

Projet Logiciel Transversal

Projet worms

Grégoire de Faup – Antoine Delavoyppierre



Figure 1 - Worms Armageddon

© Steam

Update: Fonctionnement tour par tour p7 ; Changements autonomes p12 ; Conception logiciel p13

Sommaire

1	Présentation générale	3
1.1	Archétype	3
1.2	Règles du jeu	3
1.3	Ressources	4
2	Diagramme et conception des états	6
2.1	Description des états :	6
2.2	Conception logiciel :	7
3	Rendu : Stratégie et Conception.....	9
3.1	Stratégie de rendu d'un état.....	9
3.2	Conception logiciel :	9
4	Règles de changement d'états et moteur de jeu	12
4.1	Changements extérieurs	12
4.2	Changements autonomes :	12
4.3	Conception logiciel :	13
5	Intelligence Artificielle	15
5.1	Stratégies	15
5.1.1	Intelligence Artificielle Aléatoire	15
5.1.2	Intelligence Artificielle heuristique	15
5.1.3	Intelligence avancée :	16
5.2	Conception logiciel :	16
6	Modularisation.....	18
6.1	Organisation des modules	18
6.1.1	Répartition sur différents threads	18
6.1.2	Répartition sur différentes machines : rassemblement des joueurs.....	20

1 Présentation générale

1.1 Archétype

L'objectif que nous nous sommes fixé est de créer un jeu de type worms 2D mais bien sûr avec des fonctionnalités qui nous seront propres. C'est-à-dire :

- Génération d'une nouvelle carte à chaque partie
- Possibilité de détruire les éléments se trouvant sur notre carte
- Possibilité de jouer avec des amis ou contre une IA
- Création d'une équipe comprenant 1 à 5 personnages (peut varier selon le nombre de joueurs)
- Les positions de départ sont choisis par les joueurs sans qu'ils puissent savoir où se placent leurs adversaires
- Chaque personnage possède ses propres statistiques, ses attaques et a en plus des atouts.

Si cela est possible nous aimerions sauvegarder des données à chaque partie, ce qui permettrait de faire progresser les personnages (meilleures statistiques/atouts, attaques plus fortes etc).

Un personnage aura les statistiques suivantes :

- Vie
- Points de déplacements
- Nombre d'attaques/tour

Les atouts permettent aux personnages de booster ses statistiques: plus de vie ou résistance aux coups, plus de déplacements et plus d'attaques/tour.

1.2 Règles du jeu

Le joueur commence sa partie après avoir composé une équipe. Une carte de jeu est alors créée. Une fois la nouvelle carte affichée le joueur pourra placer un à un tous ses personnages en partant du haut de la carte et ce en un temps donné. Un joueur aura par exemple 15s pour déplacer ses personnages depuis le haut et les placer là où il le souhaite. Passé ce délai les personnages tomberont en chute libre.

Le jeu peut alors vraiment commencer. Puisqu'il s'agit d'un jeu tour par tour les joueurs jouent l'un après l'autre et non simultanément. L'ordre de jeu est choisi aléatoirement au début de la partie.

- Pendant son tour le joueur peut passer en revue ses personnages et choisir celui qu'il souhaite.
 - Son personnage peut alors se déplacer, utiliser son atout et attaquer
- Une fois ses points de mouvements et d'attaques utilisés le tour de ce joueur est automatiquement terminé (toutefois un joueur est libre de terminer son tour avant).
- Une fois qu'un atout a été utilisé il faut attendre qu'il se recharge après un temps aléatoire pour le réutiliser.
- Lorsqu'un personnage a perdu toute sa vie à cause des attaques des autres joueurs (ou de chutes) il est retiré de la partie et n'apparaît donc plus à l'écran.
- Le dernier joueur en jeu remporte la partie.

1.3 Ressources

Comme pour tout jeu vidéo la partie affichage nécessite d'utiliser des ressources graphiques. Dans notre cas nous allons avoir besoin de ressources pour l'affichage de notre carte de jeu, mais aussi pour représenter les personnages et leurs animations, et enfin pour afficher des informations concernant l'avancement du jeu, les statistiques des personnages etc.

Pour créer une carte de jeu nous allons appliquer un masque sur des textures représentant un ciel, un sol, de l'herbe (et d'autres textures selon notre avancement dans la partie graphisme). Le masque sera généré aléatoirement à chaque partie permettant ainsi d'avoir sans cesse de nouveaux terrains de jeu.



Figure 2 - Texture ciel terrain et frontière

Concernant les personnages nous avons choisi une approche classique consistant à utiliser des collections de sprites ou tileset. Il s'agit en fait d'une planche sur laquelle on retrouve notre personnage dans toutes les positions qu'il peut prendre en jeu. Voici quelques un des personnages choisis avec leurs mouvements et/ou des attaques. Cette section est amenée à évoluer.

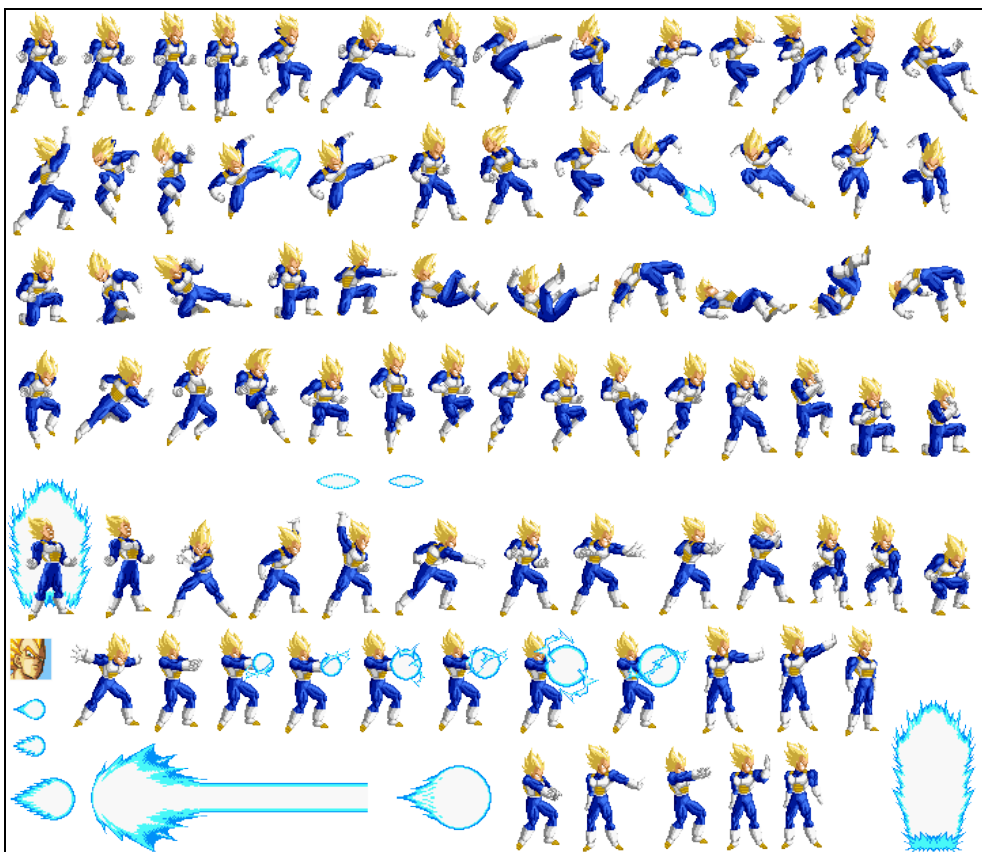


Figure 3 - Sprite Vegeta

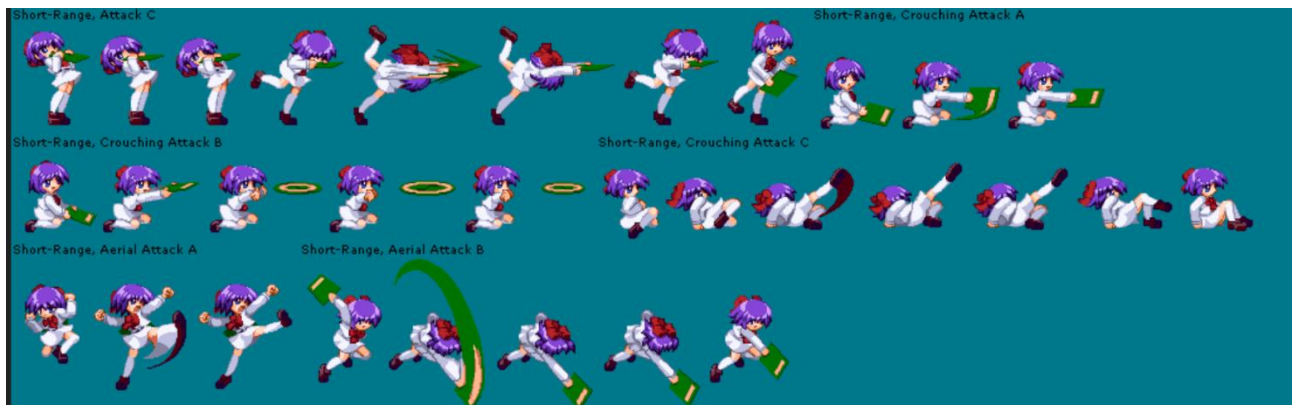


Figure 4- Sprite Mio Kouzuki

Enfin pour la dernière partie de l’affichage (les informations sur l’état de jeu) nous utiliserons une ou plusieurs polices pour l’affichage de texte sur l’écran ainsi que des sprites d’inventaires, de menu etc.

2 Diagramme et conception des états

2.1 Description des états :

Nous allons ici décrire tous les éléments pouvant être présent dans un état de jeu de la manière la plus exhaustive possible (différentes combinaisons possibles, paramètres etc).

Un état de jeu comprend donc toujours les éléments suivants :

- un ID
- une carte de jeu
- des joueurs
- des personnages

- L'ID :

L'identifiant d'un état de jeu nous permet de connaître dans quelle phase le jeu se trouve actuellement. En effet nous avons défini 6 phases qui seront utiles par la suite car elles nous permettront par exemple d'avoir des interprétations différentes de même commandes voir même d'en bloquer.

Les phases sont « *not_started* » qui servira pour l'affichage des menus avant le début de la partie, « *team_selected* » qui récupère la liste des personnages du jeu, construit une carte de jeu et place les personnages en haut de celle-ci. Vient ensuite « *team_placement* » qui correspond à la phase durant laquelle les joueurs viennent positionner leurs personnages sur la carte de jeu. Enfin on retrouve les phases « *started* » « *paused* » et « *end* ».

- La carte de jeu :

Comme expliqué dans la partie **1.3 Ressources** la construction de notre carte de jeu se base sur l'utilisation d'un masque généré aléatoirement. A l'heure actuelle chaque pixel du masque correspond à un pixel de la carte de jeu, ceci pourra évoluer par la suite (on pourrait dire que un pixel du masque correspond à un carré de pixels dans le jeu). La taille de la carte de jeu en nombre de pixels est pour l'instant de 3000x2000 pixels, cette taille pourra augmenter au fur et à mesure que sa création sera de plus en plus réaliste. Les 60 premiers % de la hauteur de la carte sont réservés à la texture de fond (ciel), les 40% restants peuvent être composés de fond, d'herbe et de sol.

- Les joueurs :

Les joueurs comme son nom l'indique représentent les personnes prenant part au jeu, cela concerne aussi bien une personne réelle qu'une intelligence artificielle. Chaque joueur possède un nom ainsi qu'un nombre de personnages qui lui sont propres.

- Les personnages :

Les personnages en jeu sont la propriété d'un seul joueur, ils possèdent un identifiant qui correspond au personnage que l'on a sélectionné (goku, vegeta etc), des statistiques, une position dans le jeu ainsi que des critères d'attaques (propriété qui sera sûrement déplacé ailleurs). Les atouts ne sont pas encore présent dans le jeu, ils seront soit inclus dans les personnages soit dans le moteur de jeu.

2.2 Conception logiciel :

Le diagramme des classes pour les états est présenté en Figure 5.

- **Hiérarchie :**

La classe **GameState** est au sommet de la hiérarchie, c'est elle que l'on appelle pour créer un état de jeu. Elle possède une liste de tous les joueurs en jeu ainsi que des personnages, la carte de jeu et son ID de phase de jeu.

Cette classe agit donc comme un conteneur d'éléments puisque chaque élément est accessible soit directement soit par le biais d'un des attributs.

Les classes Player, Character, Statistics, Position et Map en sont des agrégations.

Nous avons décidé de stocker les joueurs et les personnages dans des vecteurs de shared pointeur pour que ces derniers soient accessibles et modifiables depuis d'autres classes et cela en évitant toute fuite mémoire.

- **Fonctionnement tour par tour :**

Notre jeu est un tour par tour. Pour indiquer au moteur de jeu quel joueur a le droit de jouer pendant un tour nous avons créé la variable `current_player` dans la classe GameState ; de même dans la classe Player on retrouve la variable `current_character` pour savoir quel personnage le joueur a choisi d'utiliser. Lorsque le moteur de jeu exécutera des commandes il pourra savoir qui les lui a envoyés grâce à ces deux variables.

- **Observateurs de changements :**

GameState, Player, Map, Position et Statistics des observables. Un objet de ces classes peut donc notifier un changement de ses attributs à un observateur : il s'agit du pattern « observer ». L'ajout de ce pattern est dû au rendu graphique. En effet lorsque l'on nous allons devoir afficher un état il faudra également que cet affichage s'actualise lorsque les données de l'état seront modifiées par le moteur de jeu. Pour cela il faut faire communiquer les objets des classes d'état avec les objets responsables du rendu ou de l'actualisation du rendu.

Le procédé de communication entre une observable et un observateur est le suivant. L'observable notifie tous ses observateurs (via `notifyObservers`) qu'un événement (objet de type `Events`) s'est produit (l'événement à un ID unique). Les observateurs définissent la méthode abstraite `stateChanged` qui analyse l'événement passé en argument et agissent en fonction.

Toutes les observables possèdent par héritage une liste de leur observateurs, cette dernière est statique pour s'assurer que les observables partagent tous les mêmes observateurs.

Lorsque la vie d'un personnage tombe à 0 ou que sa position a changé on notifie l'observateur, de même lorsque le dernier personnage d'un joueur est mort. Pour identifier précisément quel personnage ou joueur a changé on transmet dans l'événement un pointeur vers cet objet. Enfin on notifie les observateurs lorsque la carte est modifiée par les joueurs.

Remarque : Chaque classe observable possède son propre sous type Event (classe fille de Event). Il est

possible de n'utiliser que Event et nous utiliserons peut être cette stratégie plus tard. GameState n'a pour l'instant pas besoin d'être observable et cette propriété pourrait être supprimé prochainement.

Notre état de jeu est défini comme le diagramme UML ci-dessous le représente :

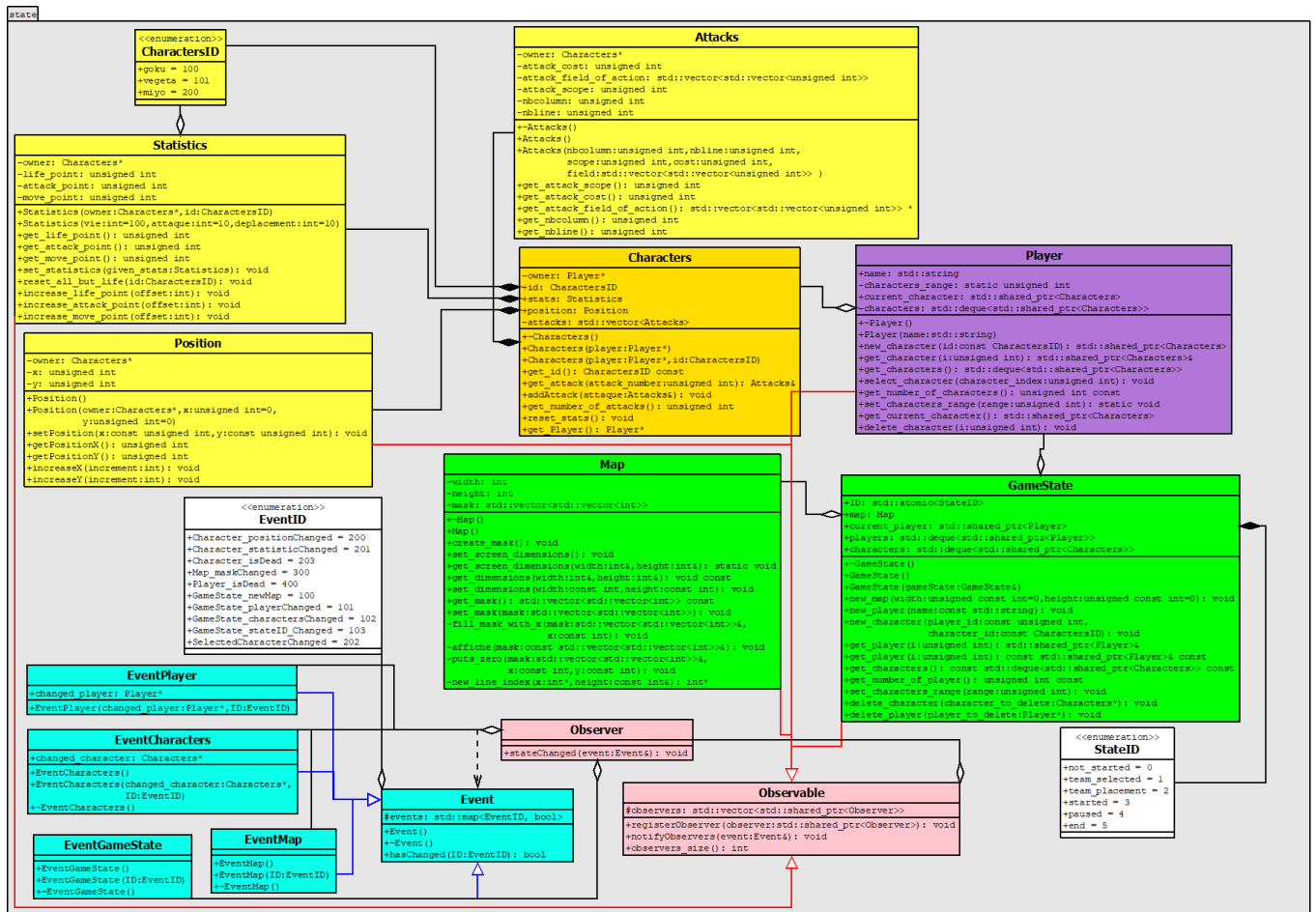


Figure 5 - Diagramme des classes d'états

3 Rendu : Stratégie et Conception

3.1 Stratégie de rendu d'un état

La stratégie adoptée pour réaliser le rendu d'un état de jeu est celle de la **décomposition en layers**. Tous les éléments graphiques ont une nature différente et leur affichage peut être traité de manière séparée étant donné qu'ils ne sont pas modifiés à la même fréquence. Avec une approche layer nous n'avons plus à recalculer à chaque rafraîchissement l'ensemble des ressources graphiques mais seulement celles ayant changé entre deux rafraîchissements.

Il existe 3 types de layer pour coïncider avec les 3 types de ressources décrits dans la partie 1.3, c'est-à-dire un layer pour les personnages, un autre pour la carte de jeu et enfin un layer pour les informations sur l'état de jeu affiché à l'écran.

Les ressources des layers personnages et informations sont regroupés dans des **Tileset**. Pour afficher la **Tile (ou tuile)** que l'on souhaite au sein d'un Tileset nous utilisons une matrice à 4 valeurs (position haut gauche de la tuile et ses dimensions) ainsi qu'une image source qui sera sauvegardé dans une variable image. On associe une texture à cette image et enfin on liera une **sprite** à cette texture.

On remarquera sur le diagramme UML du render (affichage) - qui se trouve plus bas – la présence de deux classes en verts. Ces deux classes « controller » et sfEvents permettent de surveiller les événements de type SFML générés dans le thread d'exécution de la fenêtre SFML. Nous reviendrons en détail sur ces deux classes dans la partie 4. de ce rapport.

3.2 Conception logiciel :

- **Layers :**

Nous allons commencer par le plus important : les layers ! Pour gérer les 3 types nous utilisons un système d'héritage. Une classe parente '**Layer**' fournit les méthodes et attributs nécessaires pour gérer tout type de layer et les filles redéfinissent si besoin les méthodes selon leurs spécificités.

Chaque layer possède une méthode 'update' pour se mettre à jour lorsqu'un élément du render a changé ainsi qu'une méthode 'setSurface' pour afficher le contenu du dans la fenêtre SFML. De même ils possèdent tous deux attributs qui sont des vecteurs de pointeurs uniques vers une **Surface** et un **Tileset**.

Les classes filles de **Layer** : **Character** et **Background**, définissent respectivement les éléments de terrain et les différents personnages décrits dans l'état de jeu. À tout moment notre rendu est donc composé de deux instances de **Layer**. Chaque personnage en jeu possède son instance de surface et son Tileset.

- **Surface :**

La classe **Surface** possède les méthodes et attributs permettant d'afficher nos ressources dans une fenêtre SFML. C'est-à-dire une texture ainsi qu'un sprite pour les attributs, et des méthodes pour charger une texture depuis une image SFML, associer un sprite à une texture, positionner la texture à une position donnée mais aussi de définir la texture comme n'étant qu'une partie de l'image d'origine et pour finir

afficher le sprite dans la fenêtre SFML. L'image d'origine est le plus souvent un tileset et la texture une tile du tileset.

- **Tileset et Tile :**

La classe **TileSet** permet de charger des ressources tels que les tileset en format png (ou autres) dans un élément `sf::Image` ce qui servira ensuite à créer des textures.

La classe **Tile** sert simplement à définir un rectangle à appliquer à un tileset pour afficher le sprite que l'on souhaite.

- **Scene :**

La classe **Scene** permet de piloter toutes les classes que nous venons de lister et c'est elle qui va donc superviser le rendu d'un état ! En effet une fois que l'on a créé une instance de Scene son constructeur créé ensuite des instances pour chacun des layer définit ; Scene a un attribut de chaque type de layer : *background* pour le layer de la carte et *characters* pour le layer des personnages.

Pour afficher les layers on appelle depuis la fonction main la méthode draw de notre objet de classe Scene. Cette méthode appelle ensuite pour tous ses attributs de type Layer leur méthode setSurface qui appelle la méthode draw de Surface. Ce chemin d'appel se contente seulement d'afficher les sprites contenus dans toutes les instances de Surface et ceci est par ailleurs obligatoire puisque l'on rafraichit en permanence le contenu de la fenêtre de jeu.

Lorsque l'état de jeu est modifié nous avons vu dans la partie 2.2 notre diagramme d'état implémente le pattern observer, ainsi lorsque l'état de jeu est modifié l'observable ayant changé appellent ses observateurs et les notifie qu'un changement de type Event avec un ID de type EventID s'est produit. Pour l'instant le seul observateur de l'état de jeu est la classe Scene. Nous avons donc défini dans Scene la méthode virtuelle pure stateChanged hérité de la class `state::Observer`. Celle-ci analyse entre autre l'ID de l'événement pour déterminer quel layer a changé et ce qui a changé dans le layer. La méthode met alors à jour la surface concernée. Le changement sera visible au prochain appel de la fonction update depuis la fonction main.

Lorsqu'un personnage meurt un événement est créé dans la classe Statistics ce qui a pour effet d'appeler la classe Scene et de supprimer tous les `shared_ptr` qui faisaient référence au personnage venant de mourir (de même lorsqu'un joueur meurt). On a donc ajouté un attribut `gameState` qui est une référence vers l'objet GameState créé au début de la partie.

Une fois que les différents appels de fonctions se sont terminés et que l'affichage est actualisé le personnage disparaît du jeu.

La conception logiciel du rendu d'un état est décrite dans le diagramme UML ci-dessous :

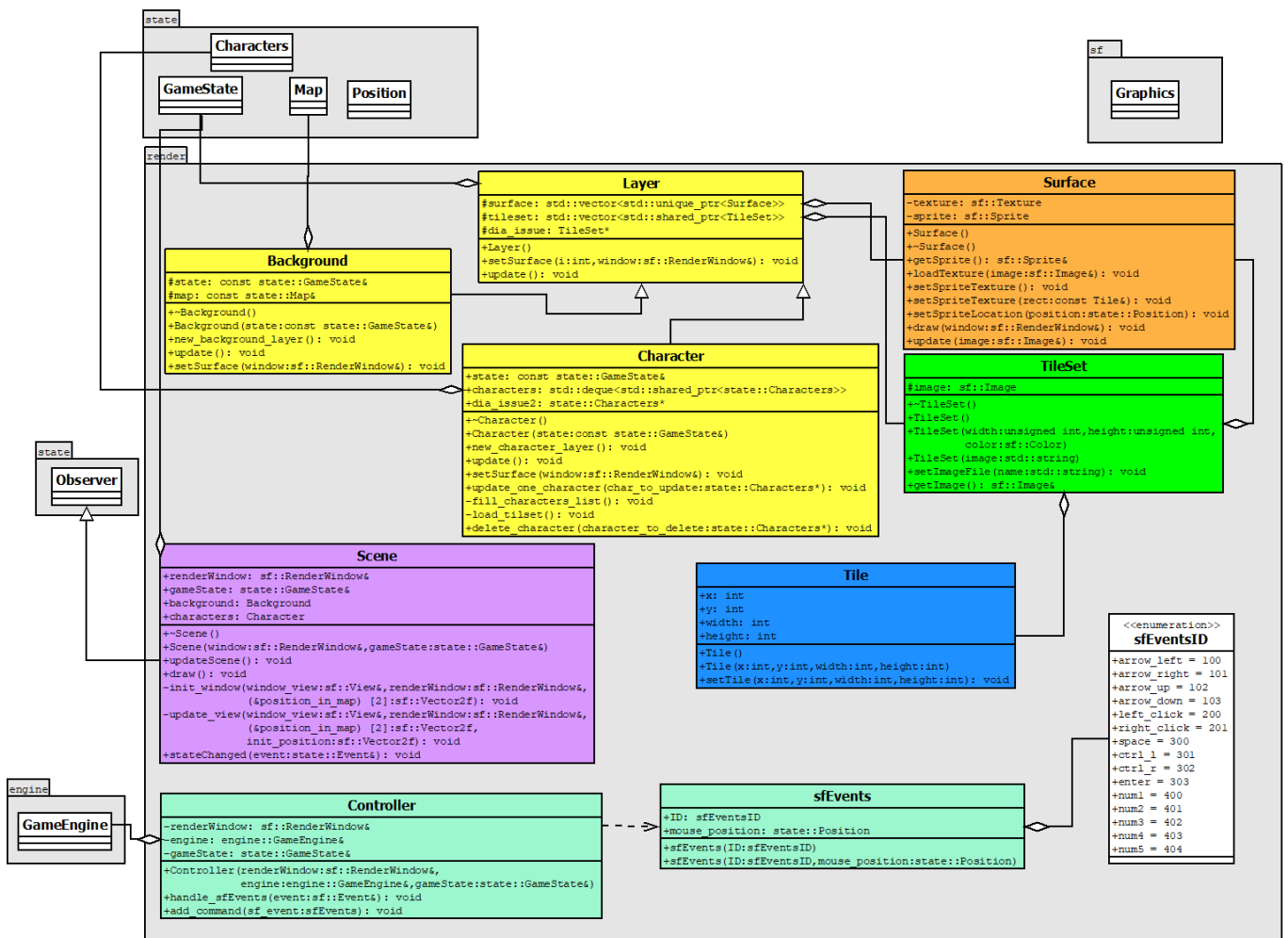


Figure 6 - Diagramme des classes de rendu

4 Règles de changement d'états et moteur de jeu

4.1 Changements extérieurs

Les changements d'états extérieurs sont ceux provoqués par les joueurs en saisissant des commandes de jeu dans la fenêtre SFML. Ces commandes vont être interprétées par le moteur de jeu qui modifiera l'état de jeu en conséquence. Les commandes sont les suivantes :

Déplacement : Un joueur peut déplacer le personnage actuellement sélectionné en utilisant les flèches directionnelles du clavier (gauche, droite). Cela a pour effet de modifier sa position sur la carte du jeu.

Choisir un personnage : A l'aide des touches haut et bas un joueur peut choisir quel est son `current_character`. Seul ce personnage peut effectuer des actions (déplacements, attaques etc).

Changement de joueur : Cela revient en fait à finir son tour. Cette commande change le joueur sélectionné et passe la main au joueur suivant. Il suffit pour cela d'appuyer sur '**Enter**'.

- ~~Seulement lorsque la phase de jeu est « team placement »~~

~~Validation du placement des équipes : Appui sur la touche espace. En mode solo vs IA cela est géré par l'IA.~~

- Seulement lorsque la phase de jeu est « started »

Lancement d'une attaque : Le joueur peut lancer une attaque en pointant l'endroit où il souhaite lancer son attaque avec la souris et en appuyant sur les touches 1, 2, 3, 4 ou 5 pour choisir l'une des 5 attaques dont dispose chaque personnage. Cette commande change les statistiques des différents personnages impactés par cette attaque et également l'aspect de la carte de jeu.

4.2 Changements autonomes :

- Lorsque le jeu n'a pas commencé on passe successivement d'un ID d'état à un autre ; on commence tout d'abord à « not started » puis on passe en « team selected » suivi par « team placement » où les joueurs peuvent se positionner en début de partie.

La transition de `not_started` à `team selected` se fait lorsque l'on quitte la page d'accueil en appuyant sur Entrée ; `team selected` passe directement à `team placement` (cette phase permettrait aux joueurs de choisir les personnages qu'ils veulent utiliser) ; `team placement` passe à `started` une fois que chaque joueur a placé ses personnages à sa convenance sur la carte. Enfin `started` passe à `end` quand la partie est terminée.

- Si un personnage n'a plus de points de vie il est supprimé du jeu (sauf s'il s'agit du dernier personnage).
- Si un joueur n'a plus de personnages il est retiré du jeu (sauf s'il s'agit du dernier joueur).
- Les statistiques (points de déplacements et d'attaques) des personnages reviennent à leur valeur de base à chaque tour. Seule la vie demeure inchangée entre chaque tour.
- Les personnages suivent automatiquement le relief du sol et sont toujours attirés vers le sol par un **système de gravité**.

Système de gravité :

Le fonctionnement du système de gravité est assez simple. Le moteur de jeu dispose d'une liste contenant la vitesse verticale de chaque personnage. Une fonction **gestion_gravite** appelée régulièrement vérifie que les personnages touchent le sol. Si ce n'est pas le cas la vitesse verticale du personnage augmente et la fonction fait tomber le personnage d'une distance proportionnelle à la vitesse stockée. Ainsi on a la sensation que le personnage tombe de plus en plus vite. Si le joueur touche le sol, la vitesse verticale du personnage est remise à zéro.

Un appui sur la **touche espace** permet de modifier la vitesse verticale du joueur sélectionné, en la mettant à une valeur négative. Le système de gravité ayant son effet le personnage s'élève un moment avant de retomber ce qui permet de sauter.

4.3 Conception logiciel :

Les changements extérieurs sont détectés grâce à la classe « **Controller** » qui surveille les événements se produisant dans la fenêtre du jeu. Ils sont ensuite traduits en commandes à envoyer au moteur de jeu. Les commandes sont en fait des objets de la classe « **sfEvents** ». On envoie ensuite les commandes de l'utilisateur à la classe GameEngine du namespace engine en utilisant sa méthode `add_command`.

Nous avons donc une liste de commandes que le moteur de jeu doit traiter. Ces événements deviennent alors des commandes de type « **Command** ». Toutes les commandes : **Move**, **Attack**, **change player/character** héritent de la classe **Command** et dispose en cela d'une méthode **isLegit** pour déterminer si la commande est légale ou non ainsi que d'une méthode **execute** pour exécuter l'action si celle-ci est légale. **Command** est une classe abstraite.

Command est une classe abstraite symbolisant une commande, on y définit les méthodes virtuelles **isLegit** et **execute** qui seront propres à chacune des classes filles. Les classes filles sont les suivantes.

- **Attack** : permet d'infliger une attaque sur une zone qui affecte joueurs et terrain.
- **Move** : déplace un personnage dans la direction souhaitée
- **ChangeCharacter** pour passer au personnage suivant
- **ChangePlayer** pour passer au joueur suivant

Diagramme UML du moteur de jeu :

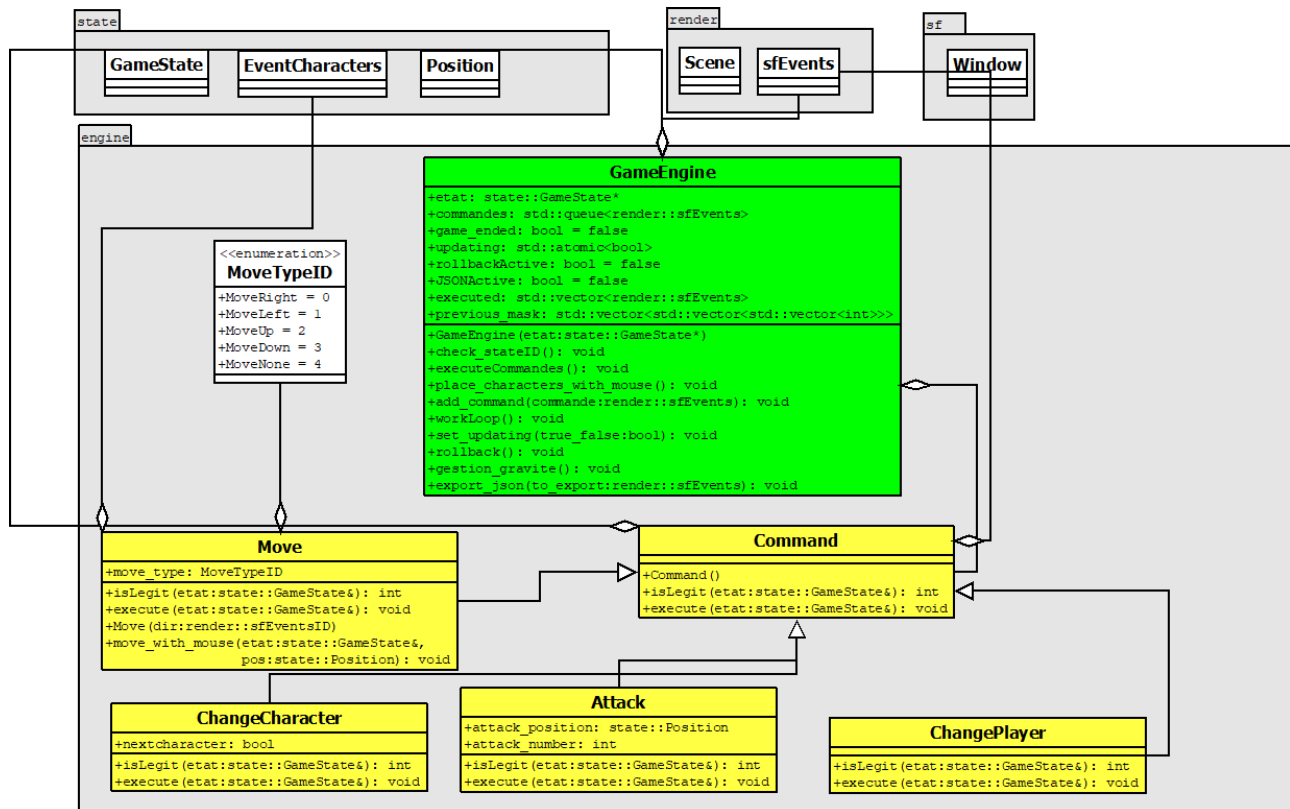


Figure 7 - Diagramme des classes de moteur de jeu

5 Intelligence Artificielle

5.1 Stratégies

5.1.1 Intelligence Artificielle Aléatoire

Dans un premier temps, nous avons conçu une intelligence artificielle jouant de manière totalement aléatoire à chaque tour. Plus précisément, l'IA décide à chaque tour de se déplacer aléatoirement vers la droite ou vers la gauche d'une distance elle aussi aléatoire ou alors d'attaquer (même dans le vide) en enfin de passer son tour.

La conception logiciel de cette IA est des plus simple, elle ne comporte qu'une seule classe : **RandomIA** qui dispose en attribut d'un moteur de jeu. Le moteur permet ensuite à l'unique méthode de cette classe, `play()`, de déterminer si c'est à l'IA de jouer, et si oui, ajoute des commandes à la liste du moteur de jeu.

Cette classe est instanciée depuis la fonction `main` du jeu et sa méthode appelée de manière régulière afin de permettre à l'IA de jouer.

5.1.2 Intelligence Artificielle heuristique

Notre deuxième intelligence artificielle s'appuie sur un ensemble d'heuristiques pour offrir de meilleures performances que le hasard et de possiblement remporter la partie.

1. Déterminer quel personnage choisir :

On cherche pour chacun des personnages de l'IA l'adversaire qui est le plus proche. Une fois trouvé il est alors ciblé par le personnage de l'IA qui en est le plus proche (l'attaquant).

2. Choix de l'attaque :

On recherche parmi les attaques de l'attaquant celle qui peuvent atteindre le personnage ciblé.

3. Attaque :

Si une attaque permet d'atteindre la cible alors on lance l'attaque sur la cible jusqu'à temps de ne plus avoir de points d'attaques. Fin du tour de l'IA.

4. Déplacement :

Si aucune attaque ne peut atteindre la cible on se déplace alors vers la cible.

5.1.3 Intelligence avancée :

Nous avons réalisé une intelligence plus avancée en suivant une méthode de résolution de problème à états finis : l'algorithme min max. Sachant que notre jeu est conçu pour être un jeu multi-joueurs et multi-personnages nous avons simplifié la description du problème pour ne pas parcourir des arbres immenses.

- Le système est un système à deux joueurs : le premier est l'IA, le deuxième tous les autres joueurs. Le premier étage est donc celui des attaques portées par l'IA, le suivant celui de l'ennemie et ainsi de suite.
- Un nœud de l'arbre correspond à une attaque effectuée par un joueur avec l'un de ses personnages. Ainsi si on joue à 2 joueurs, 3 personnages vs 3, l'IA commencera l'arbre avec 3 fils et chacun d'entre eux aura 3 fils correspondant aux 3 joueurs de l'adversaire.
- Si un nœud de l'arbre n'a pas de fils (c'est-à-dire que le joueur de l'étage suivant a perdu et donc que le joueur de l'étage supérieur a gagné) on appelle la fonction d'évaluation.
- Si lorsqu'un personnage attaque il ne peut pas ou plus trouver de cibles (il a donc gagné le jeu) on arrête la recherche dans cette branche de l'arbre et on appelle la fonction d'évaluation.

La fonction d'évaluation permet de donner à chaque nœud de l'arbre un poids, le poids qui est remonté à l'étage supérieur (nœud parent) est alternativement la valeur max ou min parmi celle de tous les fils du parent. L'IA cherche à maximiser cette fonction d'évaluation et elle récupère donc la valeur max des poids que possèdent les fils d'un de ces nœuds. Et à contrario l'ennemie cherche à la minimiser.

La fonction utilisée à proprement parlé n'est pas encore arrêtée, il s'agit pour l'instant de la différence entre la vie de l'IA et celle de son ennemie.

5.2 Conception logiciel :

La conception logicielle est résumée dans le diagramme UML présent plus loin.

Classe AI : Classe abstraite qui contient les attributs et méthodes communes à nos 3 stratégies d'IA. On a donc :

- une référence vers un moteur de jeu
- un attribut name de type string
- un constructeur
- un destructeur virtuel
- une méthode virtuelle **workloop** qui est la méthode avec laquelle est lancé le thread de l'IA
- une méthode abstraite **play** implémentée par chaque classe fille
- une méthode place_character et next_player pour respectivement placer les personnages de l'IA au début du jeu et changer de joueur.

Classe DeepAI : Il existe deux manières de parcourir l'arbre des différents états. La première est le « clonage d'état » (on réalise une copie en profondeur de GameState et lors du parcours de l'arbre des états on copie et restaure des états. La deuxième consiste à modifier notre moteur de jeu pour le rendre capable d'annuler des actions et donc de revenir en arrière – le rollback-. Les deux méthodes ont été testées et sont fonctionnelles. Le rollback est beaucoup gourmand en mémoire dans notre cas que le système de copie d'état, cependant il est légèrement plus rapide.

Concernant le livrable 3.final, la commande ./bin/client rollback ne fait que lancer une partie à 2 IA, toutes deux heuristiques. Le rollback fonctionne bien lorsqu'il est utilisé par la DeepAI mais n'est pas adapté pour un fonctionnement en jeu normal car dans une partie lorsqu'un joueur ou personnage meurt il est supprimé du jeu et ne peut pas être restauré donc le rollback ne fonctionnerait pas correctement dans cette situation. On pourrait modifier le rollback pour qu'il recrée les personnages mort mais cette fonctionnalité n'est pas prévu.

Création de plusieurs IA :

Le champ **name** est un identifiant unique à chaque IA (5 IA différentes possibles). Une IA à le droit de jouer si le nom du joueur courant correspond à son celui stocké dans sa variable name. Ceci nous permet de faire jouer plusieurs IAs ensemble.

Diagramme UML des stratégies IA:

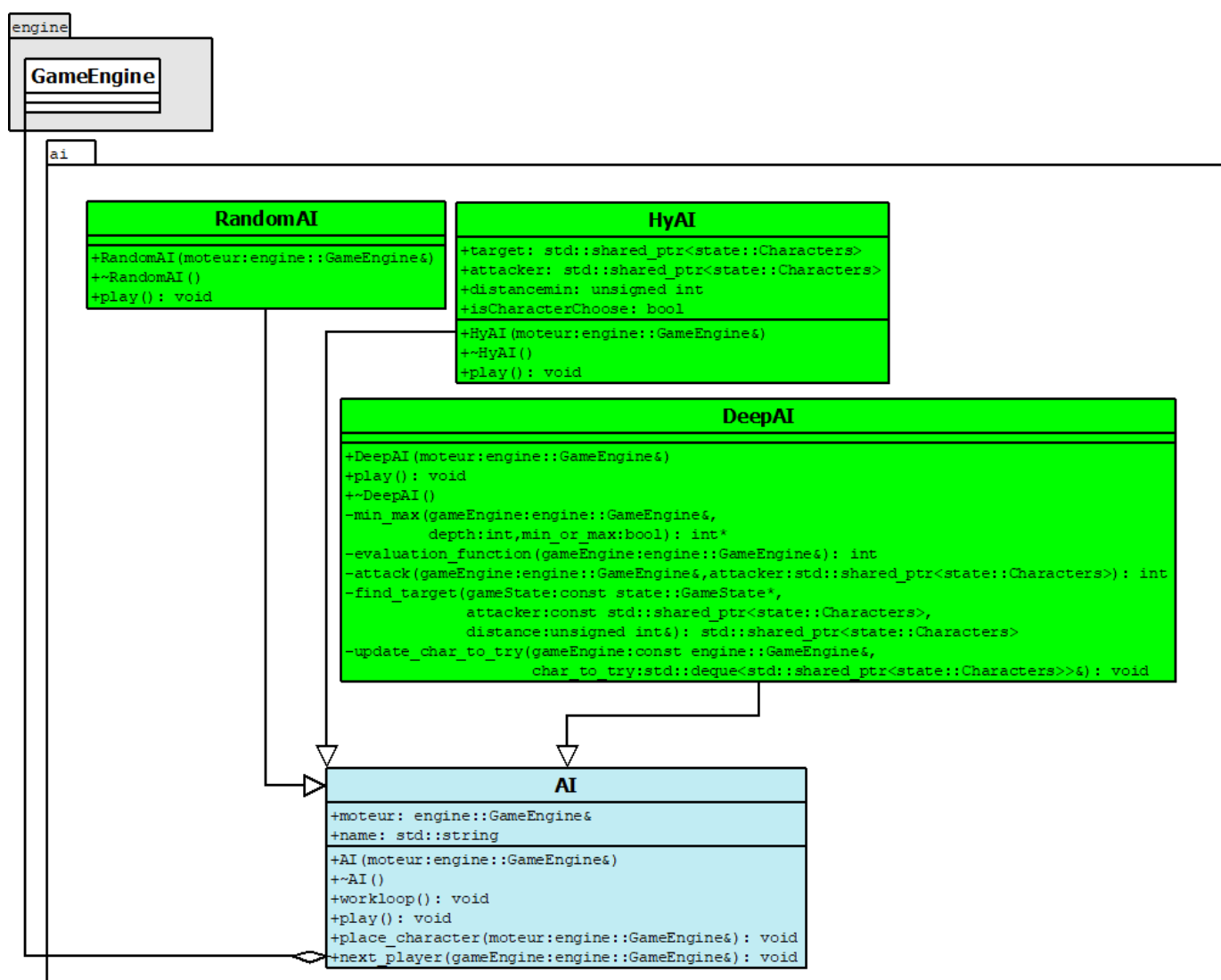


Figure 8 - Diagramme des classes d'IA

6 Modularisation

6.1 Organisation des modules

6.1.1 Répartition sur différents threads

Nous avons organisé notre jeu en différent thread qui fonctionne de manière autonome jusqu'à la fin du jeu – fin du jeu qui peut être du soit à la fermeture de la fenêtre de jeu soit au fait qu'un joueur a gagné.

On retrouve donc un thread principal pour le rendu et la coordination du jeu, un thread pour le moteur de jeu et un thread pour l'IA.

Les commandes : Lors d'une partie les commandes sont réceptionnées par le thread principal en utilisant la fonction **handle_sfEvents** qui les ajoute dans une variable **commandes** du GameEngine - il s'agit d'une FIFO où chaque élément est lu puis exécuté par le moteur de jeu. Les commandes sont acquises en temps réel mais exécuté toutes les 80ms (temps d'endormissement du thread engine).

Les notifications de rendu circulent entre le state et le render mais puisque ces deux modules sont dans le même thread il n'y a pas de problème.

Thread principal :

```
// Menu // Déclaration et initialisation des variables nécessaires au jeu
state::GameState etat;
render::Controller controller ; //Objet gérant les inputs des joueurs dans la fenêtre de jeu SFML
shared_ptr<render::Scene> scene; // Objet supervisant l'affichage de tous les elements affichable du state

// Lancement d'un thread pour le moteur de jeu
std::thread thread_engine = thread(&engine::GameEngine::workLoop, &engine);

// Lancement d'un thread pour l'IA
std::thread thread_ai = thread(&ai::DeepAI::workloop, ai_);

while (renderWindow.isOpen())
{
    // Gestion des événements (clics souris etc)
    sf::Event event;
    while (renderWindow.pollEvent(event))
        controller.handle_sfEvents(event);

    // Attendre que l'engine est fini de s'update (s'il y a update)
    while (engine.updating) {}

    // update de l'affichage
    scene->draw();
}
```

Sachant que l'exécution d'une commande modifie l'état de jeu il ne faut pas afficher l'état de jeu tant que le moteur de jeu est en train d'exécuter une commande. Cela pourrait conduire à des erreurs de segmentation (par exemple SFML tente d'afficher un personnage que l'engine a supprimé entre temps).

Thread moteur de jeu :

Le moteur de jeu tourne jusqu'à la fin du jeu mais pour éviter de consommer du temps de CPU inutilement on fait dormir le thread après avoir vidé la liste des commandes et les avoir exécutés (au travers de **check_state_ID**). Lorsque le moteur de jeu s'exécute il prend le mutex **get_engine** définit dans le fichier **global_mutex** puis le restitue une fois l'exécution faite.

```
void GameEngine::workLoop()
{
    // Tant que l'on n'a pas fini le jeu ou fermé la fenêtre de jeu
    while (!((etat->ID == state::StateID::end) || game_ended))
    {
        global::get_engine.lock();
        check_stateID();
        global::get_engine.unlock();

        std::this_thread::sleep_for(std::chrono::milliseconds(80));
    }
}
```

Thread IA :

De même que pour le moteur de jeu, les IA ont leur propre thread qui tourne en permanence jusqu'à la fin du jeu. Puisque l'IA a besoin du moteur de jeu (défini dans le main) une fois le calcul du min max terminé pour lancer son attaque on choisit de locker le mutex **get_engine** ce qui a pour effet de bloquer le thread du moteur de jeu et on peut donc exécuter les commandes de l'IA directement depuis son thread. Lorsque l'IA n'a pas joué son thread s'endort pour 500ms.

```
void AI::workloop()
{
    // Tant que l'on n'a pas fini le jeu ou fermé la fenêtre de jeu
    while (!((moteur.etat->ID == state::StateID::end) || moteur.game_ended))
    {
        if (moteur.etat->current_player->name == "IA")
        {
            std::unique_lock<std::mutex> unique_get_engine(global::get_engine);
            play();
        }

        else    std::this_thread::sleep_for(std::chrono::milliseconds(500));
    }
}
```

6.1.2 Répartition sur différentes machines : rassemblement des joueurs

Sérialisation :

Dans un premier temps, nous avons mis en place un système permettant d'encoder les commandes envoyés au moteur de jeu au format JSON.

Ce système est basé sur une unique fonction, `export_json`, qui prend en paramètre un `sf::Events`, c'est à dire une commande et crée une entrée dans un fichier JSON déjà existant (ou qui le crée dans le cas de la toute première commande) contenant les informations sur la commande (touche pressée et position de la souris).

Le but est d'utiliser cette fonction côté client pour encoder une liste de commande à envoyer au serveur afin de toutes les envoyer en même temps dans un format pris en charge par la majorité des protocoles et des langages de programmations.

Une fonction `import_json` se chargera côté serveur de lire le fichier envoyé et d'ajouter les commandes correspondantes à la liste des commandes du moteur de jeu.

Conception du serveur

D'un point de vue logiciel, le serveur est conçu comme suit :

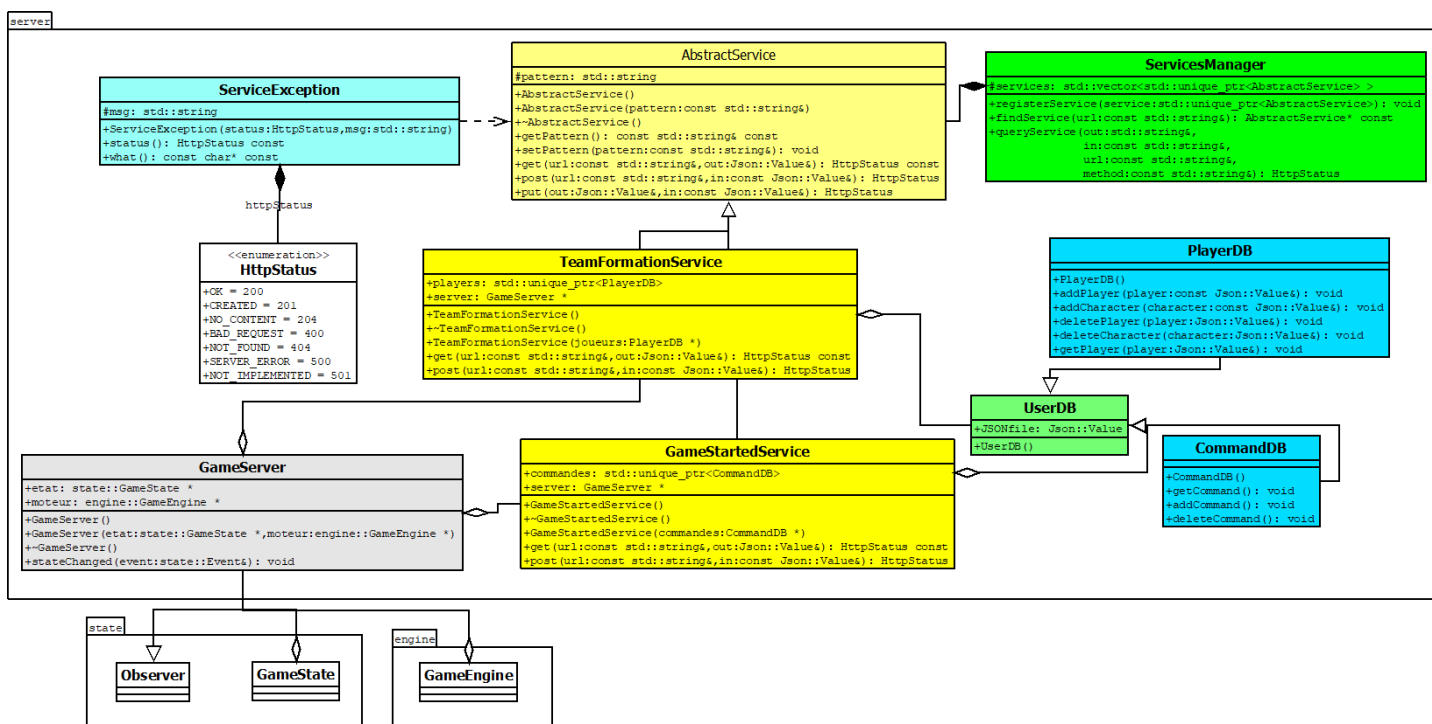


Figure 9 - Diagramme des classes pour la modularisation

Dans la conception, l'API est séparée en deux types de services :

- les services permettant la formation des équipes avant le début du jeu - implémentés dans la classe **TeamFormationService** (associé à une base de données **PlayerDB**)
- les services permettant l'envoi et la réception des commandes à exécuter - implémentés dans la classe **GameStartedService** (associé à une base de données **CommandDB**).

Cette séparation a été réalisée car les deux services ne manipulent pas les mêmes données.

Formation des équipes :

Le service de formation des équipes fonctionne de la manière suivante :

- un client se connecte au serveur
- il tente d'accéder au serveur en ajoutant son nom de joueur à la liste des joueurs
 - soit il peut rejoindre la partie et il est dans ce cas ajouté à la BDD « PlayerDB »
 - soit il ne peut pas et selon la raison il peut essayer de se reconnecter
- une fois enregistré il peut ajouter des personnages dans son équipe

La base de données manipulée est un JSON décrivant chaque joueur ainsi que les personnages qui compose son équipe. Concrètement le fichier ressemble à ce qui suit :

```
{
  "players": ["Joueur 1", "Joueur 2"],
  "team": [
    {
      "characters": [ 100, 100, 100 ],
      "name": "Joueur 1"
    },
    {
      "characters": [ 200, 200, 200 ],
      "name": "Joueur 2"
    }
  ]
}
```

Pour le moment, ce service de l'API peut recevoir 4 requêtes différentes :

- Ajout d'un joueur à la partie :

methode : POST

URL : */TeamFormationService/player*

donnée : description du joueur au format JSON :

```
{
  "name": "Joueur 2"
}
```

- Ajout d'un personnage à un joueur existant :

methode : POST

URL: */TeamFormationService/character*

donnée : description du personnage au format JSON :

```
{
  "name": "Joueur 2"
  "character": "103"
}
```

- Obtention de tous les joueurs et personnages enregistrés :

methode : GET

URL: */TeamFormationService*

donnée : JSON de la BDD décrit plus haut.

- Suppression d'un joueur de ses personnages :

methode : POST

URL: /TeamFormationService/delete_player

donnée : description du personnage au format JSON :

```
{  
    "name" : "Joueur 2"  
}
```


Table des figures :

Figure 1 - Worms Armageddon	1
Figure 2 - Texture ciel terrain et frontière	4
Figure 3 - Sprite Vegeta	4
Figure 4- Sprite Mio Kouzuki	5
Figure 5 - Diagramme des classes d'états	8
Figure 6 - Diagramme des classes de rendu.....	11
Figure 7 - Diagramme des classes de moteur de jeu	14
Figure 8 - Diagramme des classes d'IA.....	17
Figure 9 - Diagramme des classes pour la modularisation	20