Projet de TERD

Pr. Rémy Garcia

Antoine Vidal-Mazuy / Juliette Sabatier

Luca Orengo / Raphaël Pietrzak

2020-2021

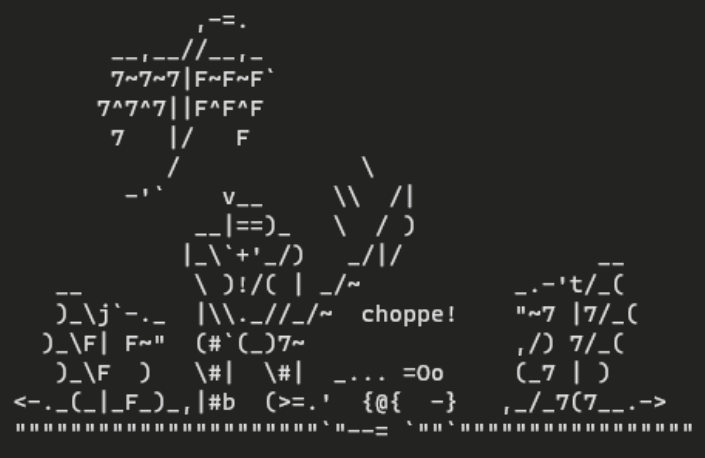

Table des Matières

1. Introduction

1.1. Présentation du jeu

Bienvenue sur Rogue Souls©, un jeu de type Rogue Like que nous avons dû programmer dans le cadre de l'enseignement 'Travail Encadré de Recherche et Développement', dispensé par le professeur Rémy Garcia.

Nous avons essayé de nous approprier le jeu autant que possible. C'est pourquoi le jeu a des airs de ressemblance avec le jeu Binding of Isaac.

Le but du jeu est d'avancer dans le jeu sans mourir. La mort étant permanente, le joueur devra prendre ses précautions et avancer prudemment dans le donjon. Il pourra rencontrer des monstres, des marchands, des pièges et bien d'autres choses.

Afin d'avancer dans sa quête, le joueur pourra se battre pour survivre et espérer une vie meilleure hors de ce donjon… Qui sait ce qu'il attend en haut de la tour… ? N'attendez plus et tentez l'aventure de Rogue Souls, mais prenez garde, la pente jusqu'à la victoire est raide. Atteindre le sommet de cette tour infernale ne sera chose aisée.

1.2. Lancement du jeu

Pour lancer le jeu, utiliser la commande : `java -jar RogueSouls.jar`

Pour plus d'information consulter le README.md.

1.3. Wiki et règles du jeu

Toute la documentation et les règles du jeu sont disponibles ici.

2. Organisation du projet

2.1. Kanban

Nous avons utilisé l'outil Projects de Github pour s'assigner des tâches afin de voir qui travaillait sur quoi. Si on avait la nécessité de poser une question sur un code déjà produit, on pouvait savoir qui avait travaillé dessus.

Au début notre KanBan comptait trois colonnes, TODO, In-progress et Done. Sur les dernières semaines du projet, nous avons ajouté une colonne Bug où nous recensions tous les bugs découverts (en essayant de rajouter le plus d'informations quant au contexte du bug, et nous conservions si possible des captures d'écrans sur le serveur Discord lié à notre projet).

Nous avons découvert bien trop tard une capabilité importante de cette fonctionnalité qui nous avait échappée, et qui aurait pu être vraiment pratique pour la répartition des tâches, qui consiste à pouvoir lier une issue à une pull request, ce qui nous aurait permis de relier plus facilement, surtout sur les cas pointus où nous nous sommes entraidés, les membres du groupe aux différentes tâches.

2.2. Discord

Discord fut un outil plus que vital pour notre projet. Entre la communication, les prises de notes diverses et variées ainsi que les différentes catégories vis-à-vis des besoins d'organisations à court terme (ne nécessitant pas GitHub), l'application bien que relativement basique et commune, nous a été d'une grande aide sans laquelle il aurait été beaucoup plus compliqué de parvenir à nos fins.

2.3. Répartition des tâches

Nous avons commencé par respecter une certaine répartition des tâches sur la première moitié du projet, puis, au fur et mesure que le jeu prenait forme, chacun s'est étendu sur ce qu'il voulait/pensait pouvoir faire. Enfin, une fois que la plupart des éléments étaient mis en place, nous nous sommes réparti les dernières parties majeures de ce qu'il nous restait à faire. Ainsi, au début, la génération a été attribuée à Luca qui fut rejoint par Antoine, la création des entités de bases (non-vivante en premier lieu, et vivante plus tard) a été assignée à Juliette, Raphaël s'est occupé des éléments et outils d'affichages de base, et enfin Antoine s'est chargé de la création des salles en elles-mêmes, qui reprenait les fonctionnalités implémentées par Luca, Juliette et Raphaël.

Une fois les bases créées, chacun s'est occupé d'un peu tout, bien que nous faisions en sorte de modifier des parties sur lesquelles nous avions déjà travaillé.

Enfin, avant la mise en commun "finale" de nos différentes branches, nous avons procédé à une dernière répartition: Luca s'est chargé des classes et du système de niveau/stats associé, Raphaël a amélioré l'IA des monstres en implémentant l'algorithme A* et a mis en place un système prenant en compte de manière plus réaliste de la vitesse des entités mais qui rendait malheureusement le jeu beaucoup moins agréable à jouer et qui a donc été remplacé par un système d'esquive, Juliette a peaufiné la génération et Antoine a nettoyé une partie du code en plus de créer le premier boss avec Raphaël.

Puis, lors des derniers jours nous nous sommes concentrés sur l'équilibrage ainsi que la recherche de bugs et d'améliorations.

2.4. Entraide

Tout au long du projet, et comme dans tout développement de jeu, nous avons fait face à des difficultés qui ont nécessité parfois l'aide d'un autre membre du groupe. Que ça soit dans le but de mettre à profit les compétences des uns ou les connaissances des autres, ou simplement de poser des questions quant à un élément du jeu (par exemple comment utiliser telle ou telle méthode, comprendre pourquoi une classe complexe ne fonctionne pas comme on le pense) qui ne viendrait pas de soi, nous nous sommes entraidés du début à la fin. Par exemple, comme mentionné dans le paragraphe précédent, Luca fut rejoint par Antoine pour la génération, Raphaël et Antoine ont créé les boss ensembles, etc. Il serait trop long de citer tous les cas où nous avons travaillé à plusieurs sur un élément, mais l'entraide fut capitale.

2.5. Réunion

Nous avons effectué deux types de réunions, les premières prenaient place très souvent lors des sessions de TP, pendant lesquelles nous nous réservions la possibilité de poser des questions au professeur, d'effectuer si nécessaire un merge de ce que nous avions codé dans la semaine, et où on s'entraidait si l'un de nous en avait besoin, sur partie qui lui posait un problème.

L'autre type de réunion que nous avons effectué était plus informelle, sur le serveur Discord que nous avions créé pour le projet, durant lesquels nous procédions également à des merges assez réguliers de plusieurs fonctionnalités sur lesquelles nous avions chacun travaillées, à la résolution des conflits qui en découlaient souvent et au travail en groupe quand l'un de nous le demandait pour debug ce sur quoi on travaillait, ou pour s'entraider dans le but de coder une fonctionnalité qui nous résistait.

2.6. Conception générale

Une partie conséquente de notre conception a été faite sur un tableau Miro. Afin de simplifier l'idée globale du jeu, nous l'avons "divisée" en plusieurs groupements globaux de classes qui correspondent plus ou moins aux différents packages du projet. Nous sommes partis des éléments basiques dont le jeu aurait besoin : de quoi faire un bon affichage, les différents éléments qui composeront le jeu et sa logique ainsi que les aspects qui leur sont associés comme les items pour le joueur et les monstres, les combats, ...

Cependant nous avons partiellement fait notre conception à l'oral lors de nos réunions, notamment lorsqu'il fallait décider de comment s'y prendre pour mettre en place des fonctionnalités plus précises.

3. Structure du jeu

3.1. Structure globale

Le principal axe de conception tourne autour du GameLoop et du GameState. La GameLoop s'occupe de faire le lien entre le GameState et le joueur, tandis que le GameState fait le lien entre tous les éléments du jeu et la GameLoop. La plupart des actions nécessitent de passer par le GameState. En effet celui-ci contient toutes les informations du jeu, (Joueur, Map, Monstres, etc…).

Passer par le GameState offre un panel d'actions étendu sur le jeu. Si un monstres a accès au GameState, il pourra techniquement tout faire. L'inconvénient est que si une partie du code n'a pas accès au gamestate, sa capacité d'action sera très limitée.

3.2. Patron de conception

3.2.1. Factory

La Factory est un patron de conception que nous avons beaucoup utilisé, elle permet de créer une interface commune pour instancier un objet. Cette interface rend la création d'un objet moins complexe, par exemple sans passer par la factory, pour créer un monstre, il faudrait donner sa position, son type, son nom et sa stratégie. Or via la Factory il suffit de donner une position et un type uniquement, la Factory gère derrière l'attribution du nom et de la stratégie.

Au vu du nombre conséquent d'objets que nous devions créer, nous avons utilisé beaucoup de Factory, de la création des salles à l'attribution des items pour un marchand.

3.2.2. GameLoop

La GameLoop représente l'interface Homme-Machine de notre jeu. Elle récupère les entrées clavier du joueur, et se charge de passer les informations au reste du jeu. C'est elle qui décidera de l'affichage en fonction de l'état du jeu. Cela permet au joueur d'interagir à n'importe quel moment avec son environnement. Le GameLoop est indispensable à l'exécution du jeu. Sans lui, le jeu ne pourrait pas boucler sur lui-même, jusqu'à ce qu'il soit interrompu.

3.2.3. State

Le GameState contient l'état du jeu. L'énumération State contient tous les états possibles : FIGHT, NORMAL, PAUSE_MENU, SHOP_MENU, WIN, LOSE, END. Nous avons décidé qu'en

plus de contenir l'état du jeu, la classe GameState contiendrait tous les éléments du jeu. Comme l'état influence le comportement de quasiment tout le jeu, autant que celui-ci contienne toutes les informations.

Comme abordé dans la structure globale, le GameState est le pivot du jeu. Sans lui, il est difficile d'avoir une influence concrète sur le jeu. Cela a posé quelques problèmes de conception par moment. En effet, si une classe est éloignée de la portée du GameState, son action est alors plus que réduite. Pour remédier à ce problème nous avons dû par moment passer le GameState à des classes ou des méthodes, uniquement pour propager la portée du GameState.

3.2.4. **Strategy**

Le patron de conception Strategy a été utilisé pour donner un comportement différent aux monstres. Un objet de la classe Strategy comporte une condition et une stratégie suivante.

Si la condition est respectée, le monstre utilisera la stratégie courante, sinon il passera à la stratégie suivante où il regardera si sa condition est respectée ou non.

La seule stratégie qui n'a pas de condition est la stratégie d'attaque car elle est associée à la stratégie d'approche qui, elle, vérifie la distance entre le joueur et le monstre, et passe à la stratégie d'attaque uniquement si la distance est bonne.

Une condition est faite en utilisant de la programmation fonctionnelle, à partir de deux objets donnés, un joueur et un monstre, on crée une condition. Par exemple, comparer la distance entre le monstre et le joueur, ou bien regarder le niveau de vie du monstre.

Ce système permet aussi de gérer le système de portée de vue des monstres en mesurant la distance du monstre au joueur. Si celle-ci est trop grande, le monstre ne verra pas le joueur et ne cherchera pas à l'attaquer. Il restera en stratégie IDLE et bougera aléatoirement.

3.2.5. **Builder**

Ce pattern est utilisé pour construire les noms et la description de chacun de nos équipements, plus précisément par la méthode buildDescriptionAndName de notre equipementFactory.

3.3. GridMap

La GridMap est générée à partir d'une salle. Elle contient un tableau de toutes les Tiles composant la salle ainsi que les entités présentes dans celle-ci. C'est elle qui nous permet de décomposer la salle et de l'afficher plus facilement.

Elle contient aussi une liste de chaînes de caractères qui facilite la création du string global de l'affichage. Cette méthode permet de mettre à jour cette liste pour plus de facilité.

La GridMap contient aussi une liste de position qui correspond à la range du joueur (zone dans laquelle il peut faire des dégâts aux monstres).

De plus, cette classe contient des méthodes permettant de gérer l'affichage et le choix de l'entité à afficher si plusieurs entités sont sur la même position (par exemple lorsqu'un un monstre est sur une pièce).

Plus précisément, les Tiles permettent de créer la structure de la salle et préciser les endroits accessibles au joueur en posant les murs, le sol... Ensuite les entités posées dans la salle interagissent avec le joueur comme précisé ci-dessous.

3.4. Entité

Le jeu se compose de différentes entités, divisées en deux catégories majeures : les entités vivantes et les entités non vivantes.

Les entités vivantes sont composées du joueur, des monstres et des marchands. Elles peuvent avoir un inventaire associé et suivent un comportement défini en fonction de la situation. Elles peuvent affecter le jeu de plusieurs manières, que ce soit en bougeant, en mourant ou en attaquant.

Les entités non-vivantes quant à elles, représentent tout ce avec quoi le joueur peut interagir et qui n'évolue pas en fonction de l'état du jeu :

- Les pièces (appelées BTC dans le jeu) peuvent être ramassées dans les salles ou sur les monstres lors des aventures du joueur, et être utilisées avec les marchands.
- Les coffres contiennent des pièces d'équipements aléatoires adaptés au niveau du joueur, ainsi que des objets pouvant lui faciliter l'aventure. Il existe des coffres standards et des coffres en or nécessitant une clé pour être ouvert
- Les portes et les escaliers permettent de naviguer et de progresser dans le donjon. Les portes menant au boss et à l'escalier demandent de remplir certaines conditions pour être ouvertes. L'escalier ne permet que de monter d'un étage, pas de revenir en arrière.
- Les boutons sont la condition pour ouvrir la porte menant au boss. Ils sont éparpillés un peu partout dans les salles de l'étage, et il faut tous les activer pour pouvoir ouvrir ladite porte.

- Les potions sont des objets classiques qui peuvent être récupérés un peu partout. Elles sont un moyen d'aider le joueur dans sa progression.
- Les tombes remplacent les monstres lorsqu'ils meurent. Le monstre tué n'importe pas dans les objets qu'elles contiennent, sauf pour le boss qui donne des objets plus rares et la clé permettant d'accéder à l'étage suivant.
- Les trous et les piques sont des pièges pour le joueur. Les monstres n'iront jamais dessus, ils ne sont pas fous, mais vous pouvez utiliser cette caractéristique à votre avantage. Les trous vous ramènent au début de l'étage, et les piques vous enlèvent quelques PV.
- Les carottes sont des pièges éphémères : elles sont posées par le boss d'étage mais peuvent être détruites ou enlevées par le joueur. Elles font beaucoup de dégâts et disparaissent une fois que le joueur a marché dessus.
- Enfin, les pièges du Ranger sont des pièges que les monstres ne voient pas et qui font plus ou moins de dégâts suivant la chance du joueur. Ce dernier peut également se blesser avec donc manipulez-les avec précautions. Ils peuvent également être ramassés par le joueur.

3.5. Stuff

Le stuff représente deux types d'équipements. Les items, qui sont à usage unique, ainsi que les équipements, qui sont portables par le joueur.

Les items:

- Potion de vie, redonne au joueur une quantité de points de vie proportionnelle à son niveau.
- Elixir, redonne au joueur une quantité de points de mana proportionnelle à son niveau.
- Potion d'expérience, donne au joueur de l'expérience.
- Map de l'étage permet de découvrir la map de l'étage sans s'y balader.
- Clé dorée, permet d'ouvrir les coffres dorés.
- Clé de l'étage, permet d'ouvrir la salle des escaliers.

Les équipements se déclinent en partie du corps. En effet le joueur ne peut équiper qu'une seule arme, une seule armure, un seul casque etc …

En tout, il y a 7 types d'équipement différents : les armes, les objets secondaires (cape, bouclier…), les armures, les casques, les pantalons, les gants et les chaussures. Chaque classe possède des équipements différents (bâtons, épées, arcs) etc.

Les équipements, en plus d'êtres différents selon la classe, ont une rareté, allant de E (commun) à L (Légendaire) comme suit : E, D, C, B, A, S, L. Chacune de ces raretés change les caractéristiques des équipements. La rareté est implémentée de sorte qu'une épée de niveau 1 L soit aussi bonne qu'une épée niveau 6 E.

3.6. GameRule

Nous avons créé une classe static GameRule qui contient des fonctions permettant de modifier les valeurs des paramètres du jeu facilement. Par exemple, le nombre d'équipement dans chaque coffre, le nombre de monstres dans les salles…

Ainsi, si on veut rajouter un nouveau type de potion, il nous suffit de créer un nouvel objet potion et tout simplement modifier les fonctions qui choisissent le type de potion à donner. Pas besoin de chercher dans le code toutes les fonctions contenant un choix de potion.

3.7. Combat

Les combats font partie intégrante du jeu. Ils se déroulent en tour par tour selon le 'Turn order'. Les entités sont classées de la plus agile à la moins agile. Une action d'un monstre est soit une attaque soit un mouvement. Le joueur a, lui aussi, le droit à une attaque ou un mouvement, mais il peut aussi utiliser un objet, ouvrir un coffre ou une tombe. Il a la capacité de se tourner sans consommer son tour, à fin par exemple, de pouvoir toucher une entité qui serait derrière lui.

Chaque monstre tué rapportera au joueur une quantité d'expérience proportionnelle au niveau du monstre.

3.8. Classes et sorts

Pour cette fonctionnalité, nous avons essayé de construire un système aussi extensible que possible. Pour le moment nous avons trois classes Ranger, Warrior et Mage, une classe dans notre implémentation se compose de deux choses :

- Premièrement une liste d'entiers qui, dans la classe PlayerStats, sert à définir ce que chaque classe gagnera par niveau, et qui est utilisée par la méthode levelUp dans cette même classe. Par exemple, un guerrier gagne plus de PV par niveau qu'un ranger qui lui-même en gagne plus qu'un Mage, l'inverse est vrai pour les MP.

- Deuxièmement une classe, que nous avions décidé d'appeler path (suivi du nom de la classe) qui sert à définir quel sort chaque classe apprend à quel niveau. Ensuite à l'intérieur de PlayerStats, nous avons deux méthodes : getRewardForLevelForClass qui extrait du path correspondant à la classe du player le sort lié au niveau où la méthode est appelée, par réflexion, pour ensuite effectuer un addSpell dans la méthode levelUp qui ajoute un sort au joueur s'il existe un sort pour ce nouveau niveau. L'avantage d'utiliser la réflexion fut que pour ajouter un sort on avait simplement à s'assurer que le nom précisé dans le path de la classe soit orthographié de l'exacte même façon que la classe du nouveau Spell.

Cette implémentation a été pensée pour simplement avoir à écrire une autre liste d'entier, et à copier la structure d'une classe path et d'y effectuer des modifications légères afin d'ajouter une ou plusieurs autre(s) classe(s). Mais pour cela nous devions aussi avoir des sorts à faire apprendre à nos différentes classes.

Nos sorts étendent tous la classe AbtractSpell, et sont construits en appelant son constructeur avec leur nom, un booléen signifiant si ce sont des sorts effectuant de dégâts ou non, la portée sur laquelle ils peuvent agir, le niveau du sort, et s'ils possèdent un effet spécial en plus ou non.

Toutes ces informations sont ensuite lues par la méthode UseSpell dans AbstractSpell pour effectuer ce qui est nécessaire dans le GameState afin d'appliquer l'effet dudit spell.

4. Génération et IA

4.1. Génération du donjon

La génération du donjon se fait par étage, à chaque étage que nous devons générer dont le premier, nous générons de façon aléatoire et procédurale une HashMap liant entre elles le numéro de la salle dans l'étage et les une liste d'int contenant les voisins qui lui sont liés ainsi que sa position relative, nous définissons aussi leur type, ce qui se fait dans la classe GraphDungeon.

La méthode calculDirection tire la direction dans laquelle continuer à placer des salles à chaque fois pour la méthode drawPath. Quant aux vertices du graph elles sont créées à chaque étape par la méthode createVertice, dans le même temps est aussi gérée la création de la salle de départ et des salles de fin. Nous utilisons aussi la méthode attributeType pour générer une Liste contenant la quantité de room de chaque type.

Ensuite, la Hashmap et la Liste des types généré par GraphDungeon sont utilisés dans le constructor de DungeonStructure pour générer la roomList qui contient toutes les pièces avec leur type.

4.2. Génération des salles

La création des salles se fait grâce à la classe RoomFactory qui comme son nom l'indique est une factory.

Cette factory contient les informations de base pour créer une room mais aussi une liste currentAvailablePosition qui nous permet d'avoir les positions ou l'on peut poser une entité. Cette liste ne contient donc pas que les positions des murs mais aussi les positions devant les portes car cela bloquerait le joueur ou ferait planter le jeu.

Tout d'abord elle créer la base de la room : le sol et les murs. Ensuite suivant le type de salle que l'on veut créer on ajoute les entités nécessaires (monstres, pièces, potion, coffres…). Et enfin, elle ajoute des petits pièges (pics et trous). Pour ajouter comme il faut et suivant la GameRule les entités, des fonctions static ont été créées.

Certains éléments sont ajoutés après la création des salles par la factory, car ceux-ci dépendent d'informations que la factory n'a pas, par exemple les marchands qui ont besoin du niveau du donjon, des portes qui doivent être reliés entre elles avec le graphe du donjon.

Une fois toutes les entités posées, une vérification s'impose. Pour chaque salle du donjon, on utilise une fonction appelée verificationRoom. Celle-ci crée un graphe avec les positions de la salle et lance l'algorithme de Dijkstra pour voir s'il y a bien en chemin entre toutes les entités présentes dans la salle. Si une entité n'est pas reliée aux autres, un piège est enlevé. On relance Dijkstra jusqu'à ce que tout soit relié.

4.3. IA des monstres

Les monstres suivent une logique basique, bien que certains d'entre eux aient un comportement légèrement différent. Leurs actions dépendent des stratégies qu'ils suivent.

De base, un monstre bougera de manière aléatoire jusqu'à ce que le joueur entre dans un rayon défini autour du monstre (un peu comme son champ de vision). Ce dernier remarquera alors la présence du joueur et s'approchera de lui en fonction du chemin fourni par l'algorithme A*, qui est recalculé à chaque fois que le monstre doit se déplacer.

Cet algorithme nous garantit que le monstre saura atteindre le joueur pour peu qu'un chemin existe. Il y a évidemment le cas où le seul chemin disponible soit bloqué par un autre monstre, auquel cas les autres ne bougeront pas. C'est un comportement involontaire de base mais que

nous avons choisi de laisser afin de ne pas ajouter trop de difficulté et de se retrouver dans une situation où les monstres s'enchaînent, rendant certains combats trop difficiles.

Comme mentionné plus haut, certains monstres ont des comportements différents. Par exemple, le Gobelin cherchera à s'éloigner le plus possible du joueur si sa vie descend en dessous des 50%, et il récupère quelques points de vie à chaque tour afin d'éviter des courses poursuites sans fins. Le Vampire a un comportement différent à la mort : au lieu de laisser une tombe comme les autres, il fait apparaître une chauve-souris. Enfin, le sorcier est capable d'attaquer à distance contrairement aux autres monstres.

5. Features

5.1. Musique

Pour rendre le jeu un peu plus vivant, nous avons ajouté quelques effets sonores ainsi que de la musique. Il y a deux musiques, une musique de combat et une musique hors combat. Les musiques se lancent toute seule dans un combat et en sortant d'un combat.

On a rajouté quelques effets, comme par exemple, l'ouverture d'une porte, la mort d'un monstre, les pièces, quand on marche sur des piques etc.

5.2. Enigmes

Pour forcer le joueur à découvrir l'étage, la porte permettant d'accéder au boss est bloquée par des interrupteurs répartis dans le donjon. Le joueur devra les trouver et les actionner.

5.3. Marchand

Le marchand est important, il permet au joueur de se débarrasser des équipements dont il n'a pas besoin, mais aussi et surtout, de s'acheter des objets. L'inventaire du marchand contiendra un minimum d'équipements d'une rareté commune. Il aura en plus quelques équipements de meilleure rareté. Le niveau des objets à vendre s'adapte avec celui du joueur.

6. Bugs et limites

6.1. Vérification de la validité des salles

Nous avons eu un bug, il est arrivé que la fonction qui vérifie la validité (accessibilité de toute les entités importantes) d'une salle boucle à l'infini. Nous avons trouvé que cela arrive quand une

entité est entourée d'entités autres que des pièges, donc à ne pas enlever. Ainsi même en enlevant tous les pièges de la salle une entité reste inaccessible et la fonction boucle. Dans ce cas de figure, on arrête la fonction après un certain nombre de boucles et on recréer une salle du même type pour la remplacer.

6.2. Superposition inhabituelle d'entité

Durant la programmation de notre jeu, nous avons rencontré plusieurs bugs marquants, le meilleur exemple qui me vient à l'esprit fut un bug de superposition. Quelques fois, nous avions plusieurs entités (souvent 3) qui s'empilent les unes sur les autres. Finalement, nous sommes parvenus à trouver une ligne dans la méthode move qui nous semblait pouvoir causer le problème, et avons utilisé la condition can move à la place, ce qui semble avoir résolu le bug.

6.3. Range

Nous avons aussi manqué de temps pour utiliser la statistique range du joueur en combinaisons avec la range des sorts.

6.4. Superposition de salles dans le donjon

Nous avons aussi remarqué que quelques fois la salle du boss se superposait avec une autre salle. Cela ne dérange pas le déroulement du jeu, on peut se rendre dans la salle du boss et celle qui est en dessous, le problème est plus visuel sur la map. Ce bug provient du fait que l'on créer la salle du boss en dernier et qu'il manque une vérification sur la position de la salle. Au vu de la difficulté de corriger ce bug et du temps qu'il nous restait (nous avons remarqué ce bug que très tard dans l'avancement du jeu), et puisqu'il était surtout visuel, nous avons choisi de l'ignorer pour l'instant.

7. Pour aller plus loin

7.1. Arbre de compétences

Pour le moment, le joueur est récompensé d'un sort, en fonction de son niveau.

L'arbre de compétence, serait un arbre contenant différents sorts avec des paliers à débloquer. Ainsi le joueur aurait pu choisir des sorts ou des améliorations grâce à des points de compétences, obtenus en montant de niveau.

Une autre idée plus simple aurait été de faire choisir au joueur entre 2 propositions à la montée de niveau.

7.2. Enigmes

Les salles au trésor sont pour le moment uniquement des salles contenant un coffre doré ainsi que des pièces et des potions. Nous aurions voulu mettre des énigmes dans ces salles.

Par exemple, quand le joueur entre dans la salle au trésor, les portes se ferment. Dès lors, le joueur doit résoudre une énigme affichée dans le descripteur. Comme appuyer dans le bon ordre sur des boutons pour débloquer les portes.

7.3. Etages à thème

Dans le jeu, les étages se ressemblent tous, pourtant dès le départ nous avions prévu de faire des étages à thèmes. Par exemple, un étage avec des pièges de laves, des couleurs plus proche du rouge, ou encore un étage avec de l'eau, etc. L'idée aurait été de faire des biomes, avec un écosystème différent (des monstres différents, des boss différents). Le joueur aurait pu obtenir des items uniques à chaque biome. Cependant l'idée a été abandonnée par manque de temps, au profit d'autres fonctionnalités comme l'intuitivité des menus, les combats ou les boss.

7.4. Des monstres uniques

Pour le moment les monstres suivent la même logique à quelques exceptions près. Nous avons pour ambition d'en ajouter/modifier afin d'apporter une meilleure diversité dans les combats pour le joueur, et que les monstres se distinguent les uns des autres en étant plus que simplement "des monstres". Il en va de même pour les boss, il n'en existe qu'un pour le moment et il est relativement basique mais nous comptons bien en ajouter et les améliorer.

7.5. Multijoueur

Même si c'est une perspective sur le plus long terme, nous avons théorisé plusieurs façons pour intégrer du multijoueur dans notre jeu, premièrement un système de scoring ou les différents joueurs partageraient un simple scoreboard mis à jour par des requêtes à un serveur commun. L'autre idée que nous avions eue, aurait été de les faire évoluer dans le même donjon, une grande majorité de l'architecture de notre jeu aurait pu être déplacée côté serveur, le client ne servant qu'à récupérer les inputs et les envoyer au serveur.

8. Ressenti personnel

8.1. Antoine

J'ai appris que le travail en groupe nécessite une communication parfaite entre les membres. Des moments d'incompréhension peuvent entraîner des pertes de temps. Je pense notamment à une fonctionnalité mal réfléchie, qui a été codée pour être finalement abandonnée, car elle nuisait au jeu.

Ce projet a été enrichissant sur plusieurs points. L'importance de bien documenter son code, pour qu'il soit compréhensible par tout le monde, l'importance de rendre son code clair, pour la même raison. J'ai aussi appris à mieux utiliser les outils comme Git ou github. L'utilisation du KanBan m'a aidé à mieux concevoir la répartition des tâches.

8.2. Luca

Durant ce projet j'ai appris l'importance d'être clair, que ce soit au niveau de la communication dans l'équipe, qu'elle soit vocale ou textuelle, pour définir sur quoi on aller travailler, quand on allait se retrouver, le scope des fonctionnalités sur lesquelles on était en train de travailler, pour éviter des cafouillages.

Mais aussi d'être clair au niveau de mon code, que ce soit au niveau des noms de variables/méthodes, en essayant continuellement de garder mes méthodes le plus simple possible, pour simplifier la compréhension par les autres membres de mon équipe ainsi que pour faciliter l'itération par moi plus tard ou par d'autre, sur les fonctionnalités que j'ai codées.

Ce projet m'a en général appris beaucoup, que ce soit sur la programmation, le versionning, le travail de groupe en général, mais aussi sur moi-même comment je fonctionne dans un groupe, sous pression, ce genre de choses.

8.3. Juliette

Ce projet m'a permis d'apprendre beaucoup de choses sur l'organisation et la communication dans un groupe mais aussi la nécessité de créer de bonnes bases. De bonnes bases pour le projet en lui-même afin de pouvoir avancer sur un projet sereinement et que les membres du groupe puissent sans difficulté utiliser mon code ou bien le modifier. La documentation est elle aussi importante pour ce point.

Étant un de nos premiers gros projets, je n'ai forcément pas été parfaite au niveau organisation mais j'ai pu me rendre compte de mes lacunes. Pour un prochain projet je vais, par exemple m'obliger à plus commenter mon code.

8.4. Raphaël

Ce projet m'a permis de réaliser à quel point il pouvait être compliqué de créer un jeu, aussi simple soit-il. La cohésion d'équipe, la bonne entente et une ambiance saine en général me sont apparus comme des éléments essentiels à la réussite de ce travail.

J'ai gagné énormément d'expérience grâce à ce projet, que ça soit en Java, ou de manière globale. J'ai appris à utiliser GitHub, IntelliJ, Maven, et différentes librairies. Il m'est également beaucoup plus facile d'anticiper les comportements non-voulus de mes codes et de me projeter dans le futur afin de préparer ces codes à de futures améliorations. Je suis aussi plus capable de prédire si oui ou non je serais en mesure de mettre en place une fonctionnalité rapidement.

9. Conclusion

Pour conclure, ce projet a été enrichissant sur plusieurs points : en programmation orientée objet, en conception, en gestion de projet, en travail d'équipe. Nous avons pu voir à travers le projet, différents patrons de conception qui nous ont permis de programmer de façon claire nos objectifs. Nous avons pu découvrir les différents obstacles auxquels les développeurs peuvent faire face. C'est avec fierté que nous proposons le résultat de ce projet : ROGUE SOULS©.