Rogue Like

LICENCE INFORMATIQUE

UNIVERSITÉ CÔTE D'AZUR

MASCARO Valentin
JULIEN Raphael
JUNIN Thibault
LOPES Allan

Table des matières

Ce présent document est le rapport du projet de TERD que nous devions réaliser dans le cadre du cours de TERD en Licence 3.
Le but du projet était de réaliser un Rogue Like en Java.
Pour ce projet, nous avons décidé de faire un Rogue Like se déroulant en extérieur, avec des combats dans le style de Pokémon.

Comment nous nous sommes organisé

L'organisation a été un point primordial pour mener à bien ce projet. Pour cela, nous avions décidé d'utiliser l'outil Miro, qui est un tableau blanc en ligne collaboratif, il a été principalement utilisé lors des séances de TP que nous avons dédié à la conception de notre Rogue Like. Nous avons également utilisé Discord afin de communiquer en dehors des séances de TP et pour nous faire un point sur l'avancée du projet.
Ensuite chacun effectuait son travail de la semaine sur une branche ce qui permettez à chacun de suivre l'avancement de ses collègues ou de les aider, puis on faisait un merge au fur à mesure selon les besoins.

Les fonctionnalités prévu

Seed

Nous avons décidé, pour la génération de notre map et de tous les autres éléments du jeu, d'utiliser une seed, qui serait dans notre cas une chaîne de caractères en hexadécimal, originalement d'une longueur de 12 caractères, par la suite, cette seed a été augmenté en raison du fait qu'elle était trop petite pour nous donner beaucoup de différence. Thibault qui s'est chargé de la réalisation de cette classe, a mis en place une fonction permettant de récupérer le nombre à une certaine position de cette seed, tout en permettant de donner n'importe quelle position au préalable même si la seed était plus petite, sur le principe d'une boucle (le caractère après le dernier caractère de la seed, est le premier de cette même seed).

Generator

Nous devions générer un étage composé de plusieurs "salles", pour ce faire, Thibault à créer la classe Generator qui permet avec en entré la seed, et la définition des tailles de map que nous voulions de générer les différentes salles avec la position de leurs chemin de sortie.

Colors

Enfin, Thibault a réalisé une petite classe permettant de gérer l'affichage des couleurs en fonction du symbole présent sur la Map.

Etage

La classe Etage comprend un tableau de maps qui ensemble forme un niveau. Elle permet notamment de gérer la map où se situe le joueur et son affichage, mais aussi l'affichage de la carte du niveau ainsi que le déplacement des Props dans le niveau. Il permet aussi de spawn les monstres et les objets.

GameController

La classe GameController gère la boucle du jeu et le déroulement de ce dernier, ainsi que les entrées d'utilisateur. Il a un ensemble de fonction qui correspond chacun à des entrées spécifiques. Elle comprend plusieurs états avec dans ces derniers les actions (fonctions) possible par le joueur.

Props, Monstres et Player

Les classes Props/Monstres/Player sont les personnages du jeu. Chaque personnage possède des PV, de l'XP, une arme, etc.

Les classes Monster/AbstractMonster permettent de faire une Factory, et ainsi créer des monstres différents facilement. Elle comprend par exemple une fonction act() qui gère leur déplacement et donc le comportement des monstres. Par exemple un cerf va fuir, tandis qu'un OrcWarrior devrait se diriger vers le joueur.

Items

Les classes Items sont comme le nom l'indique les « objets » que peut posséder le joueur. Il comprend trois sous-catégories qui est les Armes, les Compétences et les Consommables. Les armes et les compétences devaient être à l'avenir différentes, mais ça n'a malheureusement pas été implanté.

Inventaire

La classe Inventaire permet de gérer les Items du joueur. Elle comprend les 3 catégories : Armes, Compétence et Consommable.

Le joueur peut ainsi équiper ou améliorer une arme ou une compétence, ou bien il peut utiliser un consommable.

Combat

La classe Combat se lance quand le joueur se situe à côté d'un monstre. C'est un combat au tour par tour très inspiré de jeu comme Pokémon.
Le joueur possède deux attaques principales : Compétence et Arme.
Il peut aussi fuir ou naviguer dans son inventaire pour équiper/améliorer une arme ou compétence, ou se régénérer des PV. Si le joueur fuit, le monstre sera freeze pendant un tour pour laisser le temps au joueur de partir.

Map : Fait par Mascaro Valentin

L'objectif de la map est de fournir un espace sur lequel le joueur peut se déplacer, chercher un chemin, combattre des monstres. Chaque map doit être différente, que l'on parle de sa taille, ou de son contenu, enfin, chaque map doit avoir une sortie accessible.

Le premier problème parmi ses objectifs est l'hétérogénéité de chaque map, en effet nous possédons une Seed en argument du constructeur de la map, toutefois une même Seed donnera toujours une même map. Il faut donc un moyen de départager les maps malgré cette même Seed.

Le deuxième problème vient de la taille, si chaque map doit avoir une sortie, relié à la map d'à côté, comment faire si la sortie d'une map est a une hauteur supérieur à la hauteur maximum de la map suivante.

Le troisième problème vient de la nécessité d'avoir une sortie accessible, en effet certaines cases seront inaccessible dû à nos différents éléments sur la map (arbre/eau/herbe/mur), il nous faut donc un moyen de nous assurer qu'il existe bien un chemin qui ne soit pas bloqué par des cases inaccessible, si cela venait à arriver, créé un chemin.

Premièrement, chaque Map demande dans son constructeur la Seed généré dans Generator, ainsi qu'un indice, une valeur allant de 1 jusqu'au nombre de map souhaité, cette identifiant va permettre de régénérer une Seed à partir du constructeur de Seed (Seed, indice) de la classe Seed, ce constructeur appel une méthode generateSeed qui retourne une nouvelle Seed dont chaque valeurs est modifié, en fonction de la Seed donné en argument. Ainsi chaque map possède bien une Seed différentes avec la même parenté (la Seed originel).

Cette Seed permettant ensuite de déterminer la tailles de la map (modifié par une taille de référence donné dans le constructeur pour éviter d'avoir des maps de tailles trop petites ou trop grandes), ainsi que la valeur de chacune des cases de la map (représenté

par un tableau de caractères), la décision pour la valeur de chacune des cases se fait dans la classe DecisionCase expliqué plus bas.

Deuxièmement, maintenant que chaque map à une taille différente il faut s'assurer que le chemin reliant les maps soit logique.

Admettons qu'une map soit haute de 20 cases, que sa sortie vers la map suivante soit située à la hauteur 19.

Admettons que la map suivante soit haute de 5 cases. Où se trouve son entrée ?

Il faudrait que son entrée soit tout en bas, mais la transition ne serait pas logique, le joueur se retrouverai téléporté 15 cases vers le haut. Nous avons donc décidé de décaler toutes nos maps de 2 cases, c'est à dire que sur un tableau de 20x20, la map se trouve dans un tableau interne de 18x18, les 2 cases libres de chaque côté étant nécessaire pour créer des couloirs de déplacements.

(La taille de la map varie, mais pas la taille du tableau qui la contient, c'est à dire que si le tableau est d'une taille fixe de 50, une map pourra être au maximum de taille 48, si la map est de taille inférieur à 48, les couloirs traversera donc tous les cases entre 50 et la map)

Dans la situation précédente, le joueur arrive donc sur la map suivante dans un couloir situé en hauteur 19, 14 cases en dessous de la map, et il lui faut maintenant se déplacé le long du couloir pour rejoindre la map.

La création des couloirs se fait à deux étapes différentes, tout d'abord les couloirs de sortie, admettront que notre tableau est de taille 50. Notre map fait 20x20 (elle pourrait aussi bien être de taille k x y, k et y quelconque), la sortie vers la map suivante est toujours à la hauteur 19 vers la droite, il reste donc 30cases a traversé pour atteindre le bout du tableau de taille 50, au moment de la création de la map on va donc créé un couloirs allant de la fin de la map, jusqu'à la fin du tableau, le joueur changera de map en arrivant au bout de ce couloirs.

Maintenant viens la génération du couloir d'arrivé dans la nouvelle map, ce couloirs se fait après avoir généré toute les maps d'un Etage. Avec la méthode CreationCheminDepuisExte, cette méthode prend en argument la position d'arrivé du joueur sur la map actuelle, depuis la sortie de la map précédente.

Toujours dans l'exemple précédent, le joueur est sortie de sa map de taille 20, en prenant un couloir à la hauteur 19, vers la droite. Il arrive donc sur la map suivante de taille 5, en étant à la position gauche 0, et hauteur 19 (on rappelle que le tableau contenant la map est bien de taille 50, donc suffisant), il faut donc créé un couloirs depuis la position 19.0 , vers la map (qu'importe la taille de la map, la coordonné de son

extrémité haut gauche sera toujours 2,2 , puis elle s'étend selon sa taille). Ainsi la méthode va créer un couloir pour rejoindre l'entrée en 19.0, et un bord de la maps situé entre 2,2 et 7,7 (puisque de taille 5x5). La méthode déterminera en fonctions de l'angle d'arrivé, si il faut faire des détours, ou simplement un couloirs parfaitement droit, dans ce test là il faudra remonté de 14cases, puis tourner à droite de 2 cases, et enfin percé les murs entourant la map, puis pour s'assurer que le couloirs débouche sur une case accessible, la méthode détruira les 2 premières cases devant la sortie de son couloirs, et les remplacera par des cases accessibles.

Enfin, viens le problème d'avoir une sortie accessible (autrement dit, un chemin entre l'entrée de la map, et sa sortie, que l'entrée soit un couloir venant d'une autre map, ou du point d'apparition du joueurs, d'ailleurs ces deux notions sont partagés, l'entrée est le point d'apparition du joueur). Pour réglé ce problème nous avons créé deux méthodes IsCheminFromTo et creationCheminInterne. La première vérifie l'existence d'un chemin entre deux Position, et la deuxième créé un chemin si la première indique qu'il n'en existe pas, à noter que la méthode IsCheminFromTo vérifie qu'il existe bien au moins un chemin, qu'importe sa longueur, si il existe un chemin demandant de faire 3 fois le tour de la map, c'est un chemin valide. Enfin si le chemin n'existe pas, on en créé de manière 'brut' c'est à dire que l'on creuse simplement au travers de la map en supprimant des blocs inaccessible (on creuse simplement sur l'axe x puis y sans se soucié de savoir si le premier bloc supprimé suffit à rendre accessible un chemin).

Les deux premières solutions règle leur problème respectif sans amener de bug, ou de fonctionnement étranges, en revanche la troisième amène une situation gênante, majoritairement dû au fonctionnement de Java. Java travaille uniquement par adresse, c'est à dire qu'écrire A = B, avec B une map déjà créé et A une variable de type map, reviens à dire que chaque modification sur A, se fera aussi B. Or, la méthode recherchant un chemin, détruit la map au fur et à mesure qu'elle la traverse (pour être sûr de ne pas revenir en arrière), ce qui rend obligatoire la création d'un constructeur de map visant à faire une copie d'une map donné en arguments. Action plutôt coûteuse qui justifie qu'on ne recherche pas l'existence d'un chemin entre Entrée et Sortie après la destruction de chaque élément pouvant bloquer un chemin.

Ce fonctionnement aurait pu être réglé en ne posant pas derrière chaque cases traversé une trace, mais plutôt en enregistrant chaque Position traversé pour ensuite

vérifié dans la liste des position déjà traversé si il est nécessaire de vérifié l'existence d'un chemin depuis cette position.

Notes supplémentaires : Comme expliqué précédemment , chaque map possède une entrée (le point d'apparition pour la première map) et une sortie, se situant sur un des 4 murs entourant la map, toutefois dans le code il existe 4 variables haut/bas/gauche/droite qui prévoie 4 sortie potentiel, il s'agit d'un reliquat datant d'une époque où nous voulions que toute les maps puissent être relié, l'étage utilisant toujours les variables haut/bas/gauche/droite pour appeler la méthode creationCheminDepuisExte avec les bon arguments, je n'ai pas modifié ces 4 variables, toutefois, du point de vue intérieur à la classe Map, c'est la variable posSortie qui remplace complètement ces 4 variables. De plus il existe dans le constructeur un argument int Sortie, cette argument permet de déterminé où doit se trouver la sortie, il s'agit simplement d'un int 1000 100 10 ou 1, pour haut/bas/gauche/droite, l'idée étant à la base que l'envoie comme argument de la valeur 1001, fasse une sortie en haut, et à droite, dans les faits, l'envoie de l'argument 1001 produira effectivement 2 sortie.

Toutefois ce n'est jamais utilisé, d'où le remplacement des variables haut/bas/gauche/droite par une unique variable posSortie.

Maintenant que nous avons vu la création de la map, viens la génération des cases à l'intérieur de celle-ci, les cases sont choisies lors de la méthode RemplissageMap qui demande la valeur de chaque case à la classe DecisionCase.

DecisionCase : Fait par Mascaro Valentin

L'objectif de cette classe est de fournir à la map une suite de caractères ayant une forme 'logique' c'est à dire que l'on ne souhaite pas obtenir une suite hasardeuse de caractères, mais bien une suite ordonnée, un arbre est rarement seul, un Lac n'est pas une seul case L mais une suite etc.

Le principal problème est d'obtenir cette logique de suite de caractères ordonnée, dans un tableau à deux dimensions, et le tout, sans se retrouver dans des abus générant des tableaux entier d'un seul et même caractère.

Viens aussi la question de comment transformer une suite de centaines de chiffres hexadécimaux (La Seed) en une map,
Il est impossible d'obtenir deux fois la même map puisque DecisionCase se créé à partir de la Seed de la map le créant, l'hétérogénéité des maps est donc déjà assuré.

Tout d'abord DecisionCase se créé avec comme argument, le seed, et 4 caractères (le 6éme argument est un reliquat d'ancienne version qui n'est plus utilisé). Ces 4 caractères détermineront les 4 caractères qui serviront de cases à la map (. , L X ou . , T X selon s'il s'agit d'une map Lac ou Forêt). N'importe quel caractères pourrait être utilisé, toutefois il faut ensuite modifié la méthode IsValide() de la Map pour rendre valide les caractères demandée (IsValide retourne True si le joueur à le droit de marché sur la cases visé) .

Afin de convertir la Seed, en un tableau de caractères on utilise la méthode suivante :

Chaque Seed possède 500 nombre entre 0 et 15. Notre map possède X cases (x quelconque), on va donc regarder le 1er nombre de Seed pour la première cases, puis on incrémente de 1 à chaque fois, si la map possède plus de 500 cases, la Seed reboucle sur elle-même, autrement dit demandé le 501éme nombre de la seed reviens à demander le 1er toutefois cela ne pose pas de problème de répétitions de cases, pour des raisons qui suivront.

Ensuite viens la décision de quel nombre, donne quel caractères.

Cette décision se schématise de la manière suivante :

0 1 2 3 4 5 | 6 7 8 9 | 10 11 12 | 13 14 15. les | indique des bordures, c'est à dire que le dernier caractères donner au constructeur de DecisionCase (nommé caseExceptionnel) demande que la nombre dans la Seed à la position regardé, soit entre 13 et 15 compris. Pour la caseRare, il faut que le nombre soit entre 10 et 12 compris. Etc.

Finalement viens la partie compliquée, obtenir une forme d'ordonnancement dans les cases qui sont sélectionné. Pour créer des formes de Lac/Foret (répétition de L ou T) il faut augmenter les chances de trouver un même type de case après que celle-ci ai été trouvé. C'est pour cela que :

Si on trouve un 11, donc une caseRare, on va modifier les valeurs des bornes :

0 1 2 3 4 5 | 6 7 8 9 | 10 11 12 | 13 14 15 devient

0 1 2 3 4 5 | 6 7 8 | 9 10 11 12 | 13 14 15 Ainsi j'ai moins de chance d'obtenir une caseCommune, et plus de chance d'obtenir une caseRare.

On répéte l'opération jusqu'à 4 fois, pour éviter de supprimer complètement les case inférieurs, une fois la répétition fait 4 fois on reset la valeur des bornes, limitant aussi les risques que des lignes soit entièrement d'un même caractère.

De plus, lorsque map demande une nouvelle case à DecisionCase, celle-ci donne trois arguments :

- Le premier, la valeur de la case précédente dans la ligne, celle-ci va influencer les bornes, si la case précédente est un X (caseExceptionnel), alors la borne de X va diminuer (donnant donc plus de chance d'obtenir un autre X)
- Le deuxième, la valeur de la case supérieur à la case qui va être créé, l'influence est la même que pour une case précédente dans la ligne.
- Si les deux arguments ont la même valeur alors la case demandée sera forcément de cette même valeur.
-Enfin le troisième argument, la valeur de la case en diagonal de celle visée, si la case en diagonal est la même que casExceptionnel, alors les chances d'avoir une casExceptionnel sont augmenté (ainsi les 'X' on tendance à se répété en diagonal)

Avec ces différentes règles le remplissage de la map voit apparaître des formes de biomes (des forêts, des lacs, des chaînes de collines). Les contrôles de répétition permettent d'évité que la map entière soit rempli d'un même caractère, mais tende a favorisé de très forte concentration d'un même caractère.

Notes importantes : La méthode expliqué ci-dessus prend compte la méthode DonneMoiUneCase(), il existe aussi une méthode DonneMoiUneMap() qui n'est pas implémenté (pour être honnête je penser même l'avoir supprimé des dernier merge request, et contre toute attente il en reste quelque lignes loin d'être fini) cette méthode avait pour but de modifié la génération en ne retournant pas les cases une par une, mais par bloc de biome.
L'idée étant de séparé la map en 2 à 4 cadran, chacun étant 1 biome différent. Toutefois j'ai abandonné cette méthode en cours de route, car bien moi convaincante que celle utilisé actuellement dans la release. L'objectif étant de limité la tendance des biomes a s'étendre sur toute la map en en créant des plus petits sur des parties défini sur la map, le point étant négatif étant que la génération actuelle marche bien sur des maps très grandes, mais moins bien sur des maps plus petites, puisque la répétition permet d'avoir des suites de 4 caractère identique, or pour peu que 2répétitions se déclenche d'affilé on se retrouve avec une immense ligne de 8 caractères identique, or ces 8 caractères identique vont énormément influencé la ligne du dessous, et donc potentiellement créé un biome de 8x4, sur une grande map de 25x25 cela ressemble simplement à une immense foret ou un immense lac, mais si la map est séparé en 4 cadran, alors on se retrouve avec un cadran entièrement composé d'un seul caractère. Or l'ajout de règle

visant a limité ces répétition a fini par créé une situation gênante dans lesquels les maps se ressembler toute, j'ai donc décidé de revenir en arrière et de supprimer cette idée.

Les limites de notre projet

L'intelligence artificielle

Un des aspects du projet que nous n'avons pas réussi à terminer est l'Intelligence Artificielle des monstres, ceux-ci, devait se déplacer vers le joueurs, commencé par Thibault, en utilisant l'algorithme A*. Une version (non fonctionnelle) du travail accompli est disponible sur la branche "pathfinding".

Interaction avec les éléments du jeu

Pendant que Thibault s'occuper de classes utilitaire, comme Seed / Colors / Generator, que Raphaël s'occuper des déplacements entres les Maps et que Valentin s'occuper de la map et de son contenu, une 4éme personne aurait pu s'occuper de ce qu'on appelle des Props.

L'idée étant qu'un « Props » est un élément du décor quelconque, sur la map le joueur, son icône, est un « Props ».

L'interface lié aurait donc eu une déclaration d'une méthode getSkin() qui retourne le caractère lié au Props sur la map, et une autre méthode aurait été interaction() qui, selon le Props aurait fait des interaction différentes, par exemple interaction() sur un marchand aurait déclenché une phase de commerce, interaction() sur un certain élément d'une zone de repos aurait soigner le joueur, interaction() sur un coffre aurait ouvert une liste d'item ramassable. Toutefois, cette interface n'a jamais vu le jour.

De plus, nous souhaitions mettre en place des interaction() avec le décors, pour ce faire, le joueur aurait eu une énième entrée au clavier possible, 'E', qui suivit d'une touche directionnel (z,q,s,d) aurait déclenché une interaction sur la case visé par la touche directionnel. Cette méthode nécessitant la case visée en argument ainsi que la position du joueur et ses compétences aurait donc dû être déclenché par le Player, et aurait selon certaines conditions déclenché des actions. Par exemple, si le joueur possède une compétence nommé 'coupe' et qu'il déclenche une interaction vers un arbre, alors l'arbre disparaît et laisse place à un sol de base (l'ordre aurait donc été 'E' 'z/s/q/d' Player.interaction(char direction), là la case player contrôle la position du Player et demande, en fonction de la direction souhaité, le type de la case visé par le Player map.getCaseAt(Pos case), selon la case retourné la classe Player contrôle si le player possède la compétence requise, si non, elle le prévient que c'est impossible, si oui elle

appellerai map.interaction(Pos caseViser) et map connaîtrait les interactions liés à chaque cases.

Dans le cas d'un arbre, celui-ci disparaîtrait de la map, dans le cas d'une case de Lac, les L se rajouteraient à la liste des cases valides pour un déplacement.

Marchand / Repos / Boss

Le jeu ne possède pas de maps spécial comme des maps de repos / boss ou encore des zones avec des marchands.
Générer des maps spécialement prévues à ce genre de rencontre n'aurait pas été particulièrement compliqué, pour un marchand ou une zone de repos une simple génération de la map avec aucune cases spéciales (uniquement des '.'), puis en fonction de s'il s'agit d'un marchand ou d'une zone de repos il aurait suffi de faire apparaître les Props liés à l'effet souhaité (comme expliqué ci-dessus).

Pour ce qui est des Boss il aurait suffi de créer une règle de génération spécial, à savoir augmenter les chances d'apparition de murs sur les extrémité de la map, et les réduire au fur et à mesure que l'on s'approche du centre (tout à fait possible en donnant les coordonnée de chaque extrémité de la map en paramètre du constructeur de DecisionCase), ce qui aurait eu tendance à créer une Arène de combat avec quelque murs pour permettre au joueur de se cacher (ou plutôt de faire tourner le boss en rond).

Il aurait tout à fait était possible de faire des combats de boss spécial, à savoir que le boss aurait pu par exemple 'lancé' des projectiles, en effets dans la classe Map se trouve des méthodes permettant de faire apparaître des caractère à n'importe quel Position, il aurait donc été possible pour le monstres de lancer des projectiles vers le joueur en appelant simplement cette méthode (méthode qui n'est, à l'heure actuel, utilisé uniquement pour effectuer des test comme par exemple pour bloquer un chemin et vérifié que nos fonction voit bien que le chemin est bloqué).

De plus les entités (Joueur et Monstre) possèdent une propriété leur indiquant sur quel type de case il se trouve, (l'intérêt étant de remettre sur leur ancienne position, la case qu'ils ont supprimé en marchant dessus) ainsi, si le joueur marche sur un des projectiles lancé par le monstres, nous avons un moyen de le savoir et donc de lui infliger des dégâts.

Conclusion

Malgré les fonctionnalités que nous avons pas réussi ou eu le temps de faire, ce projet fut enrichissant sur divers points, ils nous aura permis de travailler en groupe plus nombreux que les autres projets universitaire que nous avions eu à faire jusqu'à ce jour, il nous aura également permis d'approfondir nos connaissances de Java.