Sufod

Matthieu TRAN - Romain DELAPLAINE

Mars 2022

1 Introduction

Dans le cadre du module de C++, nous avions pour but de développer une plateforme de désmonstration de différents algorithmes d'explorations. Pour cela, nous avons créée **Sufod**, un jeu simple se reposant sur une interface graphique faite à l'aide de Qt.

2 Principe

Dans **Sufod**, le joueur incarne un personnage appelé **Yugo**. **Yugo** a deux caractéristiques des points de vie et des points de nourriture. Lorsqu'une de ces deux caractéristiques atteint 0, **Yugo** meurt et la partie est finie.

Au début d'une partie, **Yugo** apparaît sur une carte générée aléatoirement représentée par une grille. Cette carte contient plusieurs éléments avec lesquels **Yugo** peut intéragir :

- Case Grise : Mur où le personnage ne peut pas aller
- Coeur : Des soins permettant de régénérer la vie de **Yugo**
- Squelette : Des monstres que **Yugo** peut tuer pour gagner des points
- Pomme : De la nourriture permettant de régénérer la faim
- Case Violette : Téléporteur permattant de changer de carte

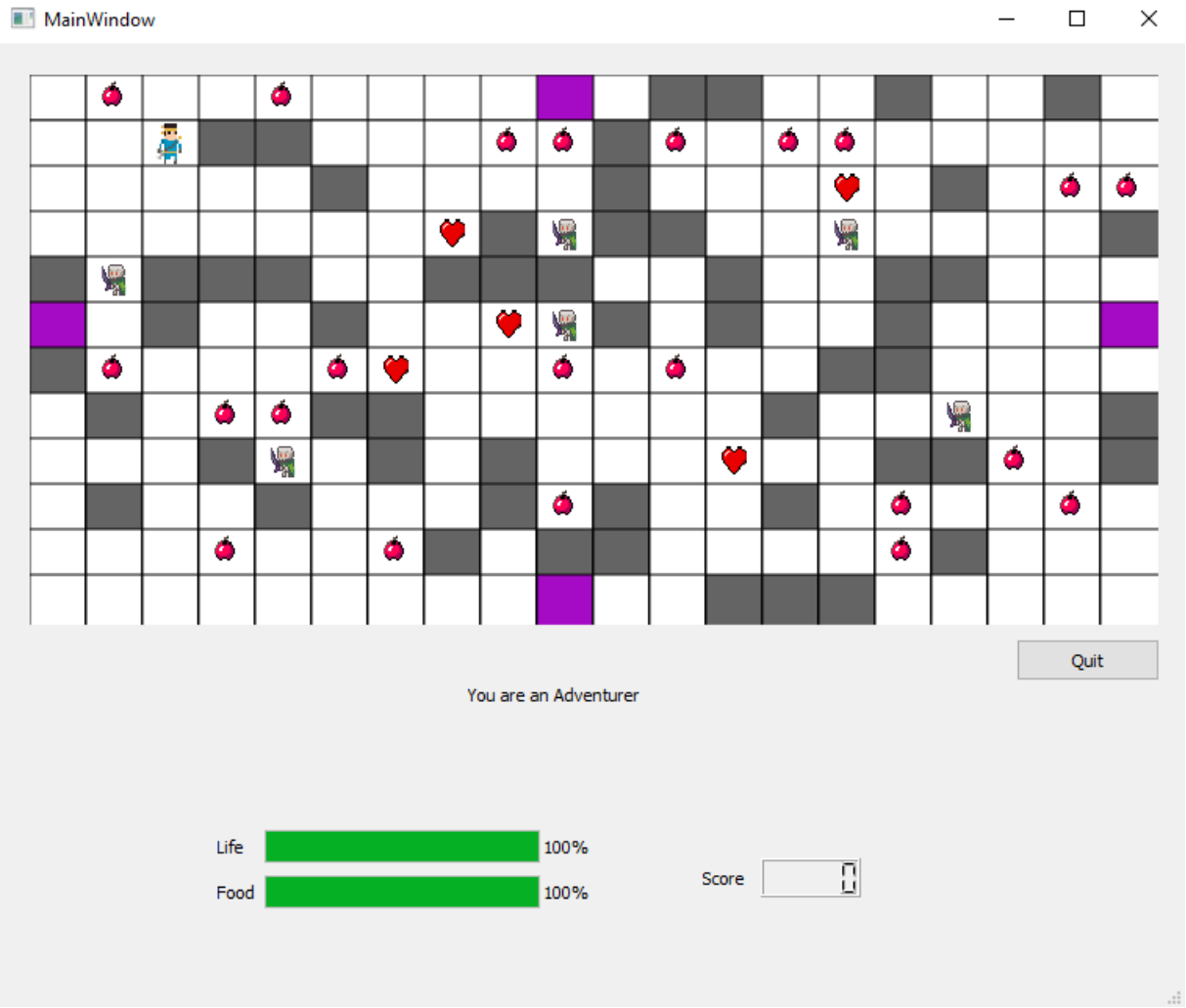

FIGURE 1 – Grille de départ

Le but du jeu est de générer le plus de points possibles. Il existe plusieurs façons de gagner des points comme se déplacer, découvrir de nouvelle carte en utilisant les portails mais surtout en se battant contre des monstres (case orange). Cependant, à chaque déplacement de case les points de nourriture de **Yugo** baissent et à chaque combat ses points de vie baissent également. Pour regagner des points de vie et de nourriture il faut aller chercher des soins ou de la nourriture se trouvant sur la carte.
Lorsque **Yugo** se déplace sur une case violette, il change de carte et se retrouve donc sur une nouvelle carte générée aléatoirement contenant de nouvelles ressources.

De plus, au lancement du jeu, le joueur peut décider de choisir sa classe entre *Adventurer*, *Warrior* et *Sorcerer*. Chacune de ses classes a différentes caractéristiques. L'*Adventurer* perd moins de nourriture, le *Warrior* perd moins de vie contre les squelettes et enfin le *Sorcerer* regagne plus de vie lorsqu'il prend des soins.

3 Fonctionnement

3.1 Carte

Tous les éléments de la carte sont générés de façon aléatoire hormis les téléporteurs (cases violettes) qui se trouvent au centre de chaque côté de la carte. Les murs sont générés en premier sur la carte. Si plusieurs murs forment une zone inaccessible, alors cette zone sera remplie de mur grace à un algotrithme de **parcours en largeur**. Si un téléporteur se trouve sur un mur alors le téléporteur sera déplacé d'une

case. C'est pour cela que les téléporteurs ne se trouvent pas tout le temps au même endroit selon la carte.

3.2 Yugo

Lorsque le joueur clique sur une case, **Yugo** se déplace vers cette case grâce à un algorithme de **Dijkstra**. Ainsi, **Yugo** prendra le chemin le plus court pour atteindre l'endroit ciblé. **Yugo** spawn de manière aléatoire sur la carte au début du jeu avec tous ses points de vie et de nourriture. Si aucune classe est sélectionnée alors la classe *Adventurer* sera choisie par défaut.

4 Algorithmie

Dans cette partie nous allons parler plus principalement des différents algorithmes que nous avons implémenté.

L'objectif premier de ce projet était de travailler sur les graphes, en réalisant notre graphe et en implémentant des méthodes de parcours pour explorer celui-ci. Pour mettre en place ce graphe nous avons choisis de creer deux structures principales : *graphe* et *noeud*. De manière globale un graphe est composé de noeuds, dans notre structure *graphe*, nous avons donc donné comme attribut à notre graphe un conteneur de *noeud*. Pour ce conteneur nous avons choisi d'utiliser des maps avec pour clef une paire d'entier et pour valeur un *noeud*, cela nous permettait d'avoir accès à chaque noeud par sa paire de coordonnées. Nous avons choisi d'utiliser une map car nous avions déjà du implémenter une grille en utilisant un conteneur semblable au vecteur. Ainsi on a pu voir comment une grille pouvait être implémentée à l'aide d'une map. Cela permettait également d'avoir un accès simple au différent noeud ce qui facilite l'implémentation.

Une fois notre graphe créé nous avons du implémenté divers algorithme de parcours afin d'explorer celui-ci. Pour l'algorithme de parcours nous avons choisis arbitrairement d'implémenter le **parcours en largeur** et pour la recherche de plus court chemin nous avons choisis d'uliser l'**algorithme de Dijkstra**.

Le parcours en largeur :

L'algorithme de parcours en largeur permet le parcours d'un graphe. Ce parcours débute a partir d'un noeud source. Puis il liste tous les voisins de la source, pour ensuite les explorer un par un. Ce mode de fonctionnement utilise donc une file dans laquelle il prend el premeir sommet et place en dernier ses voisins non explorés. Les noeuds déjà visités sont marqués afin d'éviter qu'un même noeud soit exploré plusieurs fois.

L'algorithme de Dijkstra :

L'algorithme de Dijkstra permet de résoudre le problème du plus court chemin. A partir d'un graphe pondéré (il s'agit d'un graphe dont les liens entre différents noeuds possède un poid qui peut dépendre de divers facteur comme la distance par exemple), on construit un sous graphe contenant les différents noeuds classés par ordre croissant de leur distance minimale au noeud de départ. Au début, on établit pour chaque noeud que sa distance par rapport au noeud de départ est infini, sauf pour le noeud de départ qui lui possède une distance nulle. Par la suite à chaque itération, on choisit un noeud hors de notre sous graphe (vide au départ) qui possède une distance minimale et on l'ajoute au sous-graphe. Ensuite pour chacun de ses voisins on met à jour les distances : la nouvelle distance est le minimum entre la distance actuelle et la somme entre la distance du point courant et le poids de la liaison reliant le voisin. On continu les itérations jusqu'à qu'il n'y ait plus de noeuds en dehors du sous graphe.

Nous avons également dans notre jeu la possibilité de choisir entre trois classes de personnage. Pour cela on a implémenté une structure *perso* qui est notre structure mère et qui possède trois structures filles : *aventurier* , *guerrier* , *sorcier*. Nous avons ainsi utilisé le polymorphisme afin de différencier les méthodes ayant pour rôle de changer les niveaux de vie et de nourritures. Ainsi en fonction de l'instance de notre personnage, la méthode appelée sera celle la plus spécifique (utilisation de méthode virtuelle dans notre structure mère).

5 Conclusion

Ce projet nous a permis d'avoir une meilleure approche de la programmation en C++. De plus, nous avons pu prendre en main et nous familiariser avec l'IDE Qt.
Nous avons également assimilé les différents algorithmes de parcours en particulier ceux de Dijjkstra et de parcours en largeur. Nous avons aussi pu voir le principe des graphes en les implémantant.

Remarque : Détail pour la compilation : pour que les images soient trouvées par le programme il faut que le dossier de debug créé par Qt soit dans le dossier Projet.
Si les images ne sont pas trouvées, il s'affichera sur la grille des cases de couleurs rouge, verte, et orange représentant respectivement la vie, la nourriture et les monstres.