Projet Logiciel Transversal

Les Dee & Di

AYROLES Quentin, EL BÈZE Ilan, MAERTEN Eloïse

2021 - 2022

Table des matières

1 Présentation Générale

1.1 Archétype

L'objectif de ce projet est la réalisation d'un jeu de rôle (RPG) classique inspiré par Donjons & Dragons.

1.2 Règles du jeu

Le but du joueur est de réaliser la quête définie en début de partie. Pour cela il dispose d'une équipe de 6 héros avec 6 rôles différents (Guerrier, Magicien, Assassin, Archer, Druide, Prêtre). Si le joueur est seul, il joue les 6 héros. En cas de multijoueur le joueur joue les classes qu'il choisit en début de partie (il y a donc au maximum 6 joueurs sur la même partie, si le nombre de joueurs est inférieur à 6, certains joueront plusieurs héros). Le multijoueur sera donc coopératif. Les joueurs évoluent sur une carte (sous forme de grille). Sur la carte, les joueurs peuvent être confrontés à des ennemis, événements et réaliser diverses actions(comme lancer des sorts). Ces actions auront une part d'aléatoire pour imiter les lancer de dés du jeu original. À terme, l'idée est d'aussi permettre au joueur de collecter des objets et d'améliorer ses personnages.

1.3 Ressources

FIGURE 1 – Textures des personnages

FIGURE 2 – Textures des décors

FIGURE 3 – Police d'écriture

2 Description et conception des états

2.1 Description des états

Un état de jeu est constitué d'un décor (tableaux d'éléments représentant la grille du jeu donc les murs, le sol...) et un autre tableau avec les personnages (appelé entité, qui peuvent à la fois être des héros ou des ennemis). Ce tableau est utilisé pour générer un tableau d'ordre de jeu des entités. On note aussi dans cet état à quel niveau les joueurs sont, si le niveau est fini, si le joueur actuel s'est déjà déplacé et/ou a attaqué ainsi que le résultat de dé de l'action effectuée.

2.1.1 Éléments fixes

Le décor est constitué d'une carte de case ayant chacun un type particulier, dans la liste suivante ainsi qu'une météo et une lumière.

Voilà les types de case possibles :

- None : état passif, la case "n'existe pas" pour le joueur
- Mur : élément infranchissable et un obstacle visuel (on ne peut ni voir ni tirer au travers par exemple)
- Sol : éléments de base sur lequel le joueur peut se déplacer
- Porte : élément infranchissable comme le mur nécessitant une action pour s'ouvrir et se transformer en sol
- Secret : case porte ayant une texture de mur pouvant se révéler après une certaine action
- Piège : case sol infligeant des dégâts à une entité se déplaçant dessus
- Trésor : case sol comportant un objet (ou équipement) que le joueur peut ramasser, elle peut donc devenir une case de type sol après la récupération de l'objet

Ces éléments de décor ont donc besoin d'une méthode leur permettant de passer de leur état de "cases spéciales" (trésor, piège...) à un état de case normale tout en effectuant l'action qui leur est associé (donner un équipement pour un trésor ou infliger des dégâts pour un piège).

2.1.2 Entités

Les entités sont les personnages du jeu, qu'ils soient héros (personnages interprétés par le ou les joueurs) ou ennemis (personnages contrôlés par l'IA).Une entité possède dans sa classe toutes ses caractéristiques, comme :

- niveau
- points de vie restants
- points de magie restants
- ses différentes statistiques comme les points d'attaque, de défense, de vitesse
- l'équipement qu'il possède comme des armes
- les statuts qu'il subit (gelé, confus...)
- les actions supplémentaires qu'il peut effectuer en plus de ses actions de base comme les sorts

L'ordre de jeu des entités est définie selon leur statistique d'initiative. Elle peut, durant son tour, se déplacer (selon sa statistique de déplacement) et/ou attaquer (selon sa statistique d'attaque) une seule fois. Une action supplémentaire remplace l'attaque dans son tour. Lorsque l'entité est attaqué elle se défend (selon sa statistique de défense) puis subit des dégâts et obtient possiblement un statut. Nous avons également une méthode correspondant à la mort de l'entité. Lorsqu'une entité meurt elle continue d'exister jusqu'à la fin du tour puis disparaît. Elle ne peut évidemment plus jouer à partir du moment où elle est morte.

Du côté des différences entre ennemis et héros, les héros ont, en plus de ces caractéristiques, une classe (qui définit leurs statistiques de base et leur texture) et la possibilité de récupérer de l'équipement ou apprendre des actions supplémentaires comme des sorts. Les ennemis ont une race, équivalente à la classe, qui définit donc leur statistiques de base et leur textures. Pour éviter que, pendant un tour, un ennemi soit

pris en compte alors qu'il n'est pas dans la même salle que les joueurs et donc pas accessible, on lui rajoute un attribut "actif" qui définit si les ennemis sont en jeu ou non. Cet attribut passe de inactif à actif lorsque le ou les joueurs sont suffisamment proche de lui.

2.2 Conception Logiciel

Le diagramme des classes pour les états est présenté en Figure 4, dont nous pouvons mettre en évidence les groupes de classes suivants :

Décor : La grille du plateau de jeu est constitué d'un tableau d'éléments de la classe "Décor". Un décor est constitué d'une dimension (nombre de cases en hauteur et en largeur) de tableau d'entier définissant la carte et donc le type de chaque case, un définissant la météo (neige, pluie, orage...) de chaque case et un définissant la lumière de chaque case.

Définition d'une entité : Une entité est soit de type Héros (jouable), soit de type Ennemi (non-jouable). La classe Entité est donc abstraite. Chaque entité peut avoir une liste d'équipements et d'actions supplémentaires qui améliore leur statistique et ajoute des acitons possibles.

Conteneur d'élément : La classe State est le conteneur principal, à partir duquel on peut accéder à toutes les données de l'état.

Sur la figure 4, nous avons donc en gris la classe qui contient toutes les informations, en rouge la classe qui défini le décor, en bleu les personnages en eux-mêmes et en jaune les objets et bonus pouvant être possédé par le joueur.

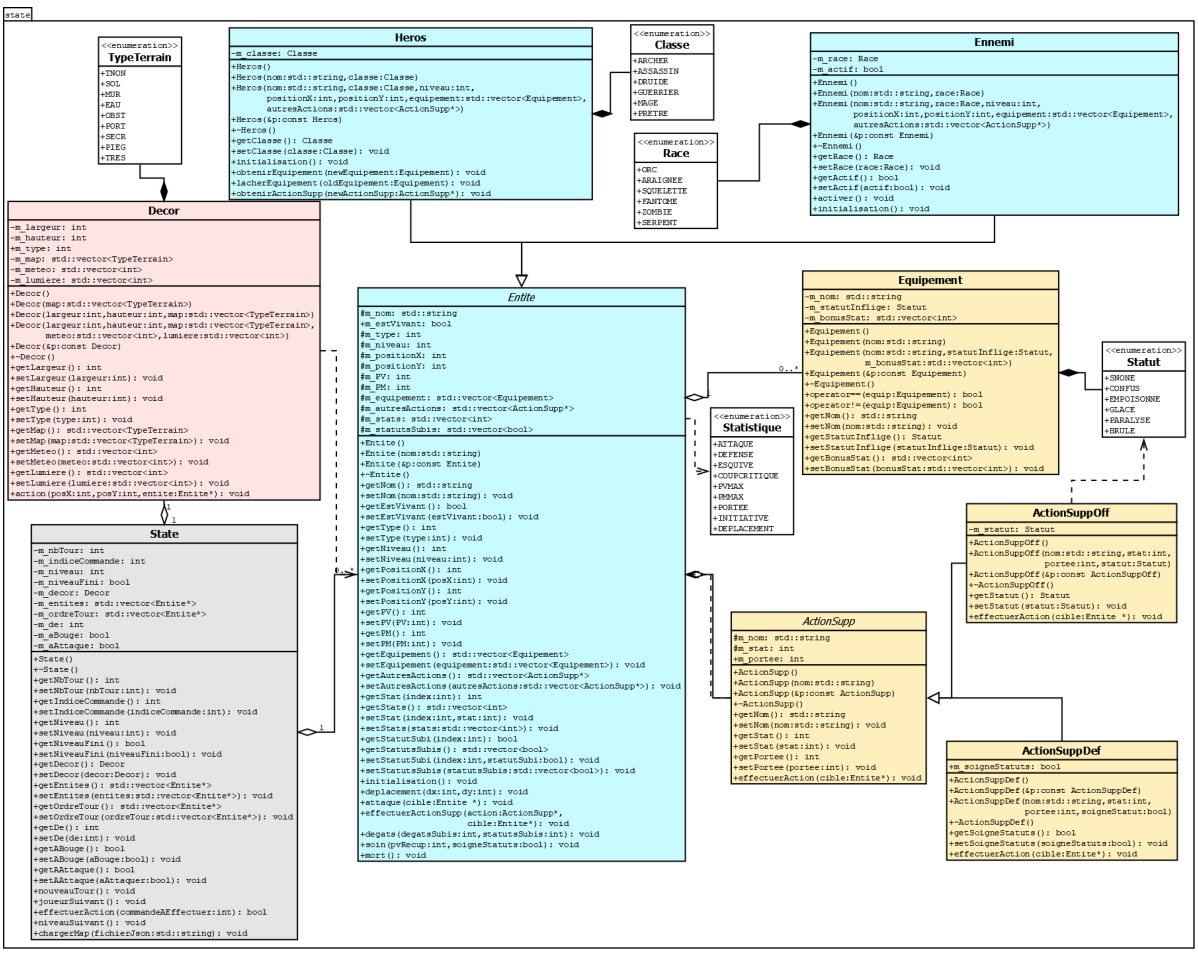

FIGURE 4 – Diagramme des classes d'état

Les différents éléments contenu dans le state en début de partie sont répertorié dans un fichier json, lu

en début de partie et qui permet d'initialiser le state en entier (à la fois le terrain et les différentes entités)

3 Rendu : Stratégie et Conception

3.1 Stratégie de rendu d'un état

Pour obtenir un rendu d'état convenable et simple à implémenter nous avons décider de fonctionner par couche. Nous utilisons la bibliothèque SFML pour permettre un affichage efficace et facile à gérer. Nous avons donc un groupe de couche de menus (1 à 5, qui affichent les informations nécessaires au joueur) et un groupe de couche de terrain (6, qui affiche l'état du jeu) en lui-même. L'affichage se présentera ainsi de cette manière :

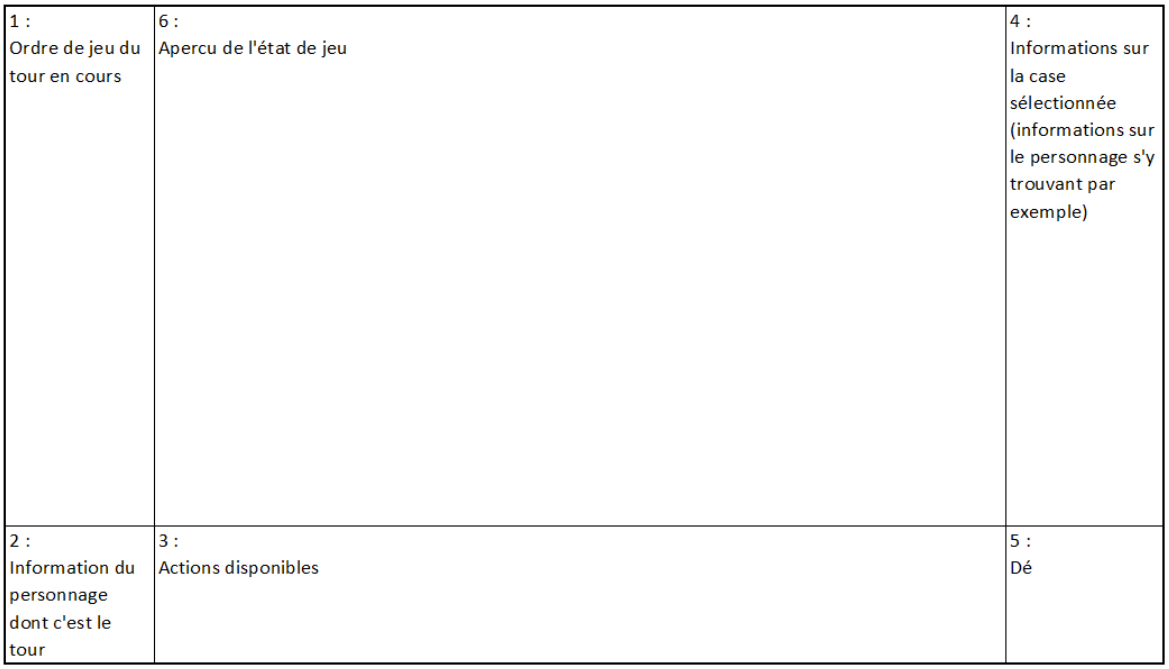

FIGURE 5 – Plan de l'affichage

- Case 1 : affichera le nom et l'ordre des entités en jeu (héros comme ennemis) et à terme pourrait aussi afficher le personnage en lui-même.
- Case 2 : affichera des informations de base (PV, PM, etc.) du personnage en train de jouer.
- Case 3 : nous permettra d'afficher les actions disponibles (attaquer, se déplacer...) et à terme nous pourrons afficher sur le terrain la portée de l'action sélectionnée.
- Case 4 : nous permet d'afficher les informations sur la case sélectionnée, si elle est vide le menu sera vide mais si un personnage est sur cette case cela affichera les informations de celui-ci. A terme nous pourrions rajouter les informations sur un objet déposé ou n'afficher les informations que si le monstre a déjà été affronté etc.
- Case 5 : nous permettra d'afficher le résultat du dé après la réalisation d'une action.
- Case 6 : nous permettra d'afficher le terrain en lui-même et les personnages.

Il sera nécessaire de gérer le nombre d'éléments maximum affichage (pour les actions ou le terrain par exemple) pour éviter qu'une case ne dépasse sur une autre ou des conflits entre les menus et les sélections du joueur.

De plus il nous faudra gérer plusieurs modes différents : le mode de jeu que nous avons défini ici, le mode menu, le mode nouvelle partie, etc.

Pour l'exemple, nous générons un état avec trois entités (Diana, Charles et Elisabeth) avec chacun une position et quelques statistiques, ainsi que des actions supplémentaires et voilà un rendu provisoire de ce à quoi le menu jeu pourra ressembler dans ce cas :

FIGURE 6 – Rendu de l'affichage

C'est également cette fenêtre qui récupère les instructions du joueur et en déduit les actions à effectuer. Selon l'action sélectionnée, le terrain aura en surbrillance les cases ou entités accessibles selon la portée ou le nombre de déplacement de l'entités en train de joueur ainsi qu'en plus foncé la case sélectionnée par le joueur. Pour valider une action, il suffit que le joueur appuis sur le bouton d'exécution à droite.

FIGURE 7 – Rendu de l'affichage

3.2 Conception logiciel

Pour la conception logiciel, nous avons décidé de fonctionner par couche, avec deux types de couches : les couches menus, qui afficherons majoritairement du texte avec un agencement précis, et les couches de terrain qui se chargerons d'afficher l'état du jeu donc le terrain en lui-même, les personnages et la portée de l'action sélectionnée. Nous avons donc deux classes qui héritent d'une classe couche globale qui nous permet de gérer plus facilement leur affichage.

L'organisation et l'affichage est géré par un objet Scene, qui va répertorier à la fois l'état de jeu, l'état du joueur, pour permettre l'affichage dans les couches. Ainsi les couches en elles-mêmes n'ont aucune information sur l'état de jeu et se contente d'afficher les informations issues de la scène. Pour permettre l'affiche, nous utilisons les classe sf : :Transformable et sf : :Drawable comme interface pour les couches.

Concernant l'affichage dynamique de l'état de jeu, nous avons une fenêtre qui reste affichée tant qu'elle n'est pas fermé et vérifie l'état de jeu toutes les seize millisecondes (60Hz), pour afficher l'état actuel. Plus tard, nous pourrons implémenter une fonction qui n'affiche l'état actuel qui si celui-ci a changé, réduisant ainsi les appels.

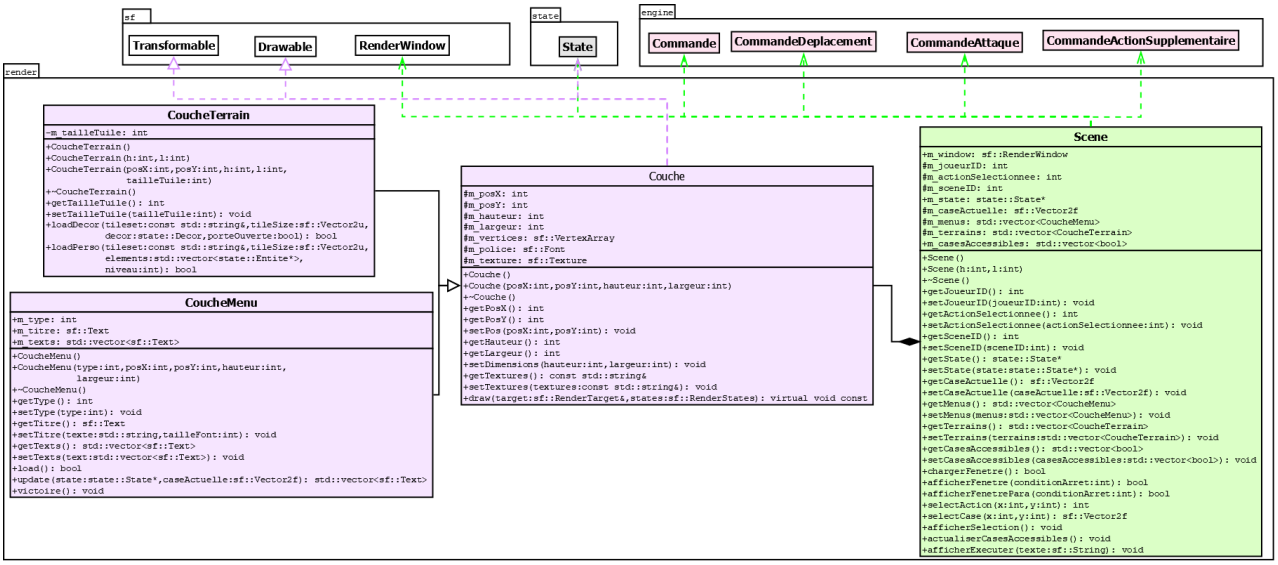

FIGURE 8 – Diagramme de classe de render

4 Règles de changement d'états et moteur de jeu

4.1 Règles

Nous allons commencer par des règles simples. Un personnage ne peut se déplacer que du nombre de cases indiqué par sa statistique de portée et ne peut attaquer ou utiliser une action supplémentaire qu'une seule fois par tour. Une fois ces actions effectuées, le joueur choisi quand laisser son tour se finir (lui laissant ainsi le loisir de lâcher un équipement par exemple). Concernant le déplacement, une entité peut se déplacer au maximum de sa statistique de déplacement sans traverser ni les murs ni les personnages et ne peut donc pas s'y arrêter. Il en va de même pour la portée des attaques ou des autres actions avec la statistique de portée. Les actions supplémentaires défensives soignent des Points de Vie (PV), dans la limite des PV maximum de l'entité (définis dans les statistiques de chaque entité.

4.2 Conception logiciel

Pour la conception logiciel, nous avons créé une classe Engine, qui stocke l'état actuel ainsi qu'un attribut json qui nous permettra à terme d'enregistrer les données du jeu. Cette classe Engine utilise des éléments de la classe Commande avec la méthode "addCommande". Cette dernière permet de vérifier si l'action lié à la commande est valide et si il l'est elle actualise l'état. De la classe abstraite Commande héritent trois classes : CommandeDeplacement, CommandeAttaque et CommandeActionSupplementaire. Elles permettent respectivement de créer une commande qui permet d'effectuer un déplacement, une attaque ou bien une action supplémentaire (comme un sort). Chacune de ses classes possède des méthodes handle (qui sont aussi virtuelle dans la classe Commande) qui permettent de vérifier les différentes règles liés à l'action.

5 Intelligence Artificielle

5.1 Stratégies

5.1.1 I.A. Aléatoire

L'intelligence artificielle aléatoire a accès aux différentes commandes du moteur de jeu. Lorsqu'elle agit, elle choisit une de ces commandes au hasard. Les paramètres associés sont aussi choisis de manière aléatoire, tels qu'ils donnent lieu à une commande valide.

L'intelligence artificielle peut déplacer l'entité dont c'est le tour sur une case du plateau accessible par cette entité. Cette case est choisie de façon aléatoire.

Elle peut aussi essayer d'attaquer une entité choisie au hasard. Si aucune entité n'est à portée, elle passe son tour.

Il est important de noter que tous les coups réalisés par l'I.A. aléatoire sont légaux.

5.1.2 I.A. Heuristique

L'intelligence artificielle heuristique a un comportement différent. Elle commence par déterminer quel ennemi est le plus proche. Pour cela elle regarde la différence (absolue) du nombre de cases entre sa position et celles des ennemis sur les deux axes.

Une fois cela fait, l'I.A. heuristique essaie d'effectuer une attaque sur l'ennemi le plus proche. Si l'attaque est possible, elle le fait et finit son tour. Si elle ne peut pas attaquer (si l'ennemi le plus proche n'est pas à portée par exemple), elle se déplace vers l'ennemi le plus proche (sur un axe seulement, l'axe sur lequel il y a le plus de cases de différence) puis réessaie d'effectuer son action. Si l'action ne peut pas avoir lieu, son tour prend fin.

5.2 Conception logiciel

Le diagramme des classes pour l'intelligence artificielle est présenté en Figure 9.

La classe AI donnera lieu à des classes héritées implémentant différentes intelligences artificielles plus ou moins poussées.

La classe RandomAI implémente une intelligence artificielle dont les actions sont dictées par l'aléatoire. Sa donnée membre m_mt est un générateur de nombres aléatoires.

La classe HeuristiqueAI implémente une intelligence artificielle qui se comporte selon des règles simples.

5.3 Exécution et comparaison

Il nous est possible d'observer le comportement de ces différentes IA. En effet en effectuant la commande `./bin/client render` permet de jouer contre une IA heuristique. Il est à noter que pour passer au niveau suivant il faut systématiquement appuyer sur la porte. La commande `./bin/client heuristic_ia` nous permet de voir deux IA s'affronter et la commande `./bin/client randomVSheur` permet de voir s'affronter une IA random (qui contrôle les ennemis) et un IA heuristique (qui contrôle les héros). Nous avons grâce à cela remarqué que l'IA heuristique gagnait quasi systématiquement (a toujours gagné sur plus de 10 essais).

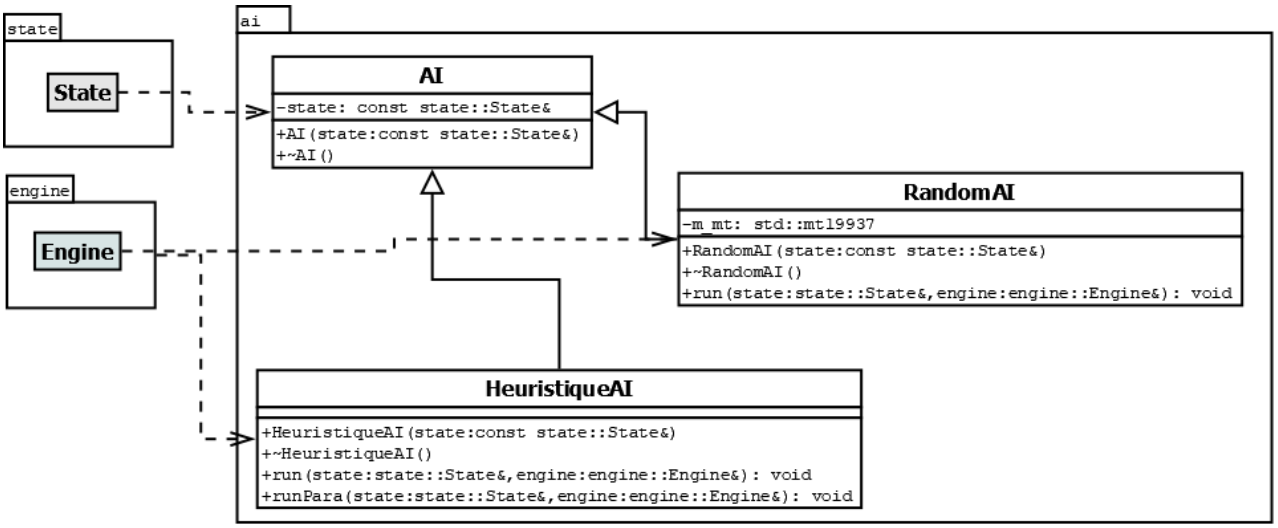

FIGURE 9 – Diagramme de classes d'ai

6 Modularisation

6.1 Organisation des modules

Pour la modularisation nous avons choisi de ne pas recréer de nouvelles classe mais de créer des méthodes en plus dans les classes existantes. Ainsi nous avons créer la méthode Scene : :afficherScenePara, qui fonctionne comme son homologue mais ici les actions sélectionnés ne sont plus directement effectuées mais stockées dans un fichier "commande.json" qui sera lu et exécutés par le moteur de jeu. De la même façon nous avons créé une fonction heuristiqueAI : :runPara qui procède de la même façon. Pour stocker ces commandes nous utilisons un format simple comportant le type d'action à effectuer, la position dans le cas d'un déplacement, la cible dans le cas d'une attaque ou action et l'action supplémentaire concernée dans le cas d'une action. Enfin lorsque l'action voulue est de passer son tour, il ne contiendra que l'information "Passer". Le document pourra par exemple contenir ces informations :

```
 {
   "Cible" : "Araignee",
   "TypeId" : "Attaque"
},
{
   "TypeId" : "Passer"
},
{
   "TypeId" : "Deplacement",
   "x" : 8,
   "y" : 4
}
```

6.2 Conception logiciel

Contrairement aux versions sans parallélisation, ici nous lançons dans un thread à part dans le main, la fonction Engine : :actualiser et ce toutes les 10 millisecondes. Cette fonction vérifie que le moteur de jeu est à jour par rapport au state (c'est à dire que l'attribut Engine : :m_indiceCommande est égal à State : :m_indiceCommande) et si ce n'est pas le cas il effectue la commande suivante sur la liste. Le moteur actualise ainsi le state et celui-ci sera visible par le joueur à la prochaine actualisation du render.

Dans notre programme, la commande `./bin/client para` permet de joueur en utilisant la parallélisation, ainsi que de sauvegarder la partie. `./bin/client thread` permet de voir s'affronter deux Ia heuristiques de la même façon, c'est à dire en utilisant un thread séparé pour le moteur et en sauvegardant la partie. `./bin/client replay` permet de revoir la dernière partie jouée.