Projet Programmation 2 : Rapport Partie 2

MANGEL Léo et MICHEL Thomas

1 Fonctionnalités ajoutés

1.1 Personnages et entités

1.1.1 Nouveaux traits

- Le trait `InteractWithItems` : Si le joueur interagit avec une entité ayant ce trait, la méthode `itemInteraction` est appelée avec en paramètre l'objet actuellement sélectionné par le joueur. Cela permet d'intégrer de nouvelles interactions en fonction des objets possédés par le joueur.
- Le trait `Hackable` : Dérive de `InteractWithItems` et ajoute un appel à la méthode hack lorsque l'objet sélectionné est "Hacking Tools".
- Le trait `FleeingWhenDamaged` : Dérive de `AIControlled`. Si le personnage reçoit des dommages, il se met à fuire. Pour déterminer sur quelle case le personnage fuit, on utilise le même algorithme pour calculer le plus court chemin entre le joueur et le personnage, mais on prend la case opposée à la place. Si la case opposée est occupée, par un mur ou un autre personnage, on regarde les cases adjacentes, dans le sens horaire puis trigonométrique, jusqu'à en trouver une libre.
- Le trait `RangedEnemy` : Implémente un nouveau comportement pour les ennemis. Pas de déplacement, attaque lorsque le joueur est à portée de tir. Pour savoir s'il est possible de toucher le joueur (si la ligne de visée n'est pas obstruée), nous avons choisi de nous basé sur l'algorithme calculant les cases visibles par le joueur implémenté durant la partie 1 du projet. Nous faisons l'hypothèse que si le joueur peut voir un ennemi alors celui-ci peut aussi voir le joueur. Nous nous étions en effet efforcé de rendre l'algorithme symétrique lorsque nous l'avions implémenté. De cette façon, il n'est pas nécessaire de calculer les cases visibles par chaque ennemi mais seulement celles visibles par le joueur puis d'autoriser les tirs ennemis si ceux-ci sont visibles sur la carte et la distance séparant le joueur de l'ennemi est inférieures à la portée de ce dernier.

1.1.2 Une boutique

Afin de diversifier les personnages non-joueurs, nous avons décidé de mettre en place une boutique accessible par l'intermédiaire d'un marchand ainsi qu'une monnaie permettant d'acheter des objets dans la boutique. La classe `Shopkeeper` dérive de la classe `Character` et possède le trait `FleeingWhenDamaged`. L'interaction par défaut est remplacée par l'ouverture d'un menu (classe `ShopMenu`) affichant les objets disponibles à la vente et se mettant à jour à chaque nouvel achat du joueur. De plus, bien qu'il ne soit pas possible d'attaquer le marchand par une attaque au corps à corps, il est tout de même possible de lui infliger des dégâts auquel cas celui-ci se met à fuir le joueur.

1.2 Curseur, armes à distance et objets lançable

Afin de pouvoir viser de manière précise, nous avons ajouté un curseur. Il devient visible et controlâble par le joueur lorsqu'on tire avec une arme ou lorsqu'on lance un objet.
Pour cela, nous avons ajouté des modes de contrôle inspiré par l'éditeur de texte vim. Il y a pour l'instant 5 modes, dont 3 qui utilisent le curseur. Il s'agit des modes `Cursor`, `Throw` et `Fire`.
Le mode `Cursor` n'est pour l'instant pas utile et permet simplement de déplacer le curseur. Nous n'avons pas eu le temps de nous concentrer sur ce mode, mais l'objectif de ce mode serait de permettre d'inspecter les cases autour du joueur et d'afficher une description des ennemis.
Le mode `Throw` permet de lancer certains objets. Pour entrer dans ce mode, il faut sélectionner l'objet à lancer et appuyer sur T. Ensuite, on peut déplacer le curseur vers la case sur lequel on veut envoyer l'objet, puis appuyer à nouveau sur T. On ne peut pas lancer d'objet sur une case qu'on ne voit pas, ni sur un mur. Certains objets sont consommés lorsqu'ils sont lancés, comme les grenades.
Le mode `Fire` fonctionne de la même manière que le mode `Throw` mais permet lui de tirer avec des armes à distance. Pour y entrer, il faut avoir une arme à distance équipée et appuyer sur F. Si on a plusieurs armes à feu équipés, on peut soit sélectionner une de ses armes pour tirer avec, soit tirer avec l'arme principale, qui est la première arme à distance équipée, lorsque l'objet selectionné n'est pas une arme à distance équipée. Les mêmes restrictions qu'on avait pour les objets à lancer s'appliquent aussi pour les armes à distance, mais en plus, il faut

qu'il n'y ai pas un autre personnage entre le joueur et la case visée. Pour visualiser ces conditions, on affiche un halo jaune sur les cases entre le joueur et la case visée. Pour calculer les cases concernées, on a préféré utilisé un algorithme naïf plutôt qu'un algorithme plus optimisé comme celui de Bresenham. En effet, vu qu'on ne calcule les cases qu'un seul segment par tour, la complexité n'a pas beaucoup d'influence.

1.3 Sauvegarde

Pour sauvegarder les informations du jeu en cours, nous utilisons la sérialisation native de Scala. Cela nous permet de rendre chaque classe sauvegardable dans un fichier en ajoutant le trait `Serializable`. Ainsi, lorsqu'on sauvegarde, on enregistre dans un fichier un objet de la classe `Game` qui contient toutes les informations sur la partie en cours. De même, lorsqu'on charge, on désérialise le contenu du fichier pour obtenir l'objet de la classe `Game` initial.
Pour sauvegarder et charger une partie, on utilise un menu principal, auquel on a accès en début de jeu, ainsi qu'à n'importe quel moment de la partie en appuyant sur Echap. Nous avons choisi d'avoir une seule sauvegarde possible, afin de ne pas avoir besoin d'implémenter un système pour sélectionner une sauvegarde.

1.4 Niveaux et objectifs

Pour changer de niveau, il faut trouver un ascenseur descendant. Il y en a un par niveau. Au dessus de l'ascenseur, il y a un cadenas qui empêche d'accéder à l'ascenseur. Pour supprimer le cadenas, le joueur a deux choix. Soit il utilise un outil de hacking pour déverouiller le cadenas, soit il utilise une arme à distance pour détruire le cadenas. Si il détruit le cadenas, il abime aussi l'ascenseur. Cela se traduit en jeu par le fait que, après avoir utilisé l'ascenseur abimé pour descendre d'un étage, il ne pourra plus l'utiliser pour monter.
Afin d'empêcher le joueur d'être bloqué, il commence avec un "Arm-cannon" qui est une arme à distance, ce qui lui permettra de détruire le cadenas.
Pour la suite, nous aimerions augmenter la difficulté progressivement lorsqu'on descend. Cela se fera probablement en augmentant les statistiques des ennemis.

2 Difficultés rencontrées

Malgré sa grande simplicité de mise en place dans le code, nous avons eu des difficultés à faire fonctionner correctement la sérialisation. En effet, sbt empêche initialement au programme de faire des forks, ce qui est utilisé par Scala pour la désérialisation. Pour régler le problème, il faut changer le fichier `build.sbt`. Nous avons pensé un peu de temps à comprendre d'ou venait le problème et à le résoudre.