

# Projet Programmation 2 : Rapport Partie 2

MANGEL Léo et MICHEL Thomas

## 1 Fonctionnalités ajoutés

### 1.1 Curseur, armes à distance et objets lançable

Afin de pouvoir viser de manière précise, nous avons ajouté un curseur. Il devient visible et contrôlable 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 équipées, 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 sélectionné 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 ait 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.

### 1.2 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 déséréalise 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.

### 1.3 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 déverrouiller le cadenas, soit il utilise une arme à distance pour détruire le cadenas. Si il détruit le cadenas, il abîme aussi l'ascenseur. Cela se traduit en jeu par le fait que, après avoir utilisé l'ascenseur abîmé 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'où venait le problème et à le résoudre.