Ecole nationale supérieure de l'électronique et de ses applications

RAPPORT PLT
Réalisation d'un T-RPG type Disgaea

Authors :
Benjamin Anslot
Mathieu Hadjadji

Années :
2019 - 2020
OPTION IS

10 janvier 2020

Table des matières

Chapitre 1

Présentation Générale

1.1 Archétype

L'objectif est de réaliser un jeu video de type Tactical RPG dans l'idée du jeu Disgaea.

FIGURE 1.1 – Aperçu de l'ecran de Jeu du jeu Disgaea

1.2 Règles du Jeu

1.2.1 But du Jeu

Les joueurs ont chacun un nombre défini de troupes positionné sur la carte. Le But du jeu est d'éliminer l'ensemble des troupes ennemies (condition de victoire). Ou d'avoir toutes ses troupes éliminées (condition de défaite).
Une unité est considérée comme détruire lorsque ses points de vie tombent à 0.

1.2.2 Tour de Jeu

Le jeu se découpe en une sucession de tours. Les joueurs jouent les uns à la suite des autres.

Dans un tour, le joueur peut :

— Déplacer ses unités sur la carte sur un nombre limité de cases.
— Attaquer avec ses unités des unités adverses adjacentes ou à distance en fonction du type d'attaque utilisable par l'unité.
— Appliquer une posture de défense sur ses unités pour augmenter la défense si attaquée.
— Utiliser des objets sur ses unités pour la soigner etc...

1.2.3 Terrain de Jeu

Le terrain est décomposé en tuiles. Une unité reçoit des bonus ou des malus en fonction de la tuile sur laquelle elle se trouve. Une unité placée sur une tuile de forêt par exemple recevra un bonus d'esquive si attaquée.

Les tuiles peuvent avoir des hauteurs différentes. Une unité placée sur une tuile en hauteur aura une meilleure portée pour des attaques à distance, mais y accèder coûte plus de points de déplacement.

1.3 Ressources

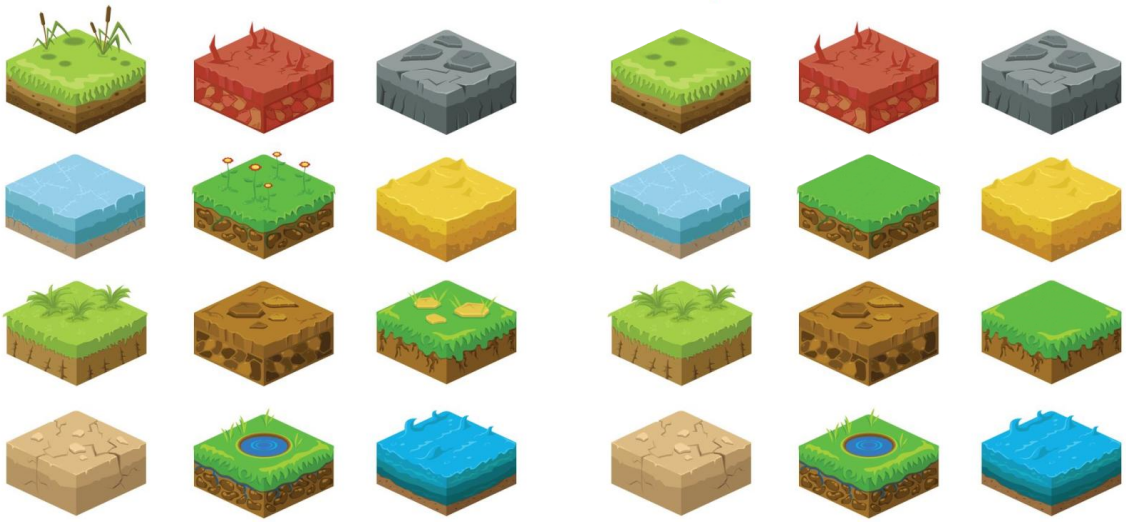

FIGURE 1.2 – Tiles pour la carte en perspective isométrique

FIGURE 1.3 – Spritesheets personnages

Chapitre 2

Description et Conception des états

2.1 Description des états

Dans un état du jeu, on retrouve des éléments fixes : les tiles (tuiles isometriques) qui composent la carte et des éléments mobiles : les personnages. Tous les éléments ont un nom et une position.

2.1.1 Etat éléments fixes

La carte du jeu est une table à 2 dimensions composée de tuiles.

Les tuiles sont générées de manière aléatoires. On retrouve dans les propriétés des tuiles :

— **le type de tuile :** Il est critère de praticabilité pour un personnage et aussi pour l'affichage du sprite correspondant côté gestion rendu.
  - Dirt
  - Grass
  - Water
  - Sand
  - Pound
  - Rock

  Les effets des types seront déterminés côté engine en fonction du type de tuile.

— **la hauteur :** Elle determine la hauteur de la tuile, modifieur impactant la distance de déplacement dans les actions et la portée d'une attaque qui seront gérés dans la partie engine. La hauteur est un entier compris entre 1 et 3.

2.1.2 Etat éléments mobiles

Les personnages appatiennent à des équipes. Chaque équipe est gérée par un joueur. Un personnage possède :

— Une race générée aléatoirement parmi la liste suivante :
  - Monster

- Beastman
- Demon
- Human

— Un job généré aléatoirement parmi la liste suivante :
- Pugilist
- Swordsman
- Archer
- Magician

— Un niveau calculé en fonction de l'expérience amassée.

Les caractéristiques (PV max, PM max, dégâts physiques, dégâts magiques, score d'évasion, score de défense, liste des compétences) sont déterminés en fonctions des 3 attributs précédents (race,job et niveau).

Un curseur permet aux joueurs de navigueur sur les tuiles et lire les propriétés (du terrain et d'un personnage si présent) et sélectionner un personnage (si présent) en vue d'effectuer une action.

2.1.3 Etat général

En plus des éléments statiques et mobiles nous avons :
— **nombre de tour :** Le nombre de tours qui ont été joués.

— **terminé :** Un booléen qui indique si le combat est terminé.

2.2 Conception Logiciel

Le diagramme des classes UML C++ pour les états est visible en Figure X. Nous pouvons mettre en évidence plusieurs groupes de classes qui sont les suivants :

— Groupe Character (bleu foncé) : Toutes les charactéristiques des personnages, qu'ils soient ceux du joueur ou non, ainsi que les fonctions pour y accéder. Il présente une hiérarchie des classes filles permettant de visualiser les différentes catégories associées aux personnages et les types de chacun de ses élements.

— Groupe Team (bleu clair) : Principalement composé de la classe Team, cette dernière englobe les ressources d'un joueur à savoir ses personnages et ses objets.

— Groupe Entity (jaune) : La gestion du curseur et de l'entité pointée par ce dernier pour afficher les propriétés de la tuile à l'écran et selectionner le personnage qui réalise une action.

— Groupe Observante (orange) : Il sera implémenté en parallèle avec le render, pour pouvoir faire les tests.

— Groupe Turn (vert) : Il représente l'état général durant un tour.

— Groupe Tile (gris) : Ce groupe permet la construction d'une Tile aléatoire à l'aide de TileFactory. La carte sera formée à partir de ces Tiles.

2.3 Tests Unitaires

Des tests unitaires ont été réalisés pour vérifier les fonctions des différentes classes. On réalise un fichier de test par classe. Le code est couvert aux alentours de 88% par les tests unitaires.

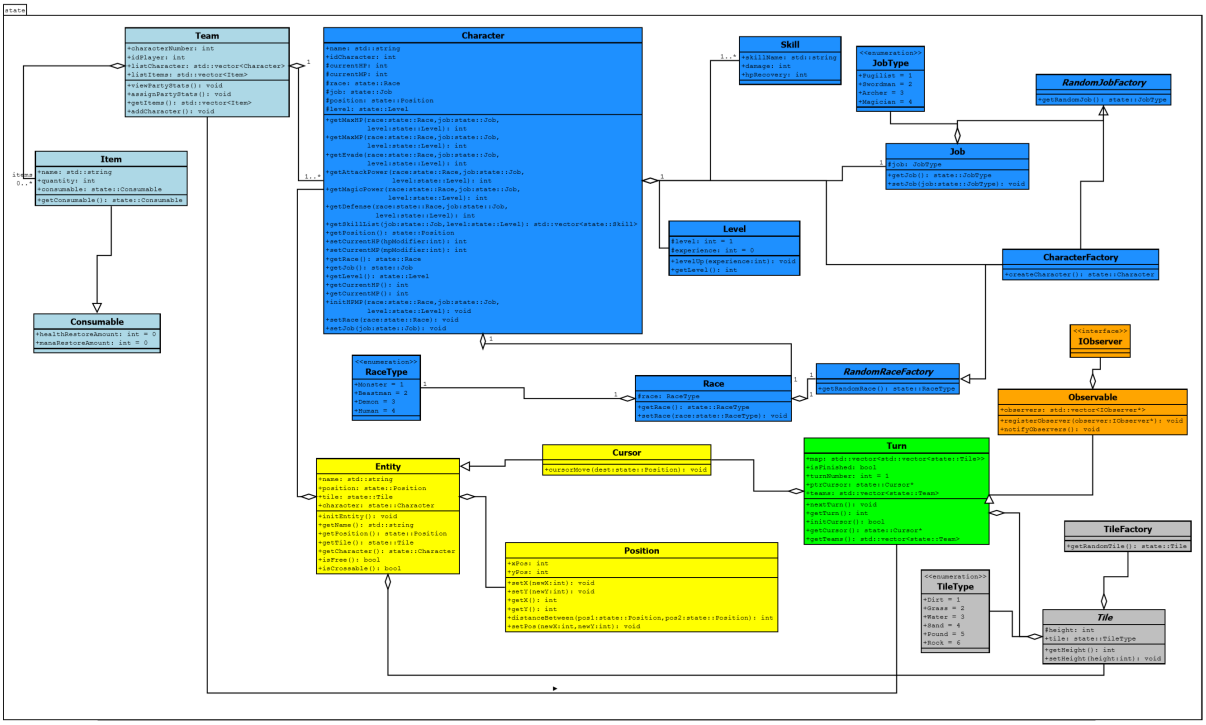

FIGURE 2.1 – Aperçu de state.dia

Chapitre 3

Description et Conception du rendu

3.1 Description de la stratégie de rendu d'un état

Pour le rendu d'un état nous avons décidé de réaliser une vue isométrique à l'aide de la librairie SFML de notre carte générée aléatoirement. Il est possible de faire entrer en rotation la vue avec les touches R et T, et déplacer la vue avec les touches directionnelles.

Nous découpons la scène à rendre en couches (ou « layers ») : une couche pour chaque hauteur de tuile (1 à 3) et une couche par personnages. Chacunes des couches de tuile est divisé en deux : les murs et le plafonds. Le plafond d'une couche à la priorité sur les murs de cette même couche et une couche supérieure à priorité sur une couche inférieure.

Chaque couche contiendra des informations qui seront utilisées par la libraire Sfml : un unique tileset contenant les tuiles (ou « tiles »), et une unique matrice avec la position des éléments et les coordonnées dans la texture.

En ce qui concerne les aspects de synchronisation, nous avons une horloge à 60Hz qui permet l'actualisation du rendu. De plus, nous avons aussi les animations des personnages fixes qui arrivent toutes les 60ms (soit environ 17Hz).

3.2 Conception Logiciel

Le diagramme des classes UML C++ pour le rendu est visible en Figure 3.3.
On divise le Rendu en 3 classes :

— **La classe TileSet :** Elle est constitué d'un constructeur où l'on précise le type (Maptile, CharaSpritesheet) pour créer un objet TileSet avec :
  - le chemin vers le tileset (imagePath)
  - la taille horizontale d'une tuile (sizeX)
  - la taille verticale d'une tuile (sizeY)
  - la marge, espace entre les tuiles (margin)

— **La classe DrawObject :** Elle regroupe les fonctions de rendu des objets (plafonds,murs et personnages) et une fonction virtuelle draw redéfinie de la librairie sfml.
  Elle possède 2 attributs utiles au rendu : un objet texture de la librairie sfml et un

tableau de vertex.
Pour réaliser le rendu isométrique de la carte, on écrit les coordonnées X et Y iso dans le repère Cartésien.
Et on définit nos vertex comme suivant :

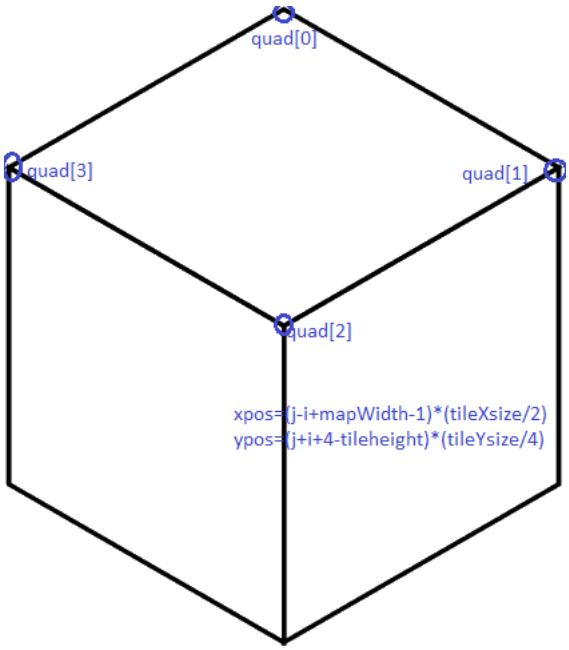

FIGURE 3.1 – Schema calcul position isometrique

quad[0].position = sf : :Vector2f(xpos + tileXsize/2 , ypos + tileYsize/2) ;
quad[1].position = sf : :Vector2f(xpos + tileXsize , ypos + 3*(tileYsize/4)) ;
quad[2].position = sf : :Vector2f(xpos + tileXsize/2 , ypos + tileYsize) ;
quad[3].position = sf : :Vector2f(xpos , ypos + 3*(tileYsize/4)) ;

— **La classe TurnDisplay :** La classe TurnDisplay est constituée d'un constructeur qui génère un objet TurnDisplay pour un tour (turn) donné. Il possède la réference à ce tour, une liste de tilesets et une liste de drawobjects vide. Sa fonction initRender appelle les fonctions d'une instance de DrawObject et les append à sa liste de drawobjects.
Pour les personnages, on fait un tableau à 2 dimensions pour stocker chaque sprites de l'animation des personnages.

FIGURE 3.2 – Aperçu du rendu d'un etat de jeu

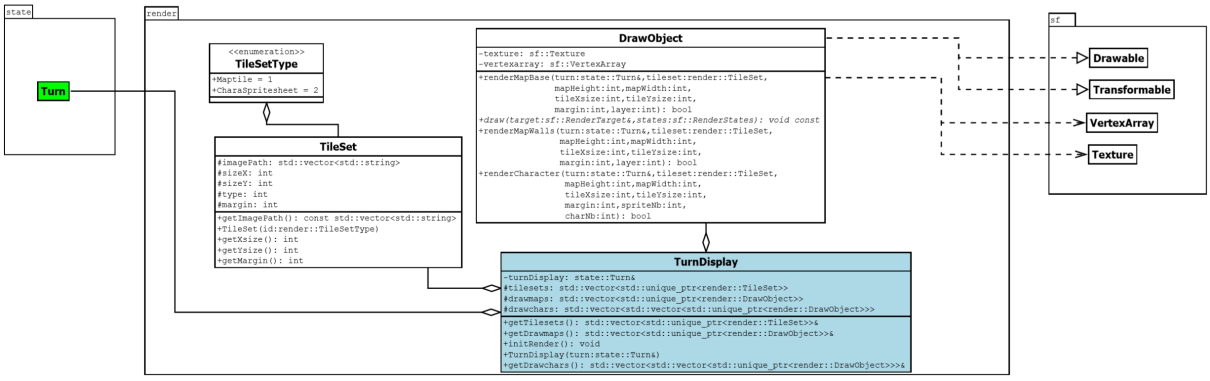

FIGURE 3.3 – Aperçu de render.dia

Chapitre 4

Règles de changement d'états et moteur de jeu

4.1 Règles de changement d'états

Le tour de jeu d'un joueur est terminé lorsque tous ses personnages possèdent un statut USED ou DEFENDING.
Les actions sont réalisées en fin de tour dans l'ordre entré par le joueur actif. C'est alors le tour du joueur adverse.
Lorsqu'un personnage est attaqué par un ennemi, il perd des points de vies correspondant à l'attaque de l'attaquant moins sa défense. Lorsqu'un personnage utilise un objet, il peut l'utiliser sur lui-même ou sur un autre personnage, pour gagner des points de vie par exemple.
Si un personnage perd tous ses points de vie, son statut évolue et prend la valeur DEAD.
Si tous les personnages d'un joueur meurent, la partie est terminée à la fin du tour et le joueur adverse gagne.
Chaque action effectuée modifie l'état des personnages concernés, autant celui la réalisant que les victimes. Le choix du personnage sélectionné et des actions effectuées est provoqué par des commandes. Pour ce jalon une séquence de commande est réalisée dans le fichier "main.cpp".
Le scénario de test comporte 5 tours, on peut jouer un tour en appuyant sur la touche E.

4.2 Conception logicielle

Le diagramme des classes UML C++ pour le moteur est visible en Figure 4.1. On divise le moteur de jeu en 3 groupes de classes.

— **Le agroupe de la classe Engine :** C'est le coeur du moteur du jeu. Cette classe permet de stocker les commandes dans un vecteur de Command (avec l'opération "addCommand"). Ce mode de stockage permet d'introduire une notion d'ordre : on réalisé les commandes dans l'ordre où le joueur les à rentrées. Lorsque turnCheckOut est appelé les commandes sont exécutées, si tout les personnages ont des actions ou s'ils passent leur tour, puis on change de joueur actif.

— **Le groupe de la classe Command :** C'est la classe maîtresse des actions. Elle définie les méthodes de manière abstraite afin quelles soient utilisables et modi-

fiables par toutes les actions grâce au polimorphisme.

— **Le groupe des classes Attack, Defend, Move, EndTurn, UseObject et UseSkill :** Les classes Attack, Defend, Move, EndTurn, UseObject et UseSkill, héritant de la classe Command, possèdent chacune une méthode "validate" qui permet de savoir si le personnage et capable de réalisé la commande et une méthode "action" qui fait exécuté au personnage l'action correspondante à la commande. Chacun possède aussi un constructeur permettant de savoir quel personnage réalise l'action, quel(s) personnage subit l'action et les spécificités liées à chacune des actions.

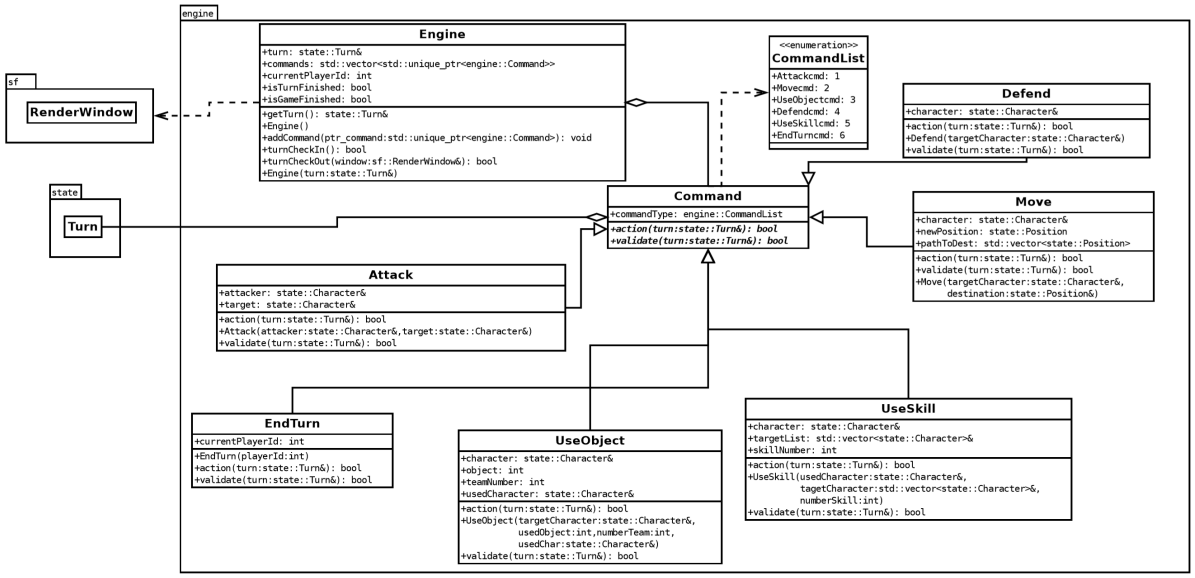

FIGURE 4.1 – Aperçu de engine.dia

Chapitre 5

Intelligence Artificielle

5.1 Stratégies

5.1.1 Intelligence Aléatoire

L' IA contrôle une liste de personnages appartenant au joueur actuel, elle va les sélectionner les uns après les autres lors de son tour de jeu. Une fois un personnage sélectionné, un entier correspondant à une action précise est choisi au hasard parmi 6 : 1 (attaquer), 2 (se déplacer), 3 (utiliser un objet), 4 (défendre), 5 (utiliser une compétence) et 6 (passer son tour).

Lorsqu'une action est séléctionnée on teste sa faisabilité, sinon un nouveau nombre est séléctionné jusqu'à ce que l'action soit valide.

Si l'action est possible alors elle est ajoutée à la liste des commandes et l'IA passe au personnage suivant.

Si "attaquer" est choisie, une cible va être déterminée aléatoirement parmi tout les autres personnages, y compris ceux de son équipe. Si la cible de l'attaque est trop éloignée alors l'attaque n'aura pas lieu, si la distance entre les deux personnages est suffisante alors l'attaque est ajoutée à la liste des commandes.

Si "se déplacer" est choisie, le personnage sélectionné tente de se mouvoir sur une case aléatoire, si cela est réalisable alors le déplacement est ajouté à la liste des commandes.

Si "utiliser un objet" est choisie, alors le personnage va utiliser un objet aléatoire de la liste des objets appartenant à son équipe, puis il applique l'effet sur une cible aléatoire, si l'objet est en quantité suffisante alors l'action est ajoutée à la liste des commandes.

Si "défendre" est choisie, le personnage sélectionné se met en position défensive, dans le cas présent puisque l'on cherche les actions des personnages dans l'ordre, cette action ne peut échouer puisque le personnage n'a pas d'autres actions dans la liste, l'action "défendre" est donc ajoutée à la liste.

Si "utiliser une compétence" est choisie, alors le personnage va utiliser une compétence

aléatoire de la liste des compétance qu'il peut utiliser, puis il applique l'effet sur une cible aléatoire, si la distance entre lui et sa cible est inférieure à la distance maximum de la compétence alors la compétence est ajoutée à la liste des commandes.

Si "passer son tour" est choisie, l'IA termine son tour d'action. Lorsque passer son tour ou lorsque tout les personnages de l'équipe du joueur actuel ont une action, alors cela fini ainsi la liste des commandes, les commandes sont exécutées dans l'ordre de la liste. Puis c'est au tour de l'adversaire.

5.1.2 Intelligence Heuristique

L'IA heuristique fonctionne sur un système de point calculé pour chaque action possible. L'IA va lister les actions possibles par personnages de son équipe et calculer un score pour chacune de ces actions. Le score est calculé en fonction de la pertinence des actions.

Concrètement, un personnage peut se déplacer puis effectuer une action ou seulement effectuer une action. Il privilégiera des déplacements où les personnages adverses ne peuvent pas l'atteindre après déplacement ou bien s'il peut attaquer un unique ennemi avec un minimum d'attaque subit le tour suivant. C'est-à-dire en termes de point, plus il y a d'ennemis qui pourront l'attaquer au tour suivant plus il va perdre de point pour cette action, s'il y a un unique ennemi qu'il peut attaquer après son action de déplacement il va gagner des points. De plus il gagne des points suplémentaires s'il est hors de porté d'attaque et de déplacement puis attaque.

Une attaque est privilégiée si l'ennemi est tué en conséquence de l'action. Elle est priviligié si le personnage ne peut subir que peu d'attaques au prochain tour, sinon l'IA privilégie la défense qui réduit grandement les dégats subits au prochain tour adverse. En termes de point, une attaque rapporte plus de point qu'une action classique puisque le but du jeu est de tué les personnages adverses. Si une attaque tue un adversaire alors l'action rapporte encore plus de points.

Un objet de soin n'est utilisé que lorsque celui-ci est nécessaire (vie$<$20% ou mana$<$20%) afin qu'il ne soit pas utilisé sur des cibles qui n'ont pas perdu des points de vie ou des points de mana comme pourrait le faire l'IA aléatoire. En termes de point l'action perds des points si le personnage à son maximum de vie ou de mana, il gagne plus de point si les points de vie ou de mana sont sous le seuil critique (20%).

En terme de point une action défensive gagne de plus en plus de point plus il y a de personnages de l'adversaire qui pourront l'attaquer au prochain tour.

5.1.3 Intelligence Profonde

L'IA profonde va lister les actions possibles par personnages de son équipe et calculer un score pour chacune de ces actions, comme le fait l'IA heuristique.

On finit dans un premier temps le tour en cours en complétant la séquence actuellement obtenue de l'IA profonde, par celle que l'on obtiendrait de l'IA Heuristique.

Le score est calculé après 2 tours de chaque équipe en considérant que ces tours sont exécutés par l'intelligence heuristique, c'est-à-dire que les tours sont réalisés dans l'optique d'obtenir les meilleures actions possibles pour le tour en prédisant les coups de l'adversaire.

On calcule le score en fonction des dégâts subis par l'équipe du joueur et ceux subis par l'adversaire.

Après le calcul du score on exécute les tours ainsi effectués dans le sens inverse pour revenir au tour initial grâce au rollback implémenté. La fonction revertTurn de Engine se charge de lancer les actions réciproques des actions exécutés durant un tour pour retourner au tour précédent et on décrémente le nombre de tours passés. On s'évite ainsi de cloner l'objet turn.

Après être revenu au tour actuel de jeu, on recharge la file d'actions retenues par l'IA profonde pour le tour en cours.

Cette boucle est ainsi exécutée jusqu'à ce que plus aucun personnage ne puisse effectué d'actions. On exécute alors la liste des commandes obtenues avec les meilleurs scores.

5.2 Conception Logiciel

Le diagramme des classes UML C++ pour l'intelligence artificielle est visible en Figure 5.1. On divise l'intelligence artificielle en 4 classes.

— **AI :** C'est la classe principale, elle définie les méthodes qui serviront lors du polymorphisme lorsque les différentes IA seront appelées.

— **RandomAI :** C'est la classe représentant l'IA aléatoire, elle a une fonction pour determiner la liste des actions
des personnages controllé par l'IA (randomCommandList) et une fonction pour executer lancer l'IA (runAI).

— **HeuristicAI :** C'est la classe dédiée à l'IA heurisitique. Elle comporte une fonction computeScore qui renvoie le score calculé pour l'action donnée en entrée, une fonction heuristicCommandList pour déterminer la liste des actions à partir des scores et une fonction runAI pour lancer l'IA.

— **DeepAI :** C'est la classe de l'IA profonde. Elle comporte une fonction minMax qui renvoie le score en considérant le meilleur tour de l'adversaire puis le meilleur tour du joueur (2 fois). Elle comporte aussi une fonction deepCommandList pour déterminer la liste des actions à partir des scores, une fonction runAI pour lancer l'IA. Elle comporte également une fonction restoreCommandlist pour restorer la

liste des actions en cours de construction après avoir fait deux tours de simulation et être retourné à l'état de tour actuel. La fonction updateQueue se charge de tenir à jour la file d'actions retenues par la deepIA.

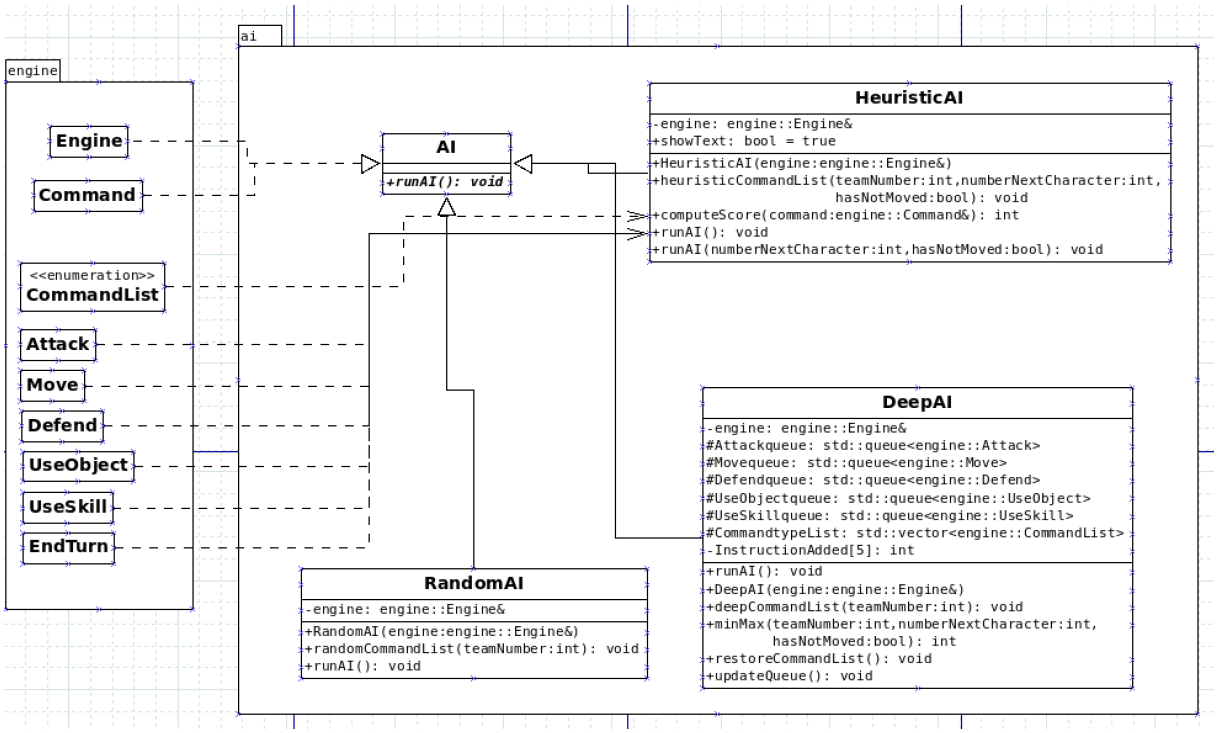

FIGURE 5.1 – Aperçu de ai.dia avec HeuristicAI et DeepAI

Chapitre 6

Modularisation

6.1 Organisation des modules

6.1.1 Répartition sur différents threads

On isole le moteur de jeu sur un thread qui lui est dédié. Le rendu reste sur le thread principal par limitation de la librairie SFML. Sont partagées entre ces deux modules la liste des commandes et les notification de mise à jour de rendu.

La liste de commandes est la seule ressource critique dans le fonctionnement et il est nécessaire de empêcher le thread du moteur de jeu de modifier la liste de commandes tant que le rendu n'est pas effectué. Pour le moment, l'operation de mise à jour dans le thread de moteur de jeu est suspendu lorsque l'on réalise une opération de rendu.

On utilise des notifications de rendu pour savoir lorsqu'une opération de rendu est en cours et que l'on ne peut modifier la liste des commandes.

6.2 Enregistrement et chargement des données

6.2.1 Enregistrement des données

Le jeu comportant de nombreux éléments aléatoires, on a dû réaliser dans un premier temps des fonctions permettant de créer une description de ses caractères sous forme de string. Puis on a réalisé les fonctions inverses qui permettent de définir ses éléments à partir des String ainsi obtenues.
On réalise ensuite une description en string des actions à l'aide des actions retenues pour l'exécution du retour en arrière.
Les différentes string sont ensuite assignées à des tags lors de la création du fichier replay.txt de type json.

6.2.2 Chargement des données

Le chargement d'une sauvegarde d'une partie se déroule en plusieurs étapes. La première est l'ouverture du fichier replay.txt.
On initialise alors le moteur du jeu à l'aide des strings correspondant à la carte et aux

différents personnages. La carte est recréée à l'aide des types de tuiles et des différentes hauteurs enregistrées dans le string. Les personnages sont enregistrés en fonction de leur classe et de leur race, ils sont décrits dans le string dans leur ordre de l'équipe, étant donné que le nombre de personnages par équipes et le même on a besoin de préciser que le nombre d'équipes.
On réalise ensuite les différentes actions de chacun des tours enregistrés dans les strings des tours, ainsi puisque les personnages ont les mêmes caractéristiques, on n'a pas besoin de renseigner des détails comme la vie des personnages puisque celle-ci redeviendra la même.

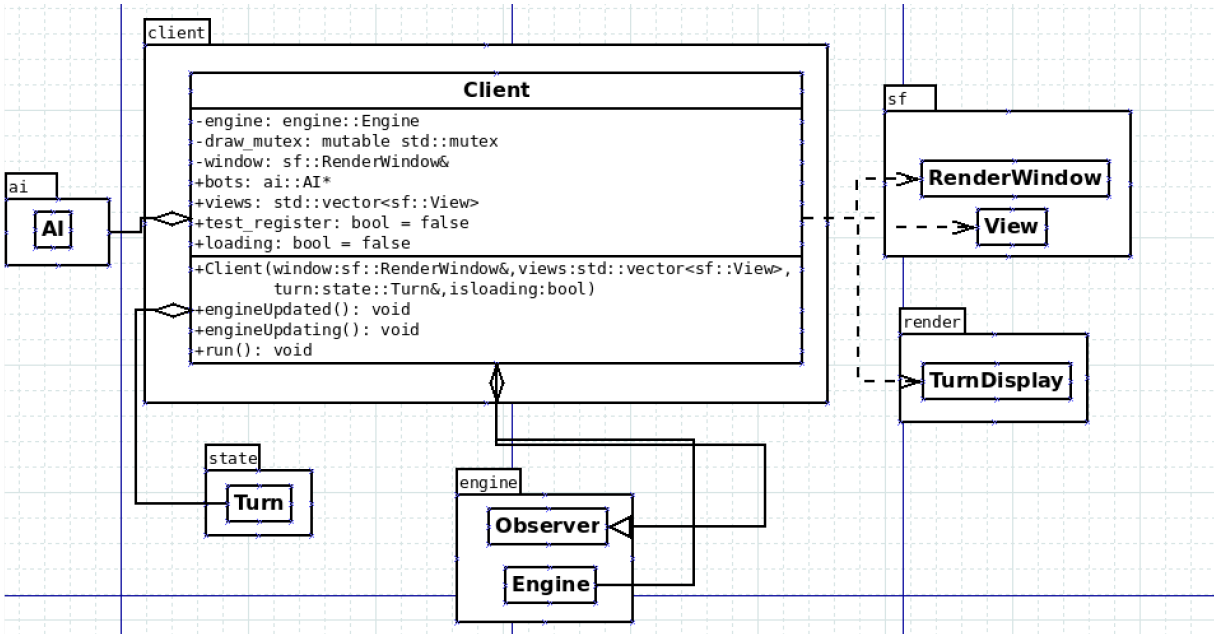

FIGURE 6.1 – Aperçu de client.dia

6.3 Réalisation d'une WebAPI

6.3.1 Description

On réalise une API REST afin de synchroniser plusieurs joueurs sur différentes machine dans une même partie.

On réalise dans ce livrable un service dédié à la gestion des joueurs dans le lobby. A savoir ajouter un joueur au lobby, retirer un joueur du lobby, lister les joueurs dans un lobby.

Le service player gère les requêtes suivantes :

- **Requete GET** /player/<id>
  Retourne pour l'id corresponant le nom du joueur et le booléen inlobby qui est à true quand le joueur n'est pas en partie.
- **Requete GET** /player/all
  Retourne la liste complète des joueurs dans le lobby avec le nom et le booléen inlobby
- **Requete PUT** /player
  On ajoute un joueur à la liste de joueurs du lobby. Le nom du joueur est placé dans le body de la requete.
  Retourne l'id de ce nouveau joueur dans la liste du serveur si le lobby n'est pas plein, sinon retourne un message d'erreur indiquant que le lobby est plein.
- **Requete POST** /player
  Analogue à la précédente, la clé-valeur "req" :"POST" est ajoutée au début du body de la requete.
- **Requete DELETE** /player/<id>
  On retire le joueur correspondant à l'id donné de la liste de joueurs.
- **Requete POST** /player/<id>
  Analogue à la précédente, on ajoute la string "disconnect" au body. On retire le joueur correspondant à l'id donné de la liste de joueurs.

6.3.2 Conception Logicielle

Le diagramme des classes UML C++ pour le serveur est visible en Figure 6.2.

Classe Game : Elle contient la liste de joueurs du lobby, avec des opérations CRUD sur la liste de joueurs (getPlayer,addPlayer,removePlayer,setPlayer)

Classe Player : Elle contient les éléments caractériques d'un joueur : son ID, son nom et le booleen inlobby qui indique si le joueur est en partie ou non.

Classe ServiceManager : Elle est la classe qui s'occupe de gérer les services.
La méthode **registerService** permet d'enregistrer un service pour qu'il puisse être utilisé plus tard lors de la gestion d'une requête.
La méthode **findService** permet de trouver le service associé à la requête reçue par le serveur : un service étant associé à une ressource, il suffit de lire la ressource demandée dans l'url pour trouver le service correspondant.

La méthode **queryService** qui va orienter la requête vers un type de réponse en fonction de la méthode de la requête et le contenu de son body pour les requêtes POST.

Classe Service : Elle est la classe mère pour les autres services tels que PlayerService et CommandService.

Classe PlayerService : Elle gère les requêtes présentées plus haut concernant l'ajout, la modification et la supression de joueurs dans la liste de joueurs du lobby.

Classe ServiceException : Elle permet de gérer les exceptions et de retourner le message d'erreur.

Classe HttpStatus : Elle est l'énumération des codes HTTP, utilisé dans les réponses aux requêtes.

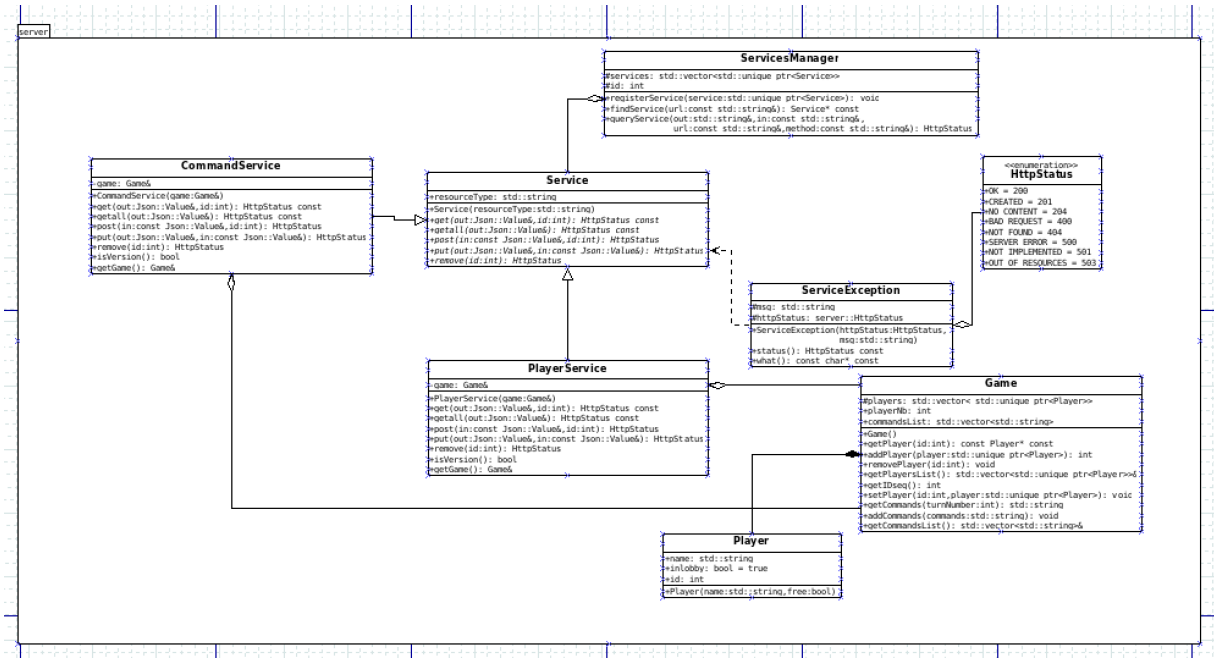

FIGURE 6.2 – Aperçu de server.dia