Projet Programmation 2 : Rapport Partie 3

MANGEL Léo et MICHEL Thomas

1 Nouvelles mécaniques de jeu

1.1 Expérience des personnages et niveaux

Nous avons ajouté des attributs et des méthodes à la classe dont héritent les personnages (`Character`) afin que ceux-ci puissent acquérir de l'expérience en éliminant d'autres personnages. La quantité d'expérience reçue est égale au niveau de l'ennemi éliminé. Le passage à un niveau supérieur améliore toutes les statistiques de base d'un personnage et augmente également la quantité d'expérience nécessaire afin de passer au niveau suivant.
Le niveau des ennemis est basé sur l'étage dans lequel se trouve le joueur mais des variations peuvent-être introduites à l'aide des fichiers parsés par le générateur d'ennemis.
Les méthodes ajoutées sont `getXp` et `levelUp` dans la classe `Character`

1.2 Portes

Nous avons ajouté des portes afin de bloquer la vision entre les salles et de mieux compartimenter la progression en rajoutant des mécanismes pouvant bloquer le joueur. Le joueur a maintenant la possibilité d'interagir avec des objets héritant de la classe `GameTile` ayant le trait `InteractableTile`. Les portes sont de tels objets. Celles-ci peuvent être ouvertes, fermées ou verrouillées. Il est possible de voir au travers des portes ouvertes mais pas des portes fermées. Un porte fermée non verrouillée peut être ouverte en se déplaçant vers la case où celle-ci se trouve.
La classe `Door` possède deux classes filles :

- La classe `KeyLockedDoor` est initialement verrouillée et peut être déverrouillée si le joueur possède une clé sur lui lorsqu'il interagit avec la porte. Les clés fonctionnent sur le même principe que l'argent, l'objet `Key` peut être ramassé, lorsque cela arrive le compteur de clés est incrémenté et l'objet n'apparaît pas dans l'inventaire.
- La classe `KeyLockedDoor` est initialement verrouillée et peut être déverrouillée lors du déclenchement d'un `Trigger`.

1.3 Déclencheurs(Triggers) et évènements

Nous avons introduit une classe `Trigger` permettant de faire le lien entre des évènements et le déclenchement d'actions. Cette classe possède deux tableaux mutables, l'un contenant des objets héritant de la classe `EventWatcher` et l'autre des objets possédant le trait `Triggerable`. Pour qu'une instance de la classe `Trigger` soit active, il faut qu'elle soit ajouté au tableau `triggers` d'une instance de `Gameboard`. Cette dernière met à jour les déclencheurs à la fin de chaque tour de jeu. Pour que le déclencheur s'active, il faut que tous les évènements contenus dans le tableau `events` se soient produit. Lors de l'activation du déclencheur, la méthode `executeAction` des objets du tableau `actions` est appelée. Une fois que les actions ont été exécutées, l'instance de `Trigger` est désactivée et ne peut plus être utilisée afin d'éviter de répéter les actions qui ont été effectuées.

1.3.1 Descendants de `EventWatcher`

- `ActivableWatcher` : Permet de surveiller l'état d'objets possédant le trait `Activable`. C'est par exemple le cas des leviers (classe `Lever`) qui est une nouvelle entité avec laquelle on peut interagir en se déplaçant vers la case sur laquelle elle se trouve. L'interaction change l'état du levier grâce à l'attribut `activated` et cet évènement est détecté par `ActivableWatcher` à la fin du tour.
- `DeathWatcher` : Cette classe surveille les points de vie d'un personnage et s'active lorsque ceux-ci atteignent 0 à la fin d'un tour.

1.3.2 Objets possédant le trait `Triggerable`

- `LogAction` : Lorsque son action est exécutée, les instances de cette classe affichent un message à l'aide d'un objet de la classe `Logger`.
- `TriggerableDoor` : Il s'agit d'une porte initialement verrouillée. Lors de l'appel à `executeAction`, celle-ci se déverrouille. Des applications simples sont de créer une porte qui se déverrouille lorsqu'un ou plusieurs leviers sont activés ou bien lorsqu'un ennemi puissant est vaincu.

1.4 Affichage

Nous avons fais en sorte que le terrain ne soit affiché qu'en partie. La zone visible est centrée sur le joueur ou bien le curseur lorsque celui-ci est visible. Cela nous a permis d'agrandir la carte tout en gardant un affichage clair des personnages et des objets.
Par ailleurs, afin de faciliter l'exploration, nous avons également décidé d'afficher en permanence l'intégralité de la carte sur l'interface du joueur afin d'avoir une vue globale de l'étage dans lequel le joueur se trouve.

2 Modification des ennemis

2.1 Effets et objets abandonnés

Afin de mieux mettre en valeur le système de statuts, les ennemis peuvent désormais appliquer des statuts en attaquant le joueur. La nature des statuts pouvant être appliqués ainsi que leurs probabilités dépendent de l'ennemi et sont décrits dans les fichiers `.edf`. De plus les ennemis peuvent laisser des objets sur le sol lorsqu'ils sont éliminés. De même que les effets, ces objets peuvent être précisés dans la description de l'ennemi.

2.2 Parseur d'ennemis

La classe `EnemyParser` dérive de la classe `RegexParsers` fournie dans la bibiothèque standard de Scala. Cette classe permet de d'analyser la description d'un ennemi.
Une description contient nécessairement le nom d'un ennemi ainsi que son type, elle peut ensuite comporter une séquence de longueur quelconque de modificateurs qui vont faire la particularité d'un ennemi donné par rapport aux autres ennemis du même niveau. Les constantes numériques et les chaînes de caractères sont détectées à l'aide d'expressions régulières et les séquences grâce au combinateur `rep1sep`. Le code relatif au parseur d'ennemis se trouve dans le fichier `EnemyGenerator`. Les fichiers lus par le parseur se trouvent dans `ressources/ennemies` où ils sont classés par difficulté pour vaincre les ennemis décrits.
Voici un exemple de description d'ennemi :

```
1  "Robot" of type melee
2      and loots  laserchainsaw +3 with weight 3
3              or heavyjacket -1 with weight 2
4              or hackingtools with weight 1
5      and reward 20
6      and with defense +2
7      and with strength +2
8      and with level +2
9      and with health +10
10     and looks like "redrobot"
11     and applies burning for 4 turns with weight 1
12         or nothing with weight 4
```

Ligne 1 : Ennemi appelé Robot combattant au corps à corps.
Lignes 2-4 : Objets pouvant être abandonnés. Le premier nombre représente la différence entre le niveau de l'objet et celui de l'étage. Le poids représente la probabilité d'apparition d'un objet par rapport aux autres (l'objet `heavyjacket` a deux fois plus de chance d'apparaître que l'objet `hackingtools`).
Ligne 5 : Argent abandonné lors de l'élimination de l'ennemi.
Lignes 6-9 : Modification des statistiques par rapport aux autres ennemis du même niveau.
Ligne 10 : Cet ennemi sera représenté par le sprite redrobot
Lignes 11-12 : Ajout d'effets lorsque cet ennemi attaque. Un effet de type `nothing` permet de créer une probabilité qu'aucun effet ne soit appliqué lors d'une attaque.

3 Nouvelle méthode de génération de la carte

3.1 Génération des salles

Pour générer des salles, nous utilisons des fichiers avec l'extension .rdf pour room description file. Les fichiers contiennent des presets de salles composés de trois parties séparées par le symbole '%'. La première partie donne des informations sur la taille de la salle ainsi que l'emplacement des sorties de la salle, la seconde décrit la disposition de la salle et la troisième donne des informations sur le contenu de la salle. Seule la troisième partie est analysée par le parseur. Celle-ci donne un sens aux symboles utilisés dans la deuxième partie et décrit les objets ou personnages pouvant apparaître, les interactions, ainsi que la répartition de ces objets entre les différentes positions désignées par le même symbole.

Pour générer une salle, l'objet `RoomGenerator` sélectionne un des presets du fichier, puis utilise un objet de la classe `RoomParser` pour lire le preset et créer un objet de la classe `Room` qui stocke la position de tous les éléments d'une pièce par rapport à l'entrée de la pièce.
Pour donner les informations sur le contenu, nous avons décidé de parser un langage que nous allons décrire à l'aide des outils proposés par Scala. Voici la grammaire utilisée :

```
  program → expr | expr conjunction program
     expr → number specialCharacter element
  element → character | item | "wall" | "lever" | "lockedDoor" | "lock" | "elevator"
character → enemy | "vending machine" | "computer"
    enemy → "easy" | "normal" | "hard" | "boss"
```

De plus, `number` contient l'ensemble des nombres entiers strictement positifs, `specialCharacter` contient toutes les lettres en majuscule et en minuscule et `item` contient l'ensemble des objets, ainsi que "item" qui permet de générer un objet aléatoire.
Chaque expression de la forme n c e du programme ajoute l'élément e sur n cases parmis celles où le symbole c apparaît dans la description de la salle. S'il y a plus de n positions avec un symbole c dans la salle alors on en choisit aléatoirement n sur lesquels on place l'élément e.
Par exemple : Supposons que l'on ait placé 4 fois la lettre d et deux fois la lettre w dans la grille décrivant la disposition de la salle (partie 2 du preset). L'expression `2 d easy and 1 w wall` indique alors qu'il faut choisir 2 d parmi les 4 disponibles et placer deux ennemis faciles à ces emplacements, puis choisir l'un des deux w et y placer un mur. Ce principe peut être utilisé afin de placer une salle secrète qui n'est révélée qu'avec une certaine probabilité (ici une chance sur deux).
Le même type d'expression permet de placer des porte verrouillée, des objets sur le sol ou encore des leviers.

3.2 Génération des niveaux

Afin de pouvoir incorporer les nouvelles salles créées grâce au parser dans les niveaux, nous avons dû recommencer à zéro la génération de niveau. En effet, avec l'ancienne méthode, lorsqu'on plaçait les chemins entre les salles, on ne vérifiait pas que l'on ne recouvrait pas une autre salle.
Désormais, nous générons une première salle de départ et nous gardons en mémoire la liste des sorties non utilisées. A chaque étape, on choisit aléatoirement une sortie non utilisée qu'on retire de la liste et on essaye de placer une salle aléatoire à proximité. Si la salle recouvre une case déjà modifiée précédemment, on ne la place pas et on essaie à la place de relier cette sortie à une autre sortie non utilisée à proximité. Lorsqu'on a atteint le nombre de salles demandé ou que toutes les sorties sont utilisées, on s'arrête. Si le nombre de salles demandé n'est pas atteint, on recommence.
De plus, on place parfois des salles spéciales. On peut placer des salles avec des leviers et, si il y a des leviers non utilisés, des salles au trésor qui contiennent une porte qui sera liée à un levier. Enfin, la dernière salle de l'étage contient un ascensceur pour passer à l'étage d'après. Il y a trois types de salles finales : les salles avec un verrou, les salles avec des portes fermées. Les salles avec un verrou fonctionnent de la même manière que dans la partie précédente. Les portes fermées des salles finales peuvent se dévérouiller de trois manières distinctes : soit en tuant un boss se trouvant dans la salle, soit en activant un levier, soit en tuant tous les ennemis de l'étage.

4 Difficultés rencontrées

Nous n'avons pas rencontré de difficultés importantes durant cette partie. L'attention portée à l'écriture du code lors des deux premières parties nous a permis de l'étendre sans difficulté avec les nouvelles fonctionnalités de cette partie du projet.
Nous avons remarqué que la multiplication des expressions détectées par l'analyseur syntaxique rend plus difficile la recherche d'erreurs. Nous sommes cependant parvenu à corriger ces éventuelles erreurs en nous ramenant à des exemples plus simples afin de tester séparément les différentes fonctionnalités de notre programme.