DAHMANI Sonia
Santos Duarte Elsa

Rapport de projet de LOA :

Sujet n°3

Création d'un jeu de combat (sans interface graphique)

Structure générale du château :

- Le château est un vecteur de vecteur d'adresses de pièces. On le voit donc comme une matrice et on récupère la pièce actuelle avec une fonction.

- Étant généré aléatoirement pour une partie, et ayant une durée de vie égale à celle de la partie, le château en est un attribut non-statique. Il est nécessairement carré et les liaisons entre les pièces sont effectuées de façon aléatoire, prenant cependant en considération le fait que chaque pièce dispose d'au moins une ouverture. Les clefs de téléportation existant, le château peut être non connexe. On a également fait en sorte qu'il soit toujours possible de se déplacer d'une pièce à une autre (ou au moins sortir de la première pièce de base, soit avec une ouverture normale, soit avec une clé.)

- Une fois le château créé, on le remplit.

Remplissage du château et gestion des pièces :

- Dans chaque pièce est placé un élément, objet ou personnage (ou plusieurs à la fois). également de façon aléatoire mais de manière à garder la partie intéressante. Notamment, le fait d'avoir toujours une arme dans la pièce de base, pour pouvoir avancer dans le jeu et s'engager dans des combats contre les personnages présents dans cette pièce et passer à la pièce suivante.

- On place un objet avec une probabilité de 60%, et en tirant aléatoirement l'objet en question. Quant aux personnages, on les place avec une probabilité de 40%, en tirant aléatoirement également le type du personnage. On s'assure que le nombre de clefs soit suffisant pour le château généré, c'est-à-dire qu'on puisse à-priori naviguer parmi les parties non connexes du château. Pour cela, on vérifie que chaque partie connexe dispose d'au moins une clef.
- La comparaison entre les différents objets qui remplissent les salles et la clef est rendue possible grâce à l'énumération "Objet" qui se trouve dans la classe Meuble et qui définit le type d'objet dont il s'agit.

Affichage du château:

- L'affichage du château est fait de façon à ce que la *première ligne* de chaque pièce représente les objets qui y sont présents. Ils sont représentés par leurs initiales. La *dernière ligne* de chaque pièce représente les personnages présents, représentés pareillement aux objets.
- Les différents types de personnages étant hérité d'une classe personnage commune, ils ont chacun une fonction d'affichage qui leur est propre. On exploite ici le polymorphisme fourni grâce à l'héritage de la classe Personnage: Deux fonctions "virtual", attaquer() qui renvoie la valeur des dégâts à infliger à l'ennemi qui sera négative et qui donc fera baisser la santé de l'ennemi ou à soi-même si jamais c'est un médicament ou un bouclier, qui elles, sont positives et seront ajoutées à la santé du personnage du joueur. Les valeurs renvoyées par la fonction attaquer() dépendent donc du type du personnage, dans le code, la bonne fonction est appelée. La deuxième fonction se charge juste de renvoyer l'initiale du type du personnage afin de mieux l'identifier dans le code principal.
- Les constructeurs sont également différents, pour chaque classe fille, le constructeur initialise les valeurs de santé et d'habileté par rapport au type de personnage.

Déroulement du jeu :

Le principal de la partie est géré dans le fichier Partie.cpp. C'est la fonction gerePartie() qui s'occupe de la boucle de jeu. Tant que la partie n'est pas finie, c'est-à-dire, tant que le joueur est vivant, et qu'il reste des adversaires, la boucle de jeu continue. Elle se découpe en quatre étapes :

I. Déplacement du joueur
II. Déplacement des personnages
III. Combat
IV. Gestion fin de combat

I. Déplacement du joueur

Elle affiche l'état actuel du château au/à le/la joueur.euse, et lui demande dans quelle direction il/elle veut se diriger, en affichant les différentes possibilités en fonction de la connexion entre la salle actuelle et ses salles mitoyennes. Les portes des pièces sont représentées par des booléens correspondant aux points cardinaux. En cas d'option invalide, on redemande au/à le/la joueur.euse de choisir à nouveau. Le/la joueur.euse a également le choix de ne pas changer de pièce et de rester dans la pièce actuelle.

II. Déplacement des personnages

Celui-ci se fait de façon aléatoire. On parcourt chacune des pièces, puis la liste des personnages qui y sont présents (il s'agit là du vector "Roommates" dans la classe Pièce).
On choisit de façon aléatoire, parmi les connexions possibles, dans quelle pièce on le déplace. On l'ajoute à la pièce en question, et on supprime le personnage de l'actuelle.
Comme il est dit en haut, l'implémentation actuelle rend possible le surplace, dans un certain cas : supposons deux salles, A et B. Lorsqu'un personnage est déplacé de la salle A à la salle B, il peut, au cours du même tour, être déplacé de la B à la A, et paraît ainsi ne pas avoir bougé d'un tour à l'autre. Cependant le/la joueur.euse pouvant également choisir de ne pas se déplacer, nous avons choisi de laisser cette possibilité à l'aléa.

III. Combat

On parcourt la liste des personnages présents dans la pièce. Si la pièce actuelle contient bien des adversaires, le/la joueur.euse doit s'engager en combat avec chacun d'entre eux. Chacun donne un coup dont les dommages dépendent de ses attributs (habileté, santé), de son type ainsi que, pour le/la joueur.euse, de l'utilisation d'objets de son sac. Si jamais le sac du/de la joueur.euse contient du poison, on offre la possibilité d'empoisonner l'arme, ce qui augmente les dégâts causés sur la santé de l'adversaire (on additionne les deux valeurs). L'ennemi a également accès à cette option, la décision est prise par un random. Le combat se poursuit tant qu'aucun des deux adversaires n'est mort. C'est donc une boucle qui permet à chaque adversaire de prendre une action, chacun son tour (voir les deux fonctions choisirAction() et subirAction()).

Au moment de choisir l'action, on affiche le contenu du sac, si jamais il est vide, on conseille au joueur de le remplir sinon on lui demande de choisir l'outil à utiliser lors de son tour. C'est ensuite au tour de l'ennemi, dans la fonction subirAction(), le choix est fait par un random. Pour les outils Bouclier et Médicament, il est possible de les utiliser que dans le cas où la santé de l'ennemi est inférieure à 150 sinon, le combat devient trop difficile, si c'est supérieur on force l'utilisation de l'arme. (C'est un cas qui ne concerne que le tour de l'ennemi)

Si le/la joueur.joueuse meurt la partie est finie. Sinon il/elle affronte les personnages restants.

À la fin d'un combat, on mettra toujours à jour la valeur d'habileté du personnage géré par le/la joueur.euse. Une façon de monter de niveaux et de gagner de l'expérience dans le jeu.

Lorsque tous les combats sont terminés, et que le/la joueur.euse est toujours vivant, il peut poursuivre vers les autres pièces et continuer à jouer.

IV. Gestion fin de combat

A la fin du combat, le/la joueur.euse choisit l'action de déposer un objet s'il/elle le souhaite afin de faire de la place dans son sac, auquel cas il est ajouté à la liste des objets déjà présents dans la pièce. Ou alors, il/elle peut ramasser un objet déjà présent, auquel cas, si l'espace dans son sac le permet et que l'objet ne figure pas déjà dans le sac, l'Objet est ajouté au sac et la boucle de la partie se poursuit.

Extension : Nous avions choisi pour extension la création de fichiers de sauvegarde. Nous nous étions alors tournées vers une librairie au code libre : rapidJSON. Son but était de simplifier la manipulation des JSON rendant la gestion de sauvegarde plus fluide. Les fichiers codes récupérés depuis cette librairie sont dans src/rapidjson. Le début d'implémentation des fonctions de création et d'utilisation de sauvegarde sont implémentées dans Partie.cpp.