d'originalité. Clément a fortement apprécié la génération du labyrinthe. On peut même dire que nous nous sommes amusés pendant toute la durée de ce projet.