Projet Programmation 2 : Rapport Partie 1

MANGEL Léo et MICHEL Thomas

1 Etat du projet

Nous avons implémenté les fonctionnalités suivantes :

Affichage :

- Affichage de la carte et du champ de vision du joueur. Seuls les entités dans le champ de vision du joueur sont visibles. De plus, seules les cases de la cartes visibles par le joueur, ou déjà visitées, sont affichées.
- Interface avec les informations sur le personnage du joueur, son inventaire ainsi que les entités dans son champ de vision.
- Affichage des événements récents sous forme de logs, comme les actions du joueur ainsi que les attaques subies.

Jeu :

- Génération aléatoire de la carte, ainsi que des ennemis et des objets sur cette carte.
- Gestion des déplacement du joueur dans 8 directions.
- Système d'objet et d'inventaire. Le joueur peut ramasser les objets au sol pour les mettre dans son inventaire. Il peut consommer, équiper(avec des contraintes sur le nombre d'objets sur une même partie du corps), jetter et déposer les objets de son inventaire.
- Système d'ennemis et de combat. Les ennemis ne bougent pas avant d'être activés, ce qui se produit lorsqu'ils entrent dans le champ de vision du joueur. Lorsqu'ils sont activés, ils se déplacent vers le joueur et l'attaquent au corps à corps. Le joueur peut également attaquer au corps à corps en se déplaçant sur la case d'un ennemi. Il y a également des statistiques qui déterminent les dégats d'une attaque et reçus par l'adversaire. Ces statistiques peuvent être modifiées par les objets.

2 Organisation du code

Le code du projet est organisé en plusieurs parties :

- Affichage : La fenêtre du jeu ainsi que les classes gérant la mécanique du jeu sont instanciées par l'objet `Main`. Nous utilisons Swing pour la gestion de la fenêtre et des entrées du joueur. Nous dessinons directement à l'intérieur de la fenêtre à l'aide des méthodes de la classe `Graphics2D`. Le dessin des différentes parties de l'interface est géré par les objets de notre package *rendering*. La carte est une zone carrée dont la taille s'adapte à la taille de la fenêtre (`BoardRenderer`). Le reste de l'interface se compose d'une barre latérale et d'un affichage des logs sous la carte. Pour afficher les différentes informations nous avons mis en place l'objet `StringRenderer` qui est capable d'afficher les chaînes de caractères sur plusieurs lignes.
- Gestion des entrées (package *input_ handling*) : Les entrées détectées par Swing sont envoyées à une instance de la classe `UI` qui gère l'état de l'interface (objet sélectionné) et qui met à jour l'état du jeu en effectuant l'action demandée par le joueur et en donnant la possibilité aux personnages non-joueurs d'effectuer une action.
- Gestion de la carte : La carte est essentiellement gérée par la classe `GameBoard`. Le terrain est un tableau d'objets héritant de la classe `GameTile` stockant les caractéristiques d'une case(explorées, bloque le joueur). La carte est peuplée par des entités qui peuvent être des objets, des personnages ou d'autres éléments avec lesquels le joueur peut interagir. Un nombre illimité d'objets peuvent coexister sur une même case, mais les autres entitées sont limitées à une par case. La carte est générée procéduralement par un algorithme expliqué dans la suite de ce rapport. La génération d'une nouvelle carte est pour l'instant effectuée au lancement du jeu, mais pourra par la suite être utilisée pour faire plusieurs niveaux. La vue du personnage du joueur est calculée par la classe `FovMap` à chaque mise à jour de la fenêtre de jeu.
- Les entités : Les entités se divisent pour le moment en deux types principaux que sont les objets et les personnages. Un diagramme des relations d'héritage se trouve en annexe.
- Les personnages : Les classes des personnages dérivent toutes de la classe `Character`. Tous les personnages ont des statistiques(points de vie, défense, points d'attaque) et peuvent se déplacer sur la carte, attaquer

et recevoir des dégâts. On peut créer un personnage avec un inventaire à l'aide de certains traits (fichier `CharacterInventories.scala`). Les personnages non-joueurs ont un trait `AIControlled` qu'il faut ensuite associer avec d'autres traits pour définir le comportement final du personnage. Les personnages qui y sont autorisé décrivent leurs actions à l'aide d'une instance de la classe `Logger` associée à la carte. Ces logs sont ensuite affichés à l'écran.

— Les objets : Nous différencions les objets eux même de l'entité qui les représente sur la carte. Les objets sont définis dans le package *items*. Les objets concrets peuvent êtres créés en associant des traits déjà définis qui ajoutent de nouvelles actions possibles aux objets telles que consommer l'objet, le lancer ou encore l'équiper. Certaines associations sont déjà faites de façon à créer des armes, armures ou de la nourriture facilement. Les objets équipables ont également un attribut indiquant la partie du corps sur laquelle ils sont portés. Le nombre maximal d'objets sur chaque partie du corps est défini par des traits de personnage (par exemple `Humanoid` dans `CharacterInventories.scala`). Un diagramme d'héritage pour les objets se trouve également en annexe.

3 Algorithmes utilisés

Nous avons utilisé plusieurs algorithmes existants pour certaines parties du projet.

3.1 Génération aléatoire de la carte

Pour la génération de la carte, nous avons utilisé un algorithme appelé "Tunneling Algorithm". Cela consiste à générer des salles de tailles et de position aléatoire à la suite, de vérifier qu'elles n'intersectent pas avec des salles précédentes, puis de les relier entre elles dans l'ordre de génération à l'aide de couloirs en forme de "L".

Pour la suite, nous envisageons de rajouter d'autres types d'environnement qui utiliseraient des algorithmes différents de génération de carte.

3.2 Champs de vision

Pour le champs de vision, nous avons utilisé l'algorithme dit de "recursive shadowcasting". Cet algorithme consiste traiter chaque octants autour du personnage de manière indépendante. Sur chaque octant, on parcourt chaque colonne (ou chaque ligne en fonction des octants) en partant du joueur vers l'extérieur. Lorsque l'on arrive sur un mur, on calcule les cases qui ne sont pas visibles à cause de ce mur. Pour cela, on calcule les équations des droites qui forment le contour de la zone d'ombre.

Cet algorithme a l'avantage d'être l'un des plus rapides, car il ne considère que les cases visibles et pas plus de deux fois.

De plus, nous avons ajouté la contrainte qu'une case ne contenant pas un mur n'est visible que si le centre de la case est visible. Cela rend l'algorithme symmétrique, dans le sens où une case B est visible depuis une case A si et seulement si la case A est visible depuis la case B si A et B ne sont pas des murs.

3.3 Déplacement des ennemis

Dans la version actuelle, les seuls ennemis implémentés dans le jeu combattent au corps à corps, ainsi le plus adapté est d'utiliser un algorithme de plus court chemin. Nous avons donc décidé d'utiliser l'algorithme A*. Cet algorithme consiste à ajouter une heuristique au choix des arcs à considérer dans l'algorithme de Dijkstra qui prennent en compte la distance euclidienne au point d'arrivée désiré. Cela permet d'être plus rapide même si ça ne renvoie pas nécessairement la meilleure solution.

4 Difficultés rencontrées

La principale source de difficulté a été d'utiliser Scala.

Par exemple, le fait qu'il n'y ai pas de commande `break`, ou au moins qui ne fonctionne pas comme dans d'autres langages de programmation, nous a forcé à adapter notre code à cette contrainte.

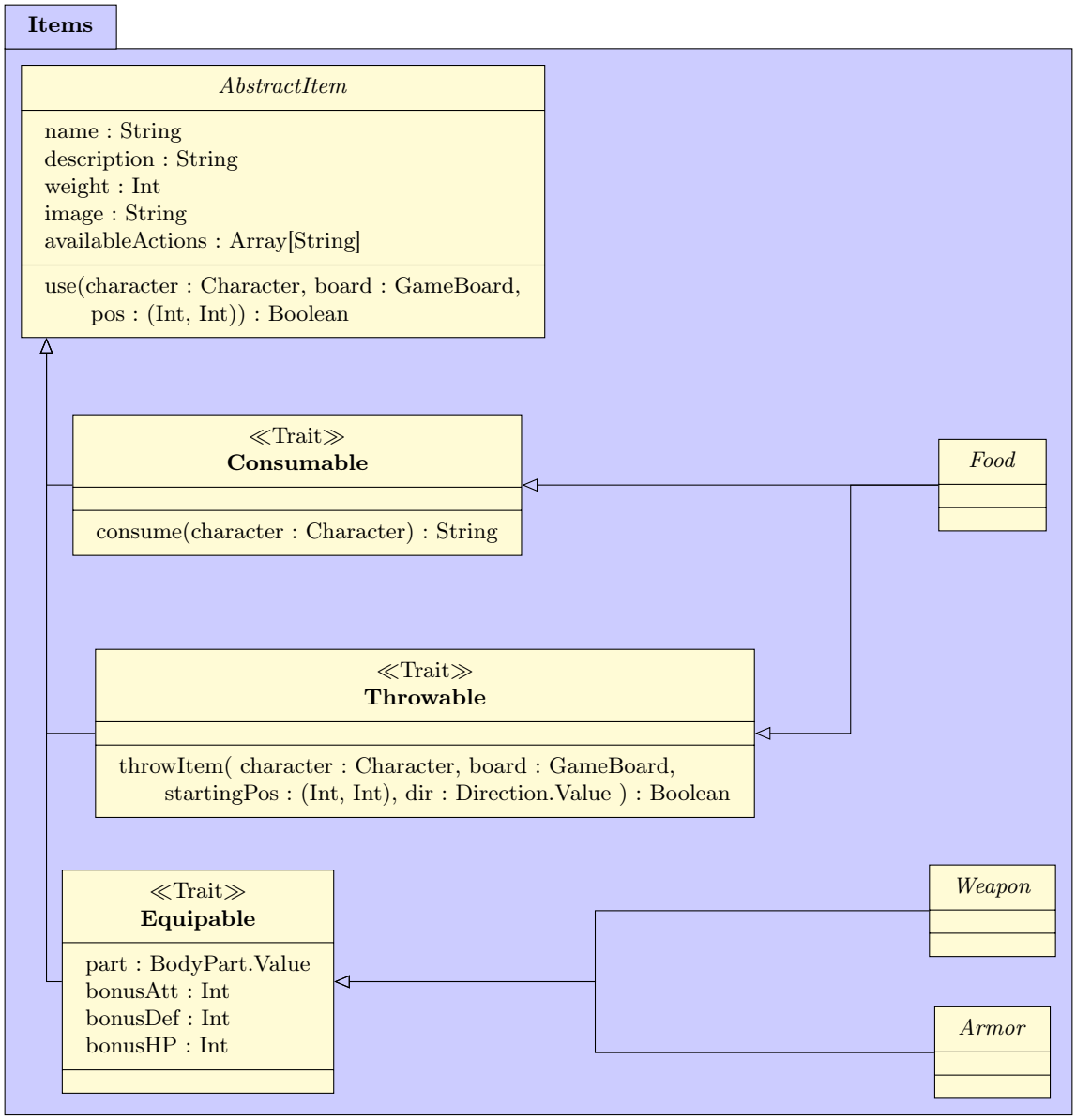
De plus, ce n'était pas toujours évident de trouver de la documentation compréhensible, tout particulièrement pour l'interface que nous avons réalisé avec un package commun avec Java (Graphics). Par exemple, pour afficher une image avec Graphics2D, il existe une demi-douzaine de fonctions qui demande chacune des arguments différents et des classes différentes pour l'image et les différences ne sont que peu documentées.

Cependant, nous avont été surpris de la facilité de prise en main de Scala de manière générale. L'aspect programmation orientée objet de ce début de projet ne nous a pas posé de problèmes. Nous avons trouvé que la notion de traits de Scala était très bien adaptée à la création de nouveaux objets et personnages pour le jeu en donnant la possibilité d'ajouter des fonctionnalités de manière très modulaire.

5 Annexe

5.1 Diagramme d'héritage des objets

La portée n'est pas indiquée car tous les attributs et les méthodes sont publics. J'ai représenté les relations entre les traits et les classes étendues par ceux-ci par des relations d'héritage (bien que les traits ne soient pas des classes).



5.2 Diagramme d'héritage des entités