

Projet Logiciel Transversal

ENSEAⁱ :
Evil Numeric System : Eldritch Artificial Intelligence

Khames Abdelmajid

Dia Mamadou

Le Guen de Kerneizon Paul

Chelki Adam

ENSEA 2021-2022



Table des matières

1 Présentation Générale	1
1.1 Archétype	1
1.2 Règles du jeu	1
1.2.1 Mécanique de Pierre-Feuilles-Ciseaux	1
1.2.2 Personnages	2
1.2.3 Déroulement d'une partie en solo	2
1.2.4 Déroulement d'une partie multijoueur	3
1.3 Ressources	3
2 Description et conception des états	5
2.1 Description des états	5
2.1.1 État du terrain	5
2.1.2 État en combat	5
2.2 Conception Logiciel	6
2.2.1 Diagramme de classe des éléments du jeu	6
2.2.2 Diagramme de classe des états	7
2.2.3 Diagramme complet	8
3 Rendu : Stratégie et Conception	9
3.1 Stratégie de rendu d'un état	9
3.2 Conception logiciel	12
4 Règles de changement d'états et moteur de jeu	13
4.1 Règles	13
4.1.1 Déroulement d'un combat	13
4.1.2 Mathématiques du jeu	13
4.1.3 Exemple : Champion IS	14
4.2 Conception logiciel	15
5 Intelligence Artificielle	17
5.1 Stratégies	17
5.1.1 Stratégie aléatoire	17
5.1.2 Conception logiciel	17
5.1.3 Stratégie heuristique	17
5.1.4 Conception logiciel	19
5.1.5 Stratégie avancée	19
5.1.6 Conception logiciel	20
6 Modularisation	24
6.1 Organisation des modules	24
6.1.1 Exécution du jeu dans un thread séparé	24
6.1.2 Conception logiciel	25

1 Présentation Générale

1.1 Archétype

Dans ce projet, nous réaliserons un jeu au tour par tour en C++. Nous avons donc décidé de nous inspirer des mécaniques du jeu mobile **Raid : Shadow Legends**, c'est-à-dire un jeu se basant exclusivement sur l'affrontement entre divers champions ou des boss, qu'ils soient contrôlés par l'IA ou par un autre joueur.



Illustration du système de combat du jeu **Raid : Shadow Legends**

La thématique du jeu aura lieu dans l'ENSEA, où le joueur incarnera des personnages représentants les Options disponibles en 3A, les attaques, statuts ainsi que les objets feront référence à des thématiques étudiées à l'ENSEA ou à des éléments de la vie étudiante. Réaliser un jeu ayant lieu dans l'ENSEA nous permettra d'ajouter une touche d'originalité à un système déjà bien connu, et d'apporter notre propre empreinte aux ressources visuelles.

Le jeu inclura une sélection de personnages avec des statistiques et des archétypes (tank, soutien, mage,...) ainsi qu'un élément appelé **Majeur** suivant une mécanique de Pierre-Feuille-Ciseaux.

1.2 Règles du jeu

1.2.1 Mécanique de Pierre-Feuilles-Ciseaux

Chaque personnage du notre jeu-vidéo est défini par une **Majeur** parmi une liste de 5 Majeurs :

- Majeur Elec
- Majeur Info
- Majeur Auto
- Majeur Signal
- Majeur Sciences-Humaines

Voici le tableau montrant les forces et faiblesses de chaque Majeur, chaque attaque sera amélioré ou réduite en fonction de la **Majeur** de la cible :

Attaquant / Cible	Elec	Info	Auto	Signal	Sciences-Humaines
Elec	×1	×1	×0.5	×2	×1
Info	×1	×1	×2	×0.5	×1
Auto	×2	×0.5	×1	×1	×1
Signal	×0.5	×2	×1	×1	×1
Sciences-Humaines	×1	×1	×1	×1	×2

1.2.2 Personnages

Avant chaque partie, le joueur devra choisir quatre champions parmi une sélection prédéfinis de 9 champions listés ci-dessous :

- Champion **IS** : Majeur Info
- Champion **SIA** : Majeur Signal
- Champion **RT** : Majeur Signal
- Champion **AEI** : Majeur Auto
- Champion **MSC** : Majeur Auto
- Champion **ESE** : Majeur Elec
- Champion **EVE** : Majeur Elec
- Champion **ESC** : Majeur Elec
- Champion **Audencia** : Majeur Sciences-Humaines

A la fin de la partie, le joueur sera amené à affronter un *boss final*, ce boss ne possède pas de **Majeur** fixe, en effet il possédera une ou plusieurs attaques pour chaque **Majeur** qui, une fois lancée, lui attribuera la **Majeur** correspondante. L'équipe du joueur devra donc être polyvalente pour pouvoir affronter le boss.

1.2.3 Déroulement d'une partie en solo

Une partie est composée de quatre arènes que l'on nommera *Arène 1*, *Arène 2*, *Arène 3* et *Arène 4*. Au début de la partie, le joueur choisit le champion avec lequel il veut commencer.

1. Dans l'*Arène 1*, le joueur affrontera successivement trois champions parmi les huit restants, une fois l'*Arène 1* complétée les trois champions rejoignent l'équipe du joueur.
2. L'*Arène 2* est elle aussi composée de trois combats successifs face à trois champions sur les cinq restants, mais une fois l'*Arène 2* complétée, le joueur pourra remplacer l'un de ses quatre champions parmi l'un des trois champions vaincus dans l'*Arène 2*.
3. L'*Arène 3* quand à elle est composée des deux derniers champions et le joueur pourra de même, remplacer un de ses quatre champions parmi les 2 vaincus.
4. Une fois les 3 premières *Arène* complétées, le joueur va désormais affronter le *boss final* dans l'*Arène 4* avec son équipe de 4 champions.

La suite de champions que le joueur va affronter est générée en début de partie. Elle peut être purement aléatoire, mais être aussi calculée intelligemment : par exemple, si le joueur choisit en début de partie un champion IS, le jeu peut décider de lui faire affronter en début de partie des champions avec des vulnérabilités de la majeure Info pour que le début de partie soit plus facile. Enfin, il est à noter que le jeu fonctionne sur la base d'une partie unique : si le personnage du joueur meurt, la partie recommence depuis le début.

Voici un schéma explicatif montrant le déroulé d'une partie :

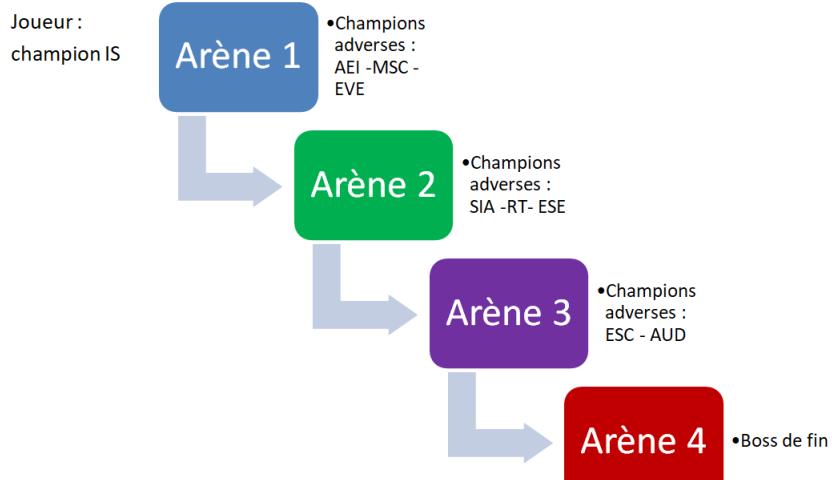


Schéma d'une partie type

1.2.4 Déroulement d'une partie multijoueur

Chaque joueur choisira 4 champions parmi les 9 proposés, ils s'affronteront ensuite directement dans un combat en 4v4.

1.3 Ressources

L'idée générale du rendu est d'avoir un aspect "rétro" 16 bits, dans le même genre que les RPG du début des années 1990 (Final Fantasy 5, Dragon Quest...). Pour cela, nous allons utiliser des ressources sous licences libres, dont la seule obligation est de mentionner les auteurs en crédits. La thématique du jeu étant centrée sur l'ENSEA, et le jeu prenant part dans ses locaux, on utilisera des sprites ayant une consonance moderne. De même, pour que nous ayons des décors fidèles à l'école, nous créerons des sprites spécifiques à l'école (logo...) en utilisant un filtre 16 bits, donnant ainsi aisément à une image un aspect "rétro".

L'interface utilisateur (IU) sera quant à elle rendue par la librairie graphique, et sera uniquement interfacée au moyens de la souris.



A gauche : Un exemple de l'aspect du rendu du jeu (Final Fantasy 5)

A droite : Une image de la rue filtré en 16 bits, qui pourrait être un décor d'arène

Pour le rendu des sprites du joueur et de ses coéquipiers, les sprites de personnage suivant seront utilisés (issus de la collection de sprites *Modern Interior* du créateur *LimeZu*)



Sprite du joueur et de ses coéquipiers

2 Description et conception des états

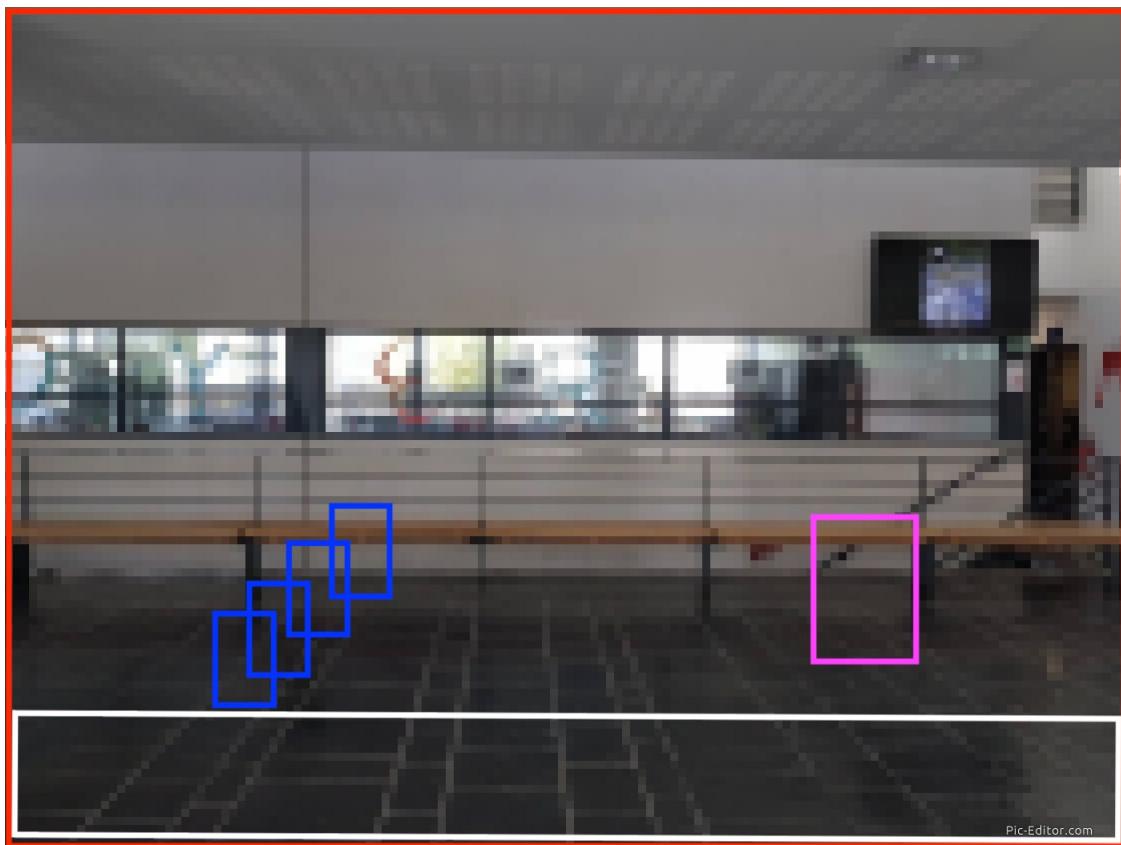
Maintenant que nous avons une description claire de notre jeu, nous pouvons passer à la partie **Unified Modeling Language**, pour modéliser plus formellement notre jeu.

2.1 Description des états

Nous commençons tout d'abord par la description de tous les états de notre jeu ainsi que de nos personnages.

2.1.1 État du terrain

Chaque élément est fixe à l'écran. A l'écran, on aura un ensemble d'emplacements fixe à l'écran. Chaque emplacement possède des coordonnées (x,y) et un numéro pour indexer les positions.



Rouge : Emplacement 0 : Image de fond du combat

Bleu : Emplacement 1.1 ; 1.2 ; 1.3 ; 1.4 : Équipe du joueur

Rose : Emplacement 2 : Ennemi

Blanc : Emplacement 3 : Interface

La souris, ou curseur, est le seul élément mobile présent dans le jeu, elle a donc une position variable et possède deux états : *inactif* et textitactif, l'état *inactif* représente l'état par défaut de la souris et l'état *actif* représente un état ponctuel qui est activé lorsque le joueur appuie sur la souris

2.1.2 État en combat

Le jeu se compose de deux états : en combat ou hors-combat.

Tout d'abord, abordons la partie combat, en combat, le jeu étant un jeu au tour par tour, nous avons deux états :

- Au tour de l'IA
- Au tour du joueur

Lors de chaque tour, chaque *Character* (champions alliés ou ennemi) possède plusieurs états :

- Vivant
- Mort
- Sélectionné
- En attente

Les états *en attente* et *sélectionné* ne sont disponibles que lorsque le *Character* est dans l'état vivant. Lorsqu'un *Character* est *en attente*, qu'il soit du joueur attendant/ayant passé son tour ou de l'IA attendant le tour de l'IA, il peut recevoir des dégâts et lorsque ses points de vie descendent à 0, le *Character* passe dans l'état *Mort*. Lorsqu'un *Character* est mort, il ne peut être *Sélectionné* et n'est plus *En attente*, de plus il ne peut plus recevoir de dégâts mais peut toujours recevoir des actions spécifiques (objets ou sorts de résurrection).

Lorsque le *Character* est sélectionné, l'ordinateur ou le joueur choisit une action, elles sont :

- Choisir une attaque ou un sort
- Ne rien faire
- Utiliser un objet

Une fois l'action effectuée, on passe au champion suivant, et ainsi de suite.

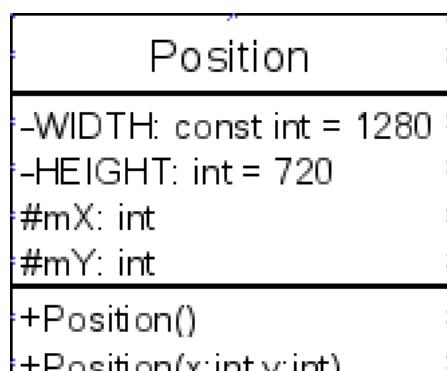
Une fois le combat terminé, il y a deux issues : le *game over* qui représente la défaite du joueur (une fois tout ses *Champions* morts) et va donc renvoyer le joueur au début du jeu pour qu'il puisse recommencer sa partie, ou le *fight win* qui représente la victoire du joueur(une fois l'adversaire mort), lui permettant de continuer ou, s'il s'agit du boss final, de retourner au menu principal pour recommencer une partie.

2.2 Conception Logiciel

2.2.1 Diagramme de classe des éléments du jeu

Abordons maintenant le diagramme de classe, indispensable à la bonne conception d'un programme. Tout d'abord, un point sur la syntaxe de nos futurs classes, attributs et fonctions. Les classes seront appelées "Classe", les attributs seront nommés sous la forme "mAttribut" et seront (sauf exception) privés et les méthodes seront appelés "Méthodes" et seront (sauf exception) public.

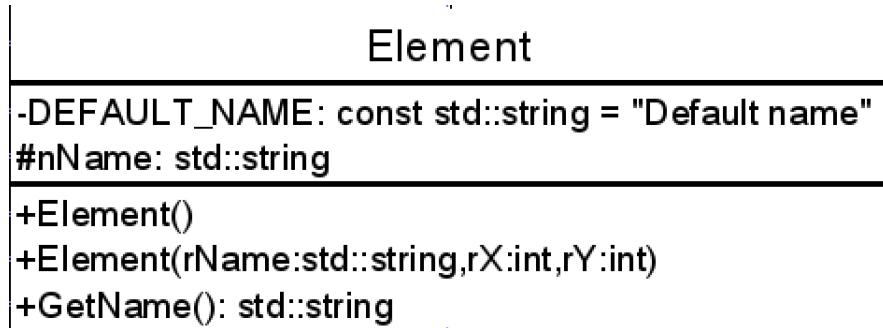
Nous commençons donc avec une classe générale qui sera la classe mère de tous les éléments affichés en jeu, la classe **Position** :



La classe Position

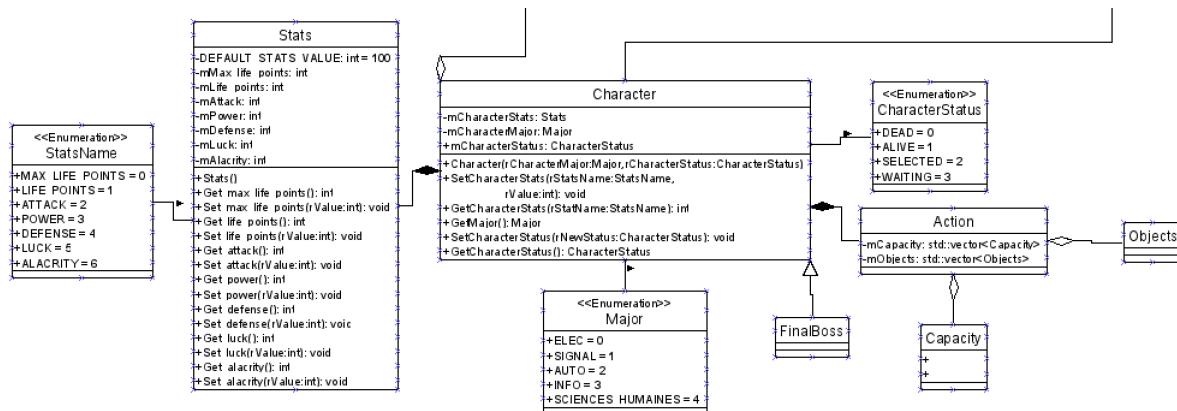
Cette classe dispose donc de deux attributs *protected*, l'attribut mX et mY qui désigne la position de chaque objet **Position**. Ainsi que la taille maximale d'un objet en longueur et en largeur. On a, pour l'instant, les méthodes classiques permettant de construire l'objet et d'obtenir ses attributs.

On peut maintenant écrire une classe qui va permettre de modéliser tous les éléments affichés à l'écran. On appellera de manière naïve cette classe **Element**.



Cette classe définit pour chaque **Element** un nom (en plus de la position héritée par la classe mère), si un nom n'est pas défini un nom par défaut est accordé.

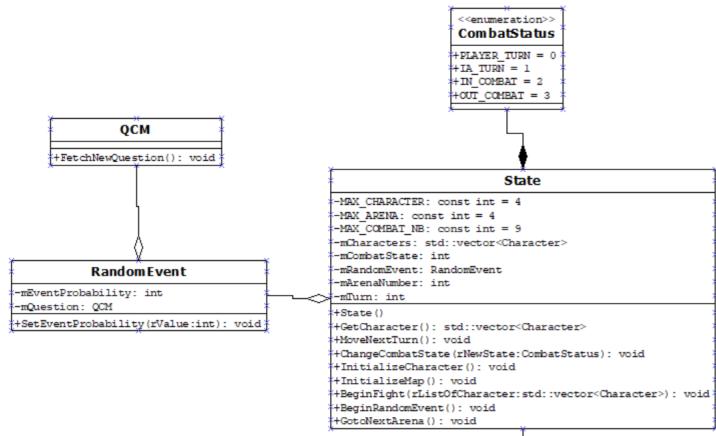
Passons désormais à la classe la plus importante de notre jeu, la classe **Character** qui représentera tous les personnages impliqués dans le jeu (y compris les champions), cette classe hérite de la classe **Element** et est composé des classes **Stats**, **Action**, **CharacterStatuts** et **Major**. Cette classe engendre la classe fille **Final Boss** qui est un champion spécial pouvant changer de majeur.



La classe **Action** utilise une liste d'objets et de capacités (liste qu'il nous reste à définir).

2.2.2 Diagramme de classe des états

Nous avons maintenant décrit tous les éléments affichables du jeu, il reste maintenant à décrire une classe **State**, qui décrit les états du jeu.

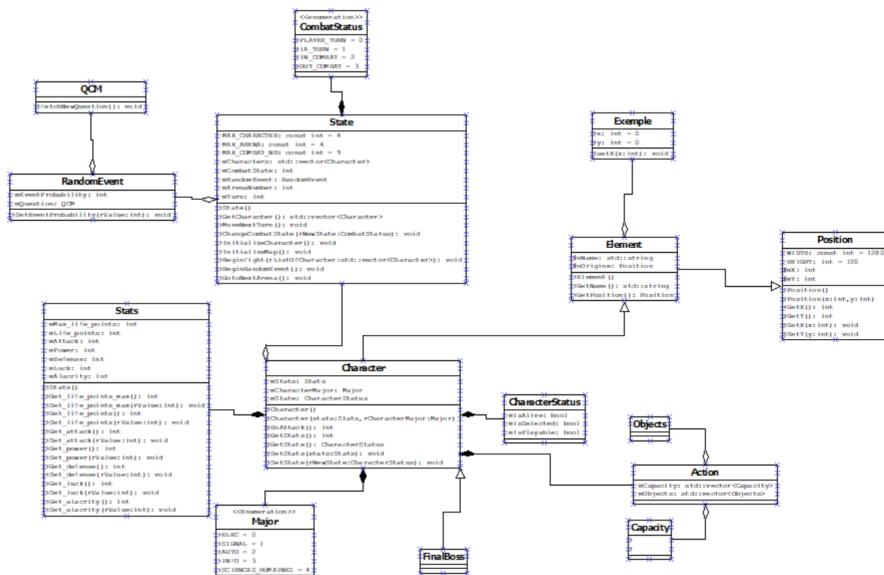


Dans un premier temps, on y retrouve les informations fixes. Il y a le nombre maximum de joueurs jouables par l'utilisateur, le nombre maximal d'arènes, et le nombre maximum de combats. En plus de la liste des éléments et informations fixes établie, on y trouve des attributs qui nous permettent de savoir dans quel état le jeu se situe comme l'état du combat, l'arène en cours ou encore le joueur en train de jouer. Bien évidemment, on trouve aussi les méthodes permettant de connaître les valeurs des attributs, et donc l'état du jeu, mais aussi les méthodes permettant de changer l'état du jeu. On y trouve par exemple la méthode permettant de commencer un combat, ou celle permettant de changer d'arène.

On peut voir que cette classe est composé d'autres classes. En effet, étant donné que le nombre d'état d'un combat est fini et facilement énumérable, faire une classe d'énumération permet de simplifier plusieurs étapes dans le développement du jeu. Concernant les évènements aléatoires, ils ne peuvent être déclenchés durant une phase de combat, donc à l'instar de la classe **CombatStatus**, une classe dédié facilite le développement et les déclenchements des phases aléatoires.

2.2.3 Diagramme complet

Ainsi, en regroupant les deux diagrammes on obtient le diagramme final et complet de notre futur jeu vidéo.



3 Rendu : Stratégie et Conception

Dans cette partie, nous allons expliquer comment nous avons créé notre rendu, et comment passer d'un état à un autre.

3.1 Stratégie de rendu d'un état

Tout d'abord, le rendu est effectué selon deux états :

- En combat : les personnages du joueur et de l'ennemi sont affichés selon une animation dites "repos"
- Hors combat : C'est l'état lorsque l'on passe d'un combat au suivant. La caméra défile vers la droite pour donner une illusion de déplacement. Les personnages du joueur sont affichés selon une animation de course.

Pour calculer les rendus, notre stratégie est assez simple. Grâce à l'utilisation de *SFML*, nous pouvons découper les étapes et les éléments afin d'optimiser les calculs.

La méthode que nous avons choisi consiste à découper un plan en plusieurs *layers* (couches). Le rendu du jeu se base sur trois layers :

- Le background (l'image de fond)
- Les sprites des différents personnages du joueur ou de l'adversaire
- Et enfin l'interface utilisateur (UI) qui affiche différents éléments d'interaction (texte, fenêtre...).

Pour chaque image, le principe repose sur le calcul de l'élément du layer à afficher. Par exemple, le sprite d'un character est animé par le défilement de plusieurs images du personnage successifs.

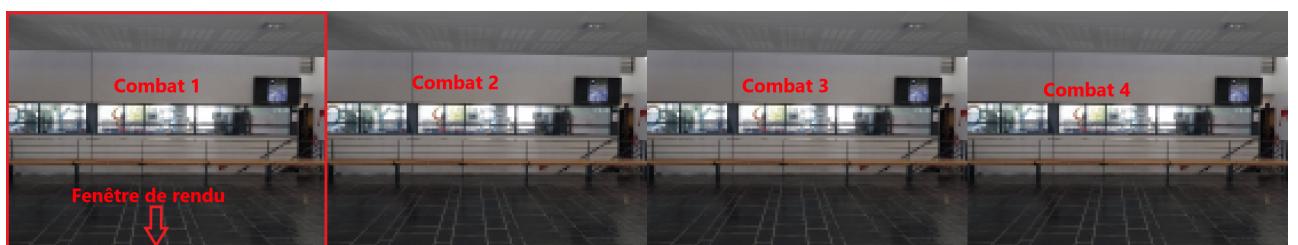


Exemple d'un sprite utilisé. La partie du haut représente les différentes images de l'animation au repos du personnage Celle du bas représente l'animation de course du personnage.



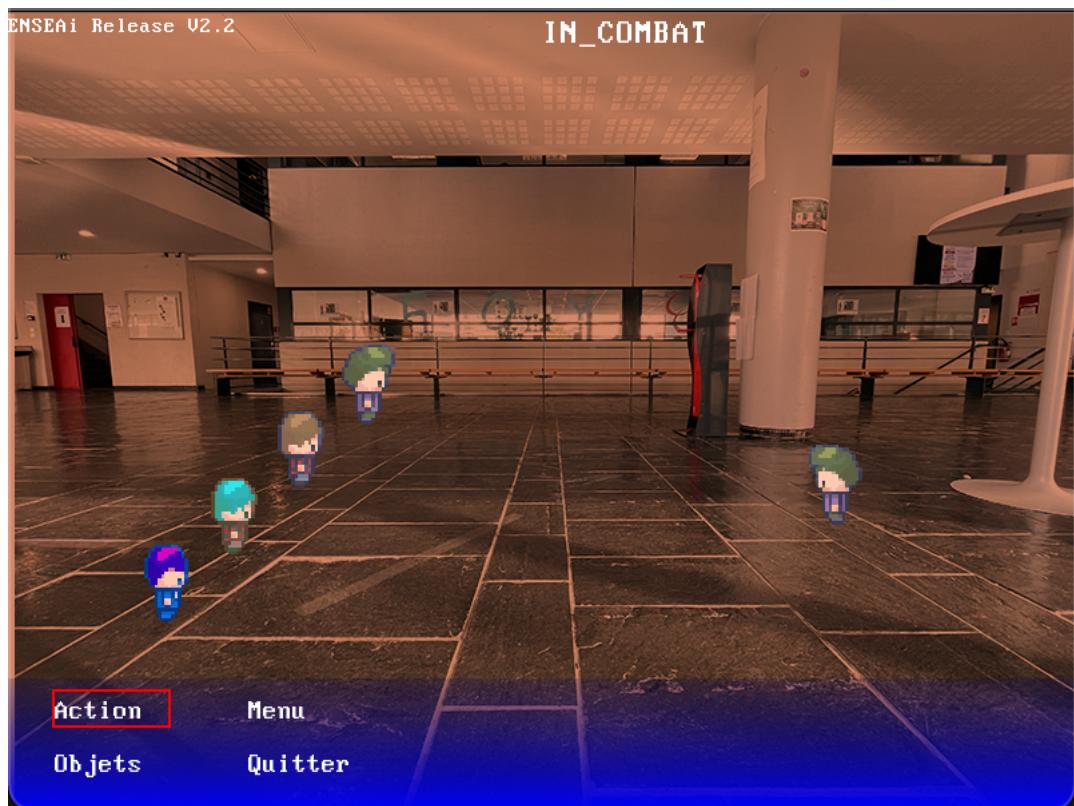
Exemple de background utilisé. Ce background constitue une autre couche à calculer pour le rendu.

Nous avons vu que dans l'état hors combat la caméra défile vers la droite pour donner une illusion de mouvement vers le prochain combat. Pour réaliser cela, on utilise la classe *View* de *SFML* qui permet de déplacer la fenêtre de rendu du jeu. Comme la fenêtre du jeu est d'une dimension de 800x600 pixels, l'idée est d'avoir une image de fond d'une image de dimension 3200x600 (pour quatre combats), dont on déplacera la fenêtre de rendu au fil des combats.

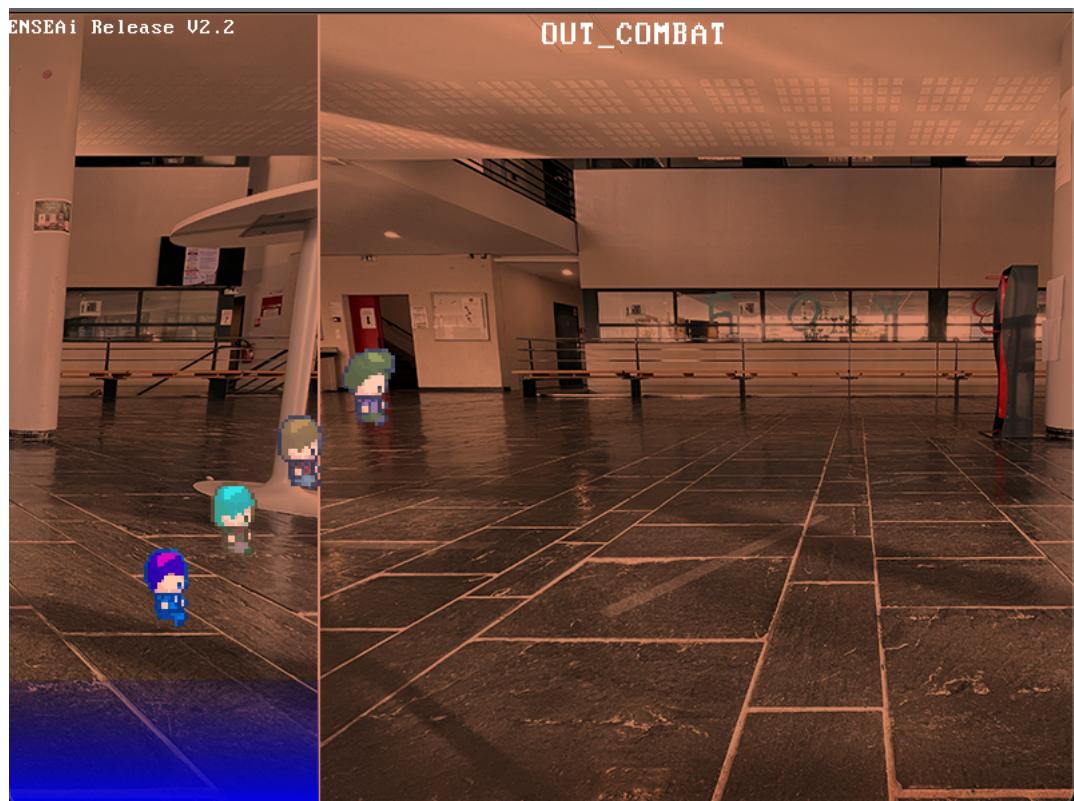


Principe du glissement de la fenêtre de rendu.

On obtient ainsi le rendu suivant :



Le rendu du jeu en combat.



Le rendu du jeu hors combat.

3.2 Conception logiciel

La conception logiciel du rendu est basée sur l'utilisation de trois classes pour chaque layer :

- Une première classe *Surface* permet la gestion du chargement d'un sprite en mémoire, mais aussi leur animation.
- Une seconde classe *UI* gère l'interface utilisateur et l'affichage de texte et de fenêtre à l'écran.
- Une dernière classe *RenderLayer* gère la mise en lien de ces deux classes avec le *namespace state*, et donc la modification du rendu en fonction des états du jeu.

On peut remarquer que chaque classe possède une fonction *draw()*, qui permet le rendu de la layer respective.

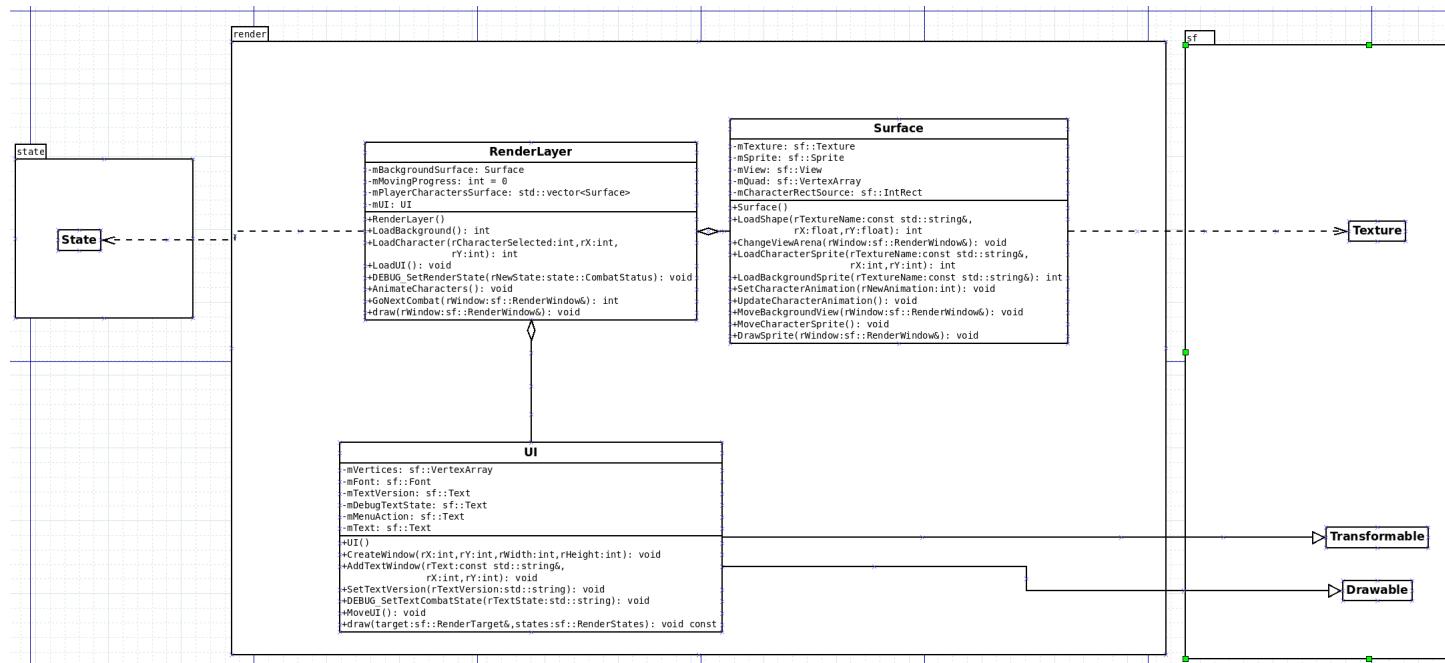


Diagramme de classe du rendu.

4 Règles de changement d'états et moteur de jeu

4.1 Règles

4.1.1 Déroulement d'un combat

Intéressons maintenant aux détails des règles du jeu et à leur implémentation. Le joueur commence la partie avec un personnage dans son équipe. Il affronte ensuite un personnage adverse selon le déroulement suivant :

1. Si le joueur a un personnage vivant et que l'ennemi est toujours vivant, on commence un tour :
 - (a) Le joueur choisit une action pour un personnage de son équipe. Une action peut être une attaque ou un sort du personnage.
 - (b) L'ennemi effectue son action en faisant de même pour son personnage.
 - (c) C'est de nouveau au joueur avec un autre personnage de son équipe.
2. Le combat se termine lorsque tous les personnages du joueur ont leur PV à 0 (dans ce cas le jeu passe en état Game Over) ou que l'ennemi a ses PV à 0. (le jeu passe dans ce cas au combat suivant).

Une fois un combat terminé, le joueur passe au suivant sans checkpoint et ses personnages restent dans l'état dans lequel il a fini le combat.

4.1.2 Mathématiques du jeu

Chaque personnage possède 6 caractéristiques :

- PVmax : Les points de vie maximum du personnage.
- Attack : La statistique d'attaque qui va impacter les ATTACKs du personnage.
- Power : La statistique de puissance qui va impacter les SPELLs du personnage.
- Defense : La statistique de défense du personnage, va impacter sa résistance aux dégâts.
- Luck : La statistique de chance, qui va impacter les coups critiques du personnage.

Avec ces statistiques, le calcul des PV perdus se fera comme suit.

$$PV_{Perdus} = 0.44 * \frac{CC*CM*Deg*Stats}{Defense}$$

- PV_{Perdus} : Le nombre de PV que la cible perd
- CC : 2 si c'est un coup critique, 1 sinon.
- CM : Coefficient Majeur, 2 si l'attaquant a une majeur avantageuse, 0.5 si l'attaquant a une majeur désavantageuse, 1 s'il n'y a aucune relation. (cf 1.2.1).
- Def : Dégats de l'ATTACK ou du SPELL.
- Stats : Prend la caractéristique d'Attak(resp Power) de l'attaquant si c'est une ATTACK (resp SPELL).
- Defense : Prend la caractéristique de défense de la cible.

Pour les statistiques de chaque personnage, on aura plusieurs archétypes :

- Mage : Un personnage orienté dégats via la statistiques Power
- Attaquant : Un personnage orienté dégats via la statistiques Attack
- Tank : Un personnage qui va pouvoir encaisser beaucoup de dégats.
- Soutien : Un personnage qui va pouvoir buff/soigner ses alliés.
- Polyvalent : Un personnage qui va pouvoir faire des dégats via ses statistiques Power et Attack.

Chaque personnage aura au maximum 4 actions, 2 SPELLs et 2 ATTACKs maximum. un Mage aura par exemple 2 SPELLs et une ATTACK et ce sera l'inverse pour un Attaquant. Chaque ATTACK ou SPELL fera un nombre défini de dégats et pourra appliquer un buff (effet bénéfique pour une compétence du personnage) ou un débuff (effet négatif sur une compétence du personnage adverse).

4.1.3 Exemple : Champion IS

Prenons par exemple le champion IS qui est un personnage de type Polyvalent, voici ses statistiques :

PV	80
Attack	140
Power	140
Defense	80
Luck	5

Et ses ATTACKs et SPELLs sont :

- ATTACK :
 - Fais 80 de dégats.
 - Fais 60 de dégats et augmente sa Power de 20.
- SPELL :
 - Fais 80 de dégats.
 - Fais 60 de dégats et augmente son Attack de 20.

4.2 Conception logiciel

L'implémentation du moteur de jeu est basée sur une vérification de conditions des paramètres du jeu selon son état.

- En combat, le jeu est dans l'état *IN COMBAT*.
 1. Le moteur de jeu se met en attente d'une commande à exécuter selon si c'est au joueur ou à l'IA de jouer.
 2. Lorsqu'une commande est reçue, le moteur l'interprète et l'exécute.
 3. Enfin, il passe la main à l'autre joueur.
- Le moteur vérifie ensuite si un des personnages du joueur ou de l'ennemi est mort.
 1. Si tous les personnages du joueur sont morts alors le jeu passe à l'état *GAME OVER*
 2. Si tous les personnages de l'ennemi sont morts alors le jeu passe à l'état *OUT COMBAT* pour passer au combat suivant.

Cette conception nous donne le diagramme de classes suivant :

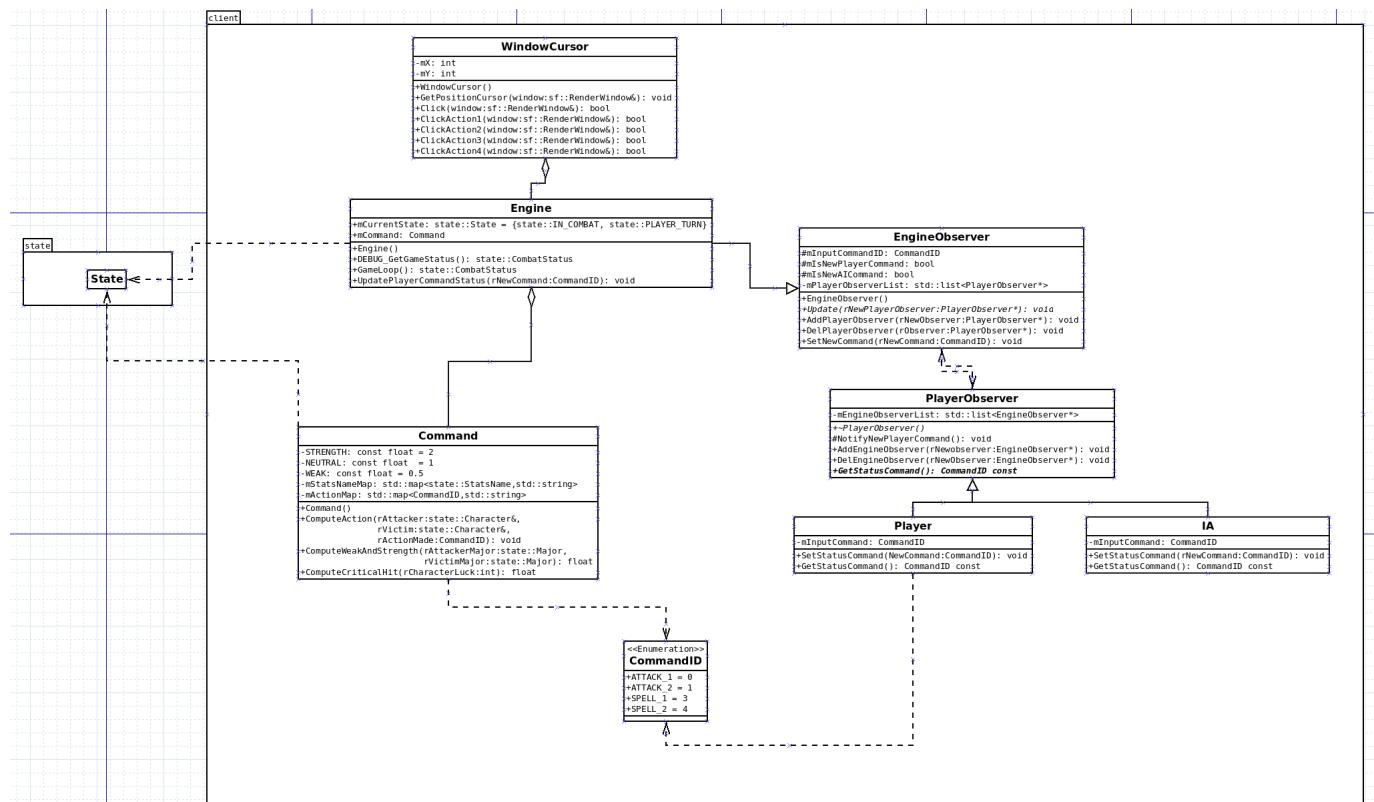
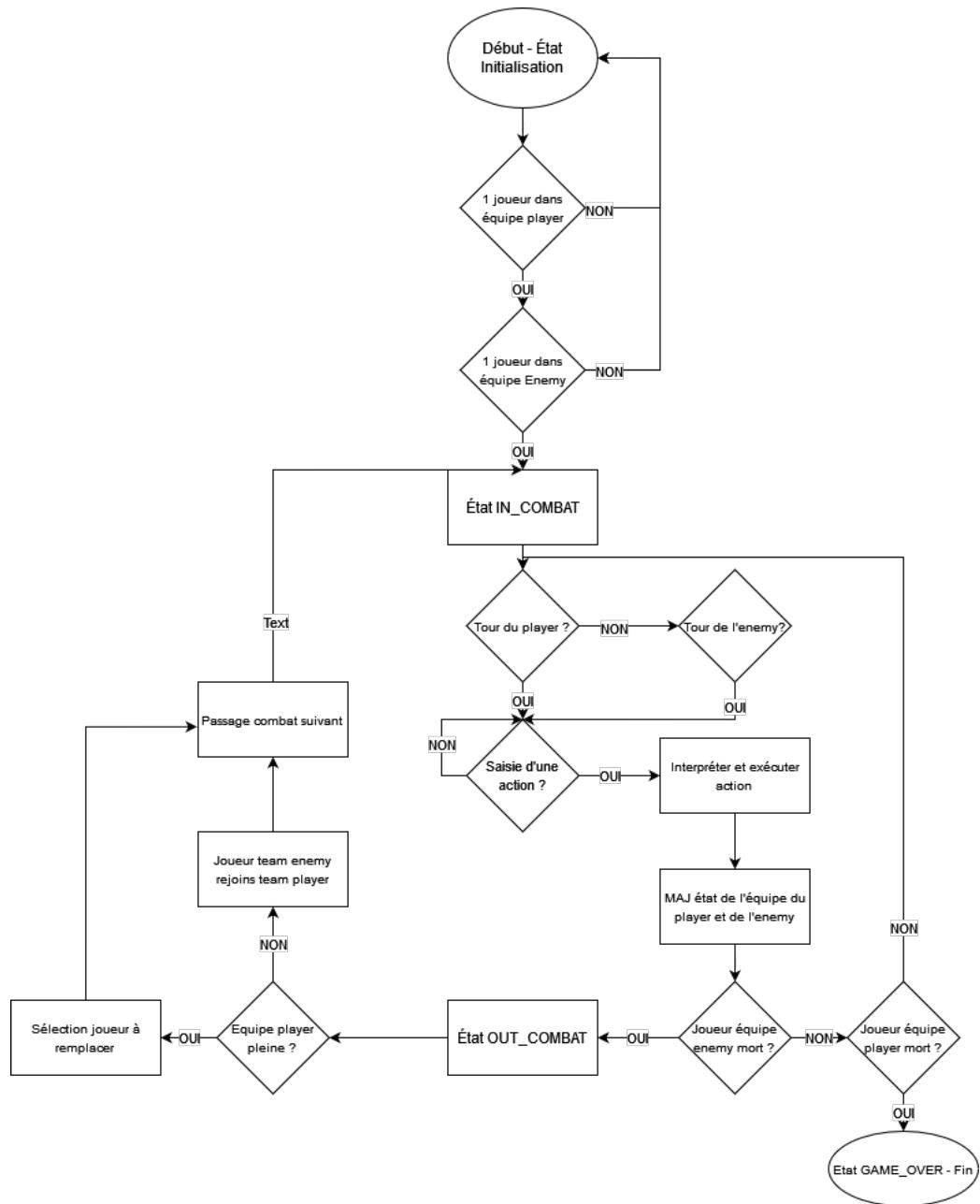


Diagramme de classe du moteur de jeu.

On utilise ici le pattern *observateur* permettant de notifier le moteur de jeu lorsqu'une commande est saisie par le joueur ou l'IA. La classe *Command* permet l'exécution de celle-ci selon les règles du jeu établi.

Le fonctionnement du moteur de jeu peut ainsi se résumer par le diagramme suivant :



Algorithme du moteur de jeu.

5 Intelligence Artificielle

5.1 Stratégies

5.1.1 Stratégie aléatoire

Le principe de cette intelligence artificielle est relativement simple, car il ne repose sur aucune logique particulière.

- Au début d'une partie, une liste aléatoire d'ennemis pour chaque combat est générée. Le joueur peut donc avoir de la chance et avoir un premier combat en sa faveur si la majeure du personnage qu'il choisit est forte contre celle générée aléatoirement pour le personnage ennemi, tout comme l'inverse dans le cas opposé.
- Lorsque c'est à l'IA de jouer, elle décide aléatoirement une action (attaque 1, attaque 2...) parmi celles disponibles pour son personnage. Ce choix n'a donc pas de logique particulière (exemple : IA qui choisit une action qui redonne des PV à son personnage si ils sont bas).

5.1.2 Conception logiciel

L'implémentation de cette stratégie n'a pas de conception particulière au vue de sa simplicité. Les choix aléatoires de l'IA sont ici gérés dans une instance de la classe IA (cf. Diagramme de classe du moteur de jeu p.15).

5.1.3 Stratégie heuristique

Le principe de cette intelligence artificielle repose sur le principe qu'une commande est choisie selon des règles arbitraires établies. La principale distinction avec l'IA aléatoire est que l'IA heuristique a connaissance de l'état du jeu. Elle peut donc prendre des décisions en fonction des différents paramètres de celui-ci.

Les règles heuristiques dépendent selon le personnage joué par l'IA au tour N. A chaque tour, l'IA choisit une de ses actions selon des règles de priorités. On peut ainsi les résumer de la façon suivante :

- **Personnage : IS**
 1. **Action 1** : SPELL, Augmente statistique d'attaque du personnage de 20, fait 60 de dégâts
Condition : Si statistique d'attaque du personnage < 200
 2. **Action 2** : ATTACK, Fait 80 de dégâts
Condition : Dans tous les autres cas
- **Personnage : SIA**
 1. **Action 1** : SPELL, Augmente statistique de chance du personnage de 5, fait 50 de dégâts
Condition : Si statistique de chance du personnage < 25
 2. **Action 2** : SPELL, Diminue statistique de chance de l'adversaire de 5, fait 60 de dégâts
Condition : Si la statistique de chance de l'adversaire > 0
 3. **Action 3** : ATTACK, Fait 80 de dégâts
Condition : Dans tous les autres cas
- **Personnage : RT**
 1. **Action 1** : SPELL, Augmente statistique d'attaque de 10, fait 50 de dégâts
Condition : Si statistique d'attaque du personnage < 200
 2. **Action 2** : SPELL, Diminue statistique d'attaque de l'adversaire de 10, fait 60 de dégâts
Condition : Si la statistique d'attaque de l'adversaire > 0

3. **Action 3** : ATTACK, Fais 80 de dégâts

Condition : Dans tous les autres cas

— **Personnage** : MSC

1. **Action 1** : SPELL, Durant 4 tours, chacun des personnages de l'équipe du joueur gagne 5 PV, fait 40 de dégâts

Condition : Si il y a plus d'un personnage dans l'équipe du joueur

2. **Action 2** : SPELL, Redonne 15 PV au personnage, fait 60 de dégâts

Condition : Si la statistique de vie du personnage < 50%

3. **Action 3** : ATTACK, Fait 70 de dégâts

Condition : Dans tous les autres cas

— **Personnage** : AEI

1. **Action 1** : SPELL, La prochaine attaque du joueur au tour N+1 fera 15% de dégâts en plus

Condition : Si il y a plus d'un personnage dans l'équipe du joueur

2. **Action 2** : ATTACK, Fait 104 de dégâts, mais le personnage perd 40 PV.

Condition : Si la statistique de vie du personnage > 50%

3. **Action 3** : ATTACK, Fait 80 de dégâts

Condition : Dans tous les autres cas

— **Personnage** : EVE

1. **Action 1** : SPELL, Durant 4 tours, chacun des personnages de l'équipe du joueur gagne 10 PV, fait 10 de dégâts

Condition : Si il y a plus d'un personnage dans l'équipe du joueur

2. **Action 2** : SPELL, Redonne 20 PV au personnage, fait 40 de dégâts

Condition : Si la statistique de vie du personnage < 50%

3. **Action 3** : ATTACK, Fait 60 de dégâts

Condition : Dans tous les autres cas

— **Personnage** : ESE

1. **Action 1** : SPELL, Augmente statistique de défense du personnage de 10, fait 60 de dégâts

Condition : Si statistique de défense du personnage < 200

2. **Action 2** : SPELL, Diminue statistique de chance de l'adversaire de 5, fait 60 de dégâts

Condition : Si la statistique de chance de l'adversaire > 0

3. **Action 3** : ATTACK, Fait 70 de dégâts

Condition : Dans tous les autres cas

— **Personnage** : ESC

1. **Action 1** : ATTACK, Fait 110 de dégâts

Condition : Dans tous les cas

— **Personnage** : Audencia

1. **Action 1** : SPELL, Durant 4 tours, chacun des personnages de l'équipe du joueur gagne 20 PV, fait 10 de dégâts

Condition : Si il y a plus d'un personnage dans l'équipe du joueur

2. **Action 2** : SPELL, Redonne 30 PV au personnage, fait 20 de dégâts

Condition : Si la statistique de vie du personnage < 50%

3. **Action 3** : ATTACK, Fait 60 de dégâts

Condition : Dans tous les autres cas

5.1.4 Conception logiciel

L'implémentation de cette stratégie est faite dans un nouveau namespace *ai*, regroupant la stratégie aléatoire et heuristique. Le fonctionnement repose sur l'utilisation d'un pattern *observateur*. Lorsqu'une nouvelle commande est saisie par l'IA, le moteur de jeu en est signalé, et découpe l'exécution associée.

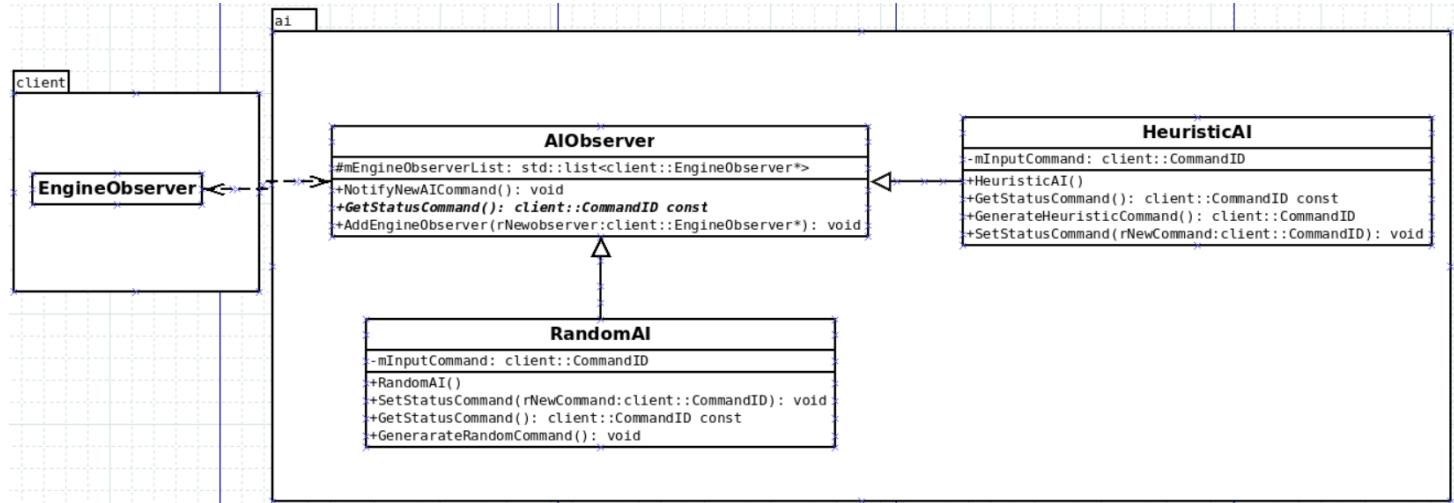
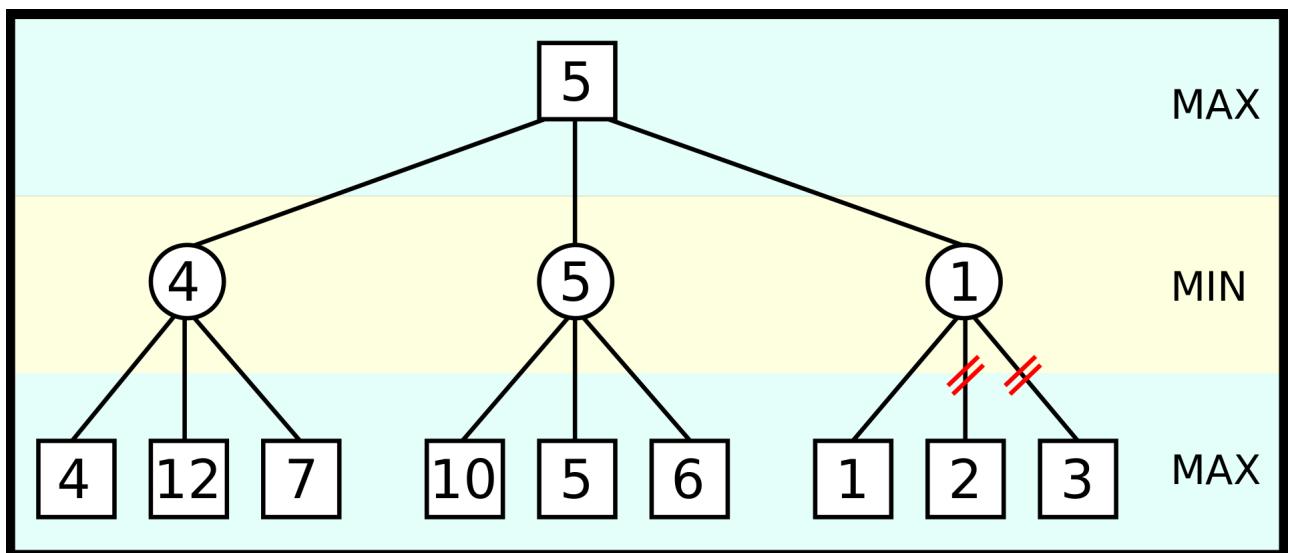


Diagramme de classes de l'IA

5.1.5 Stratégie avancée

A la différence des deux premières intelligences artificielles, cette dernière repose sur un fonctionnement où elle va chercher à réaliser à chaque tour le meilleur coup possible. Pour implémenter cette fonctionnalité, nous allons nous baser sur un algorithme d'arbre de recherche, et plus précisément sur l'algorithme de la théorie des jeux *minimax*.

L'idée de fonctionnement de cet algorithme est la suivante : A chaque tour IA, un arbre de recherches est établi suivant les actions possibles du personnage joué, sur une profondeur de recherche arbitrairement choisie. L'IA joue le rôle dit "MAX" où elle va chercher à maximiser ses gains à chaque tour. Dans l'exemple ci-dessous, elle choisit la valeur la plus élevée des noeuds à sa disposition. Le joueur (humain ou IA) joue le rôle dit "MIN" où il cherche à minimiser ses pertes. Dans le même exemple, l'IA MIN choisit la valeur la plus faible des noeuds à sa disposition.



Exemple simple du fonctionnement de l'algorithme minimax.

La difficulté est d'adapter cet algorithme au fonctionnement de notre jeu. Si chaque noeud représente, une action faite par un personnage (ATTACK 1, etc.), comment peut on numériquement qualifier l'efficacité d'un noeud ? Pour y répondre, il suffit d'analyser la manière dont doit fonctionner l'IA MIN et l'IA MAX :

- IA MAX (ennemi/adverse) : son but est de faire perdre le joueur humain ou l'IA MIN. Pour cela elle va choisir les actions permettant de tuer le plus rapidement les personnages du joueur, et d'arriver à un état du jeu GAME OVER.
- IA MIN (joueur humain ou IA) : son but est de minimiser ses pertes en empêchant que les personnages de son équipe meurt, et d'accéder à l'état de jeu GAME WIN. En effet, le nombre de personnages présents dans l'équipe du joueur est proportionnel à ses chances d'avancer le plus loin dans le jeu avant un GAME OVER.

On peut donc synthétiser l'efficacité d'un noeud selon le tableau suivant :

Type de noeud	Valeur
État GAME OVER (noeud terminal)	10^9
Mort d'un personnage joueur	100 000
Perte de PV	Valeur des PV perdus $\times 1000$
Récupération de PV	Valeur des PV récupérés $\times 100$
Passage au combat suivant	0
État GAME WIN (noeud terminal)	-10^9

On peut donc remarquer d'après ces valeurs que l'IA MAX privilégiera les valeurs hautes, alors que l'IA MIN privilégiera avant tout les valeurs faibles.

5.1.6 Conception logiciel

La conception de l'IA avancée repose sur trois fonctionnalités à implémenter :

- A partir de l'état du jeu sur lequel le tour est joué par l'IA, construire un arbre de recherche de profondeur arbitraire listant toutes les possibilités d'actions.
- A partir de cet arbre généré, exécuter un algorithme minimax afin de connaître la commande optimale à exécuter au tour actuel.
- Réinitialiser l'arbre de recherche pour la construction d'un nouvel arbre au tour suivant.

On utilisera ainsi le diagramme de classes suivant :

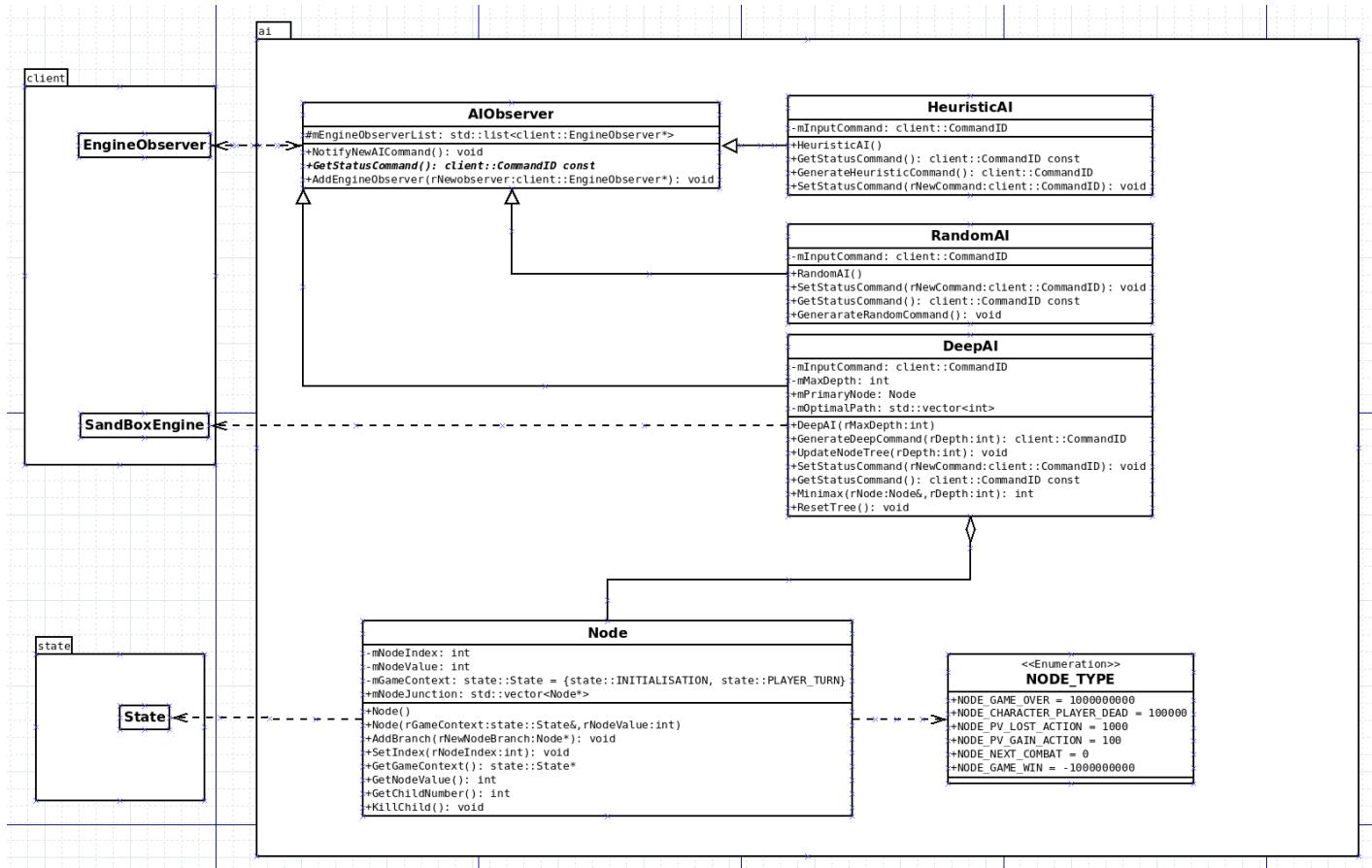
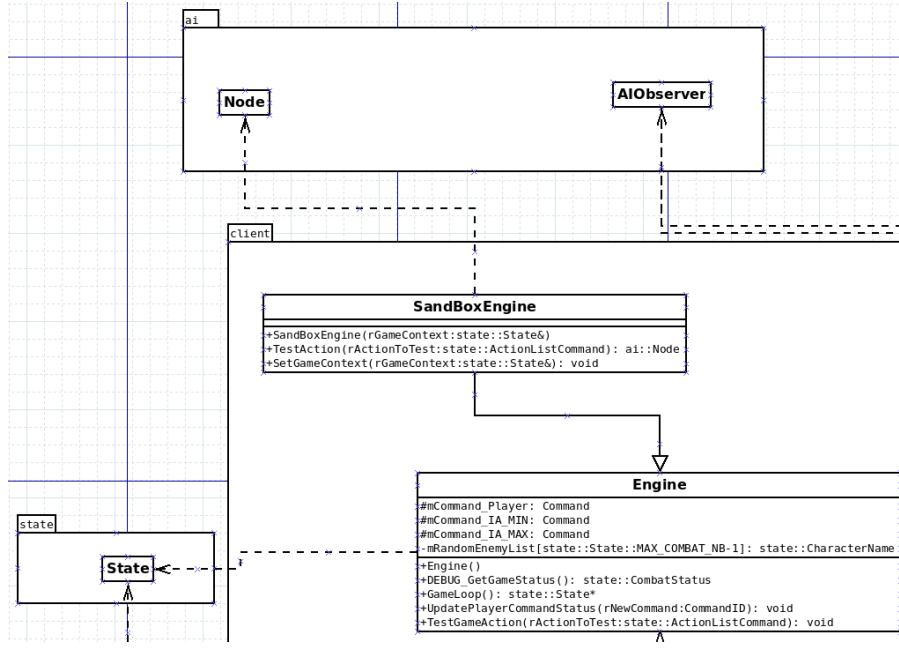


Diagramme de classes de l'IA avancée

La stratégie avancée ajoute deux nouvelles classes dans le namespace `ai` :

- La classe `DeepAI`, qui représente l'objet IA avancée au sein du jeu. Chaque objet est nécessairement instancié avec un paramètre de profondeur de son arbre de recherche.
- La classe `Node` représente un noeud de l'arbre de recherche, et possède un contexte du jeu selon l'action avec laquelle son parent l'a créé. Chaque noeud possède une valeur d'efficacité qui est définie selon le tableau p.20.

De même, une classe `SandBoxEngine` est ajoutée au namespace `client`. C'est une version héritée du moteur de jeu, qui permet de tester diverses commandes du jeu en mode dit "bac à sable", c'est à dire que cela ne va jamais influencer sur l'état du jeu en cours d'exécution.

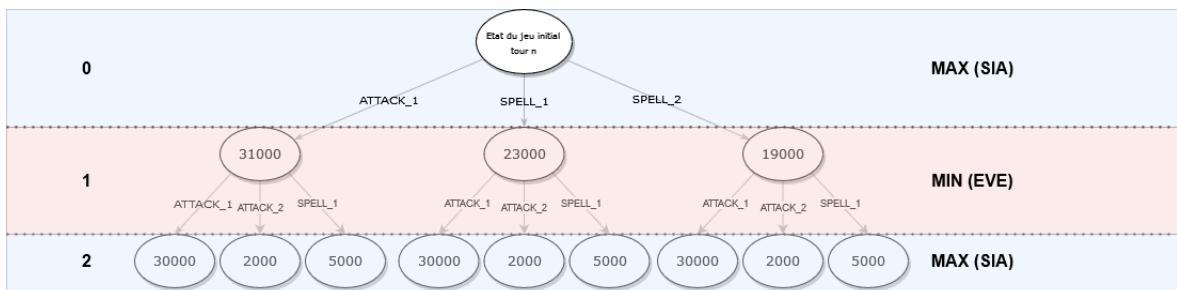


Classe SandBoxEngine

Construction d'un arbre de recherche

La première tâche à réaliser pour avoir une commande optimale est de construire l'arbre de recherche de tous les états du jeu possibles à partir de celui existant au tour en cours. La méthode qui réalise cela est la méthode *UpdateNodeTree*. Pour chaque profondeur d'arbre, elle va tester les actions possibles de chacun des noeuds présents à cette profondeur, et sauvegarder chaque résultat (l'état du jeu résultant) dans un noeud enfant de la classe *Node*. La méthode s'exécute jusqu'à atteignement de la profondeur choisie, ou bien atteignement d'une profondeur où il n'y a que des noeuds terminaux.

L'exemple suivant montre un arbre généré avec une profondeur de 2. L'IA MIN (qui simule le joueur) commence avec un personnage de type EVE, l'IA MAX (ennemi) commence avec un personnage de type SIA. A noter ici que la valeur des noeuds est leur valeur propre, résultant d'une commande, sans passage de l'algorithme minimax.



Arbre de recherche des actions de jeu possibles de profondeur 2

Application de l'algorithme minimax

Avec l'arbre d'états du jeu généré, il est maintenant possible d'appliquer l'algorithme minimax, de façon à obtenir une commande pour l'état du jeu optimale. Ce dernier est implémentée dans la méthode *Minimax* de la classe *DeepAI*, et fonctionne selon le pseudo-code suivant :

```

function Minimax(node, depth) is
    if depth = max_depth or node is a terminal node then
        return the value of node

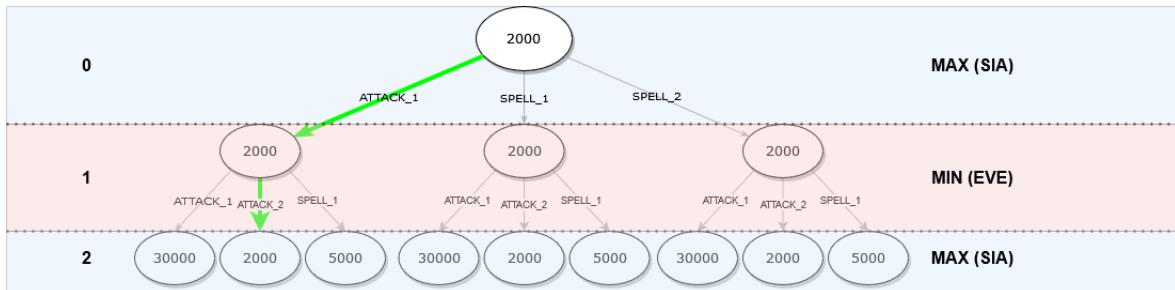
    if maximizingPlayer then
        max_value := -∞
        for each child of node do
            node_eva = Minimax(child, depth + 1)
            max_value := max(max_value, node_eva)
        return max_value

    else (* minimizing player *)
        min_value := +∞
        for each child of node do
            node_eva = Minimax(child, depth + 1)
            min_value := min(node_eva, min_value)
        return min_value

```

A chaque exécution de Minimax, le contexte de l'état du jeu contenu dans le noeud est récupéré, permettant notamment de connaître qui doit jouer.

En reprenant l'exemple précédent, on obtient ainsi l'arbre suivant, après passage de l'algorithme Minimax, (le chemin optimal est surligné en vert) :



Arbre de recherche final après passage de l'algorithme Minimax

Réinitialisation de l'arbre

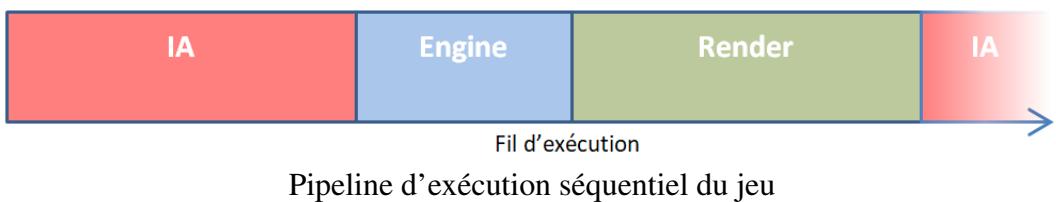
L'arbre généré nous donne une commande optimale pour le tour n, et il sera nécessaire d'en générer un nouveau au tour n+1. C'est donc pour cela qu'il est nécessaire de le réinitialiser. Pour cela, on utilise la méthode *ResetTree* de la classe *DeepAI*. Son fonctionnement est relativement simple : elle parcourt l'arbre jusqu'à la dernière profondeur où des noeuds ont des enfants. A partir de celle-ci on supprime chacun des enfants, puis on remonte progressivement jusqu'à supprimer les enfants du noeud racine.

6 Modularisation

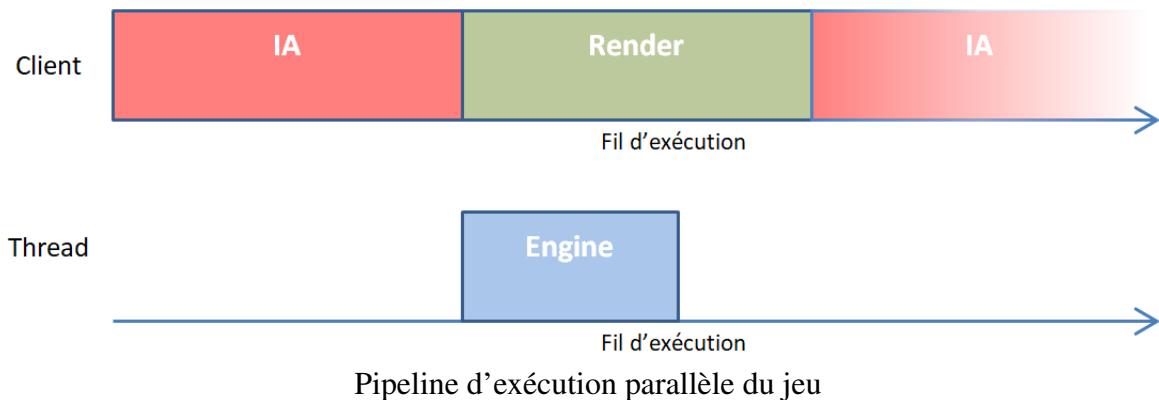
6.1 Organisation des modules

6.1.1 Exécution du jeu dans un thread séparé

Jusqu'à présent le pipeline de fonctionnement du jeu était séquentiel. L'ensemble des traitements, que ce soit le calcul du rendu, la génération d'une commande par l'IA ou l'exécution du moteur de jeu se font à la suite des autres. Le principal problème de cette solution, est que si l'un des étages du pipeline met du temps à être exécuté, c'est l'ensemble du reste du jeu qui est bloqué. Par exemple, dans l'état *IN COMBAT*, si le moteur de jeu met du temps au traitement d'une commande, la mise à jour du rendu sera bloquée, provoquant un ralentissement de l'aspect du jeu voir une impression de blocage. De même le moteur de jeu est appelé à chaque itération du thread principal, même si il n'y a pas de commande à traiter (par exemple lorsque l'on attend une commande d'un joueur humain).



La solution pour optimiser ce traitement est donc de tirer parti des mécanismes de parallélisation des processeurs modernes. L'idée est d'exécuter le moteur de jeu dans un thread séparé, de manière à ce que le rendu du jeu puisse être fait au même moment que le moteur de jeu traite une commande. Lorsqu'une commande est réalisée par l'IA ou un joueur humain, on indique au thread Engine à la traiter. De ce fait le jeu est beaucoup plus optimisé : le moteur de jeu n'est exécuté que lorsque qu'une commande doit être traitée.



6.1.2 Conception logiciel

NB : Dans les précédentes parties du rapport, il est mentionné que le moteur de jeu (classe Engine est inclus dans le namespace client. Il s'est avéré que c'était une erreur de notre part, puisqu'il est nécessaire pour cette partie de conceptualiser l'instance client du jeu. C'est pour cela que le moteur de jeu a été déplacé dans son propre namespace engine.

L'implémentation de l'exécution parallèle du jeu nécessite que celle-ci soit effectuée dans son propre namespace, afin de pouvoir conceptualiser le client du jeu (qui communiquera par la suite avec le serveur pour les parties réseau). Le diagramme de classes du client est donc le suivant :

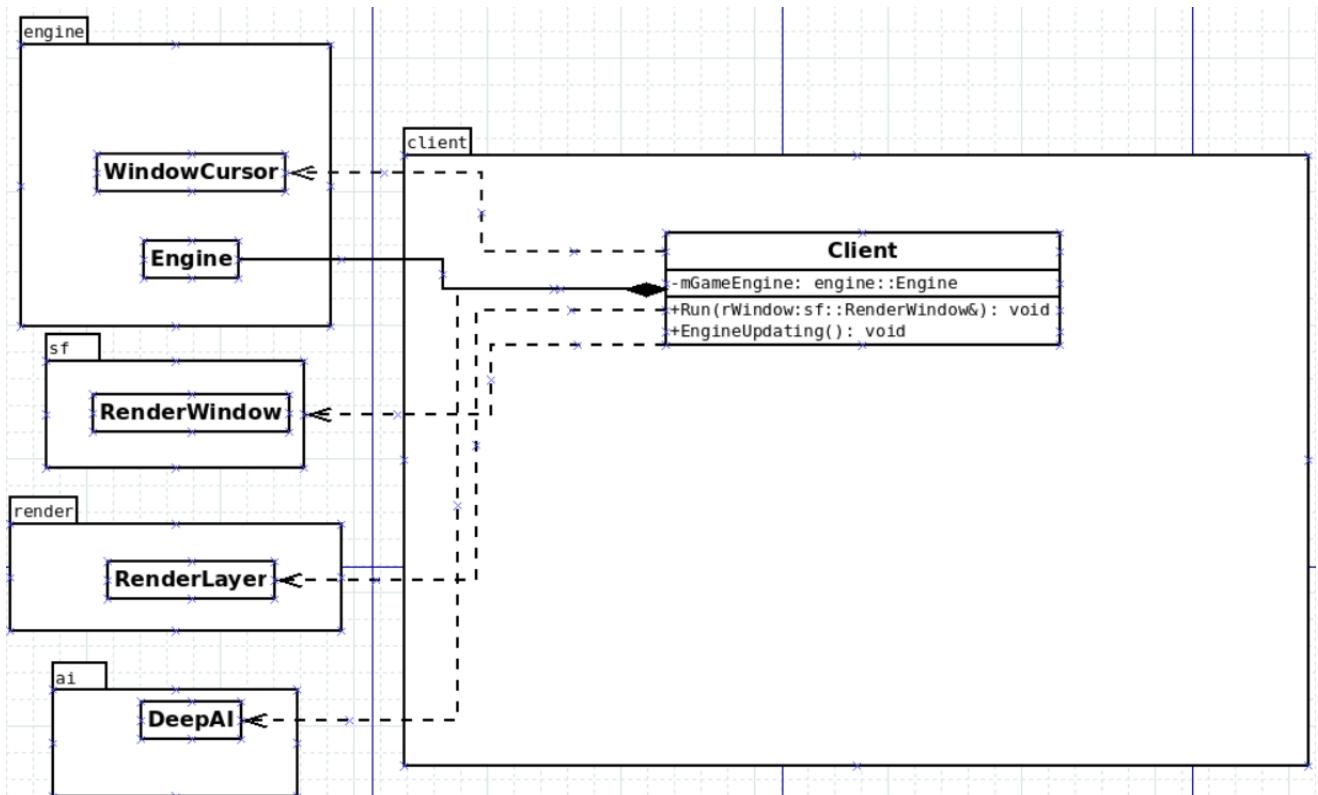


Diagramme de classes du client avec exécution parallèle

Le client est donc représenté par la classe *Client*. Il est composé de l'objet *mGameEngine* qui représente l'instance du moteur de jeu principal qui exécute le jeu. De même, il est composé des méthodes *Run* qui représente le fil d'exécution du thread principal du jeu, et *EngineUpdating* qui permet de débloquer le thread du moteur de jeu lorsqu'une commande est à traiter.

De ce fait, le thread du moteur de jeu n'est exécuté que dans les conditions suivantes régies par le thread principal :

- Dans l'état de jeu *INITIALISATION*, lorsque le rendu a fini d'être initialisé, pour indiquer au moteur de jeu que l'on peut changer à l'état *IN COMBAT*.
- Dans l'état de jeu *IN COMBAT*, lorsqu'une IA ou un personnage humain a saisi une commande.
- Dans l'état de jeu *OUT COMBAT*, pour effectuer les traitements hors combats
- Dans l'état de jeu *RENDER PROCESSING*, lorsque le rendu de transition est terminé, pour indiquer au moteur de jeu que l'on peut commencer un nouveau combat.
- Enfin, dans l'état de jeu *GAME OVER* pour afficher le record du joueur.