

Projet Ninja Saga

Moïse BEUGRE – Ouajih DADAOUA



w

Figure 1 - Exemple du jeu Ninja Saga

1	Objectif	3
1.1	Présentation générale	3
1.2	Règles du jeu	3
1.3	Conception Logiciel	3
2	Description et conception des états	4
2.1	Description des états	4
2.2	Conception logiciel	4
2.3	Conception logiciel : extension pour le rendu	4
3	Rendu : Stratégie et Conception	6
3.1	Stratégie de rendu d'un état	6
3.2	Conception logiciel	6
3.3	Exemple de rendu	6
4	Règles de changement d'états et moteur de jeu	8
4.1	Stratégie de conception du moteur de jeu et règles de changement d'états	8
4.2	Conception logiciel	8
5	Intelligence Artificielle	
10		
5.1	Stratégies	
10		
5.1.1	Intelligence minimale	
10		
5.1.2	Intelligence basée sur des heuristiques	
10		
5.1.3	Intelligence basée sur les arbres de recherche	
10		
5.2	Conception logiciel	
10		
5.3	Conception logiciel : extension pour l'IA composée	
10		
5.4	Conception logiciel : extension pour IA avancée	
10		
5.5	Conception logiciel : extension pour la parallélisation	
10		
6	Modularisation	
11		
6.1	Organisation des modules	
11		
6.1.1	Répartition sur différents threads	
11		

6.1.2 Répartition sur différentes machines

11

6.2 Conception logiciel

11

6.3 Conception logiciel : extension réseau

11

6.4 Conception logiciel : client Android

11

Objectif

1.1 Présentation générale

L'objectif de ce projet est la réalisation du jeu "Ninja Saga", avec les règles les plus simples. Un exemple est présenté en Figure 1. Des combats entre personnages pourront être faits en tenant compte des points de vie, des points de mouvement en possession selon le personnage choisi.

Les personnages auront des caractéristiques qui influencent le combat, par exemple plus un personnage sera fort grand et imposant moins il sera rapide mais ses attaques auront plus d'impact.

Modes de jeu :

-Multijoueur : Player vs Player, chaque joueur choisit un personnage et initie ensuite le combat.

-Solo : Player vs IA , le joueur choisit un personnage et le personnage contre qui il veut se battre et initie un combat contre l'ordinateur (possibilité de choisir la difficulté).

-Survie : Player vs IA, le joueur choisit un personnage et va se battre contre tous les personnages du jeu en gardant à chaque fois le nombre de points de vie avec lesquelles il a terminé le combat précédent sachant que la difficulté augmente crescendo.

Type de personnages :

-Thor -> Fort, grand et robuste il se caractérise par ses attaques dévastatrices bien aidé par son marteau ravageur. Cependant il accuse un déficit en terme de vitesse et de réactivité.



-Flint -> Connu pour sa vitesse éclair ainsi que ses pyro-attaques, il se démarque par ses réflexes supersoniques qui lui permettent d'avoir toujours un temps d'avance sur ses adversaires.



-Seku -> Grand maître des arts martiaux il est connu pour ses attaques corps à corps redoutables et précises.



-Kuro -> Célèbre Ninja reconnu pour ses attaques furtives, maître du Kung fu et du Nunchaku, très véloce mais aussi moins robuste que ses compères.



Type de coups:

- Coup de poing & Coup de pied : ce coup cause des dommages valant 20 points de vie à l'adversaire sinon 10 point s'il est en position de défense.
- Uppercut & Flash Kick : ce coup cause des dommages valant 30 points de vie à l'adversaire sinon 15 points s'il est en position de défense.

L'upercut est très puissant pour repousser l'adversaire il a un seul désavantage c'est que sa zone d'impact est très réduite. Quant au Flash Kick, il est rapide et efficace. Il couvre un large champ d'action et limite les interventions et parades de l'opposant.

1.2 Règles du jeu

Au début de la partie, chaque joueur choisit un combattant et les combattants ont le même nombre de points de vie. La partie se termine lorsqu'un joueur n'a plus de points de vie.

1.3 Conception Logiciel

Les textures de personnages sont de tailles 100x100 pixels tandis que les textures de terrains de taille 700x400.

Les textures de personnages ont été ajustées de sorte à avoir quatres différentes lignes de sprites:

- La première ligne représente la position de "garde". C'est la position initiale du Fighter en début de partie.
- La deuxième ligne met en animation l'attaque "coup de poing" d'un Fighter.
- La troisième ligne est le mode "défense" qui permet de réduire les dégâts subis lors d'une attaque
- La dernière sera utilisé lorsque le Fighter n'a plus de points de vie. C'est le statut "DEAD".



Figure 2 - Textures pour les personnages



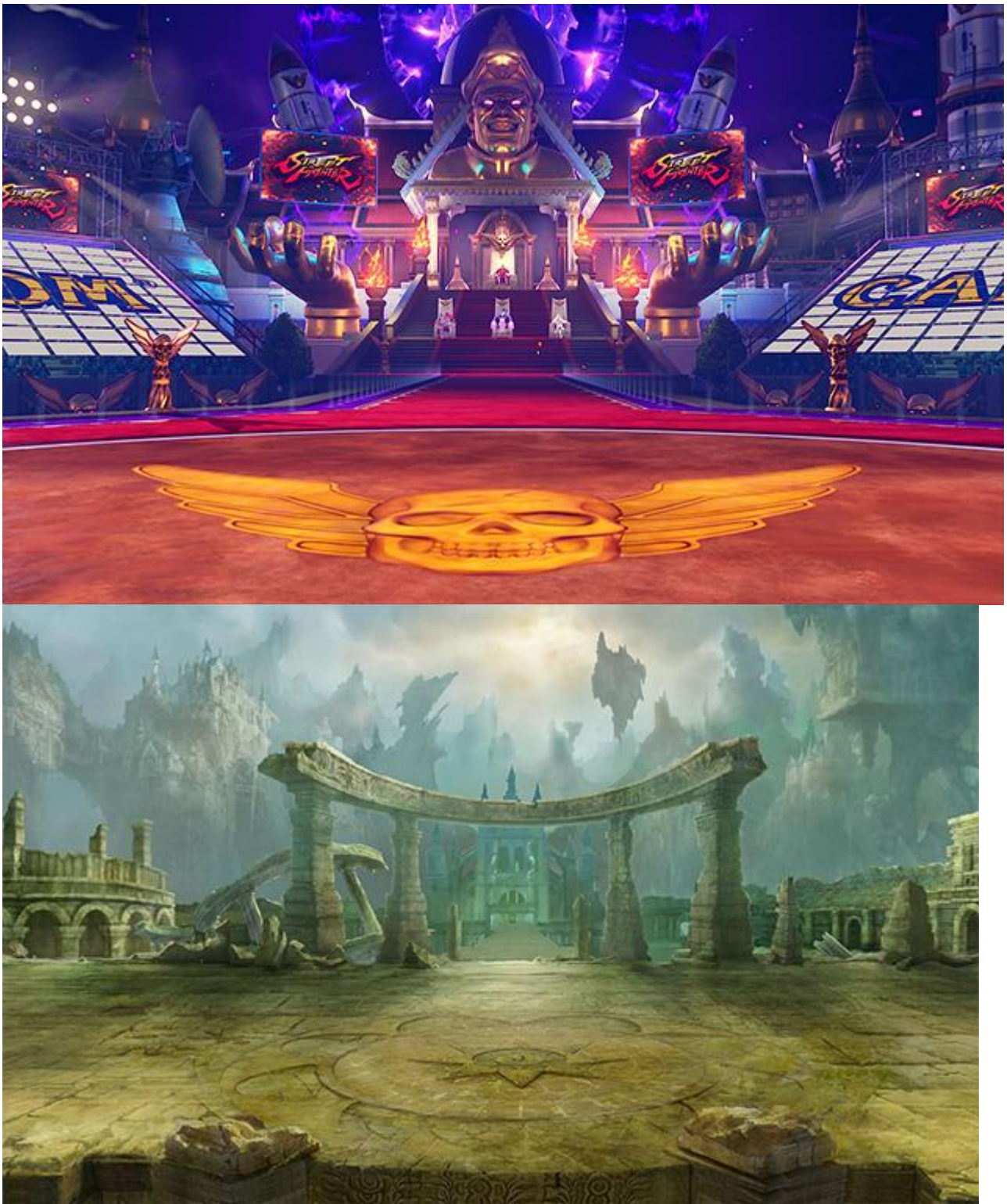


Figure 3 - Textures pour les terrains de combat



Figure 4 - Textures pour les tuiles de lettres et chiffres

2 Description et conception des états

2.1 Description des états

Un état du jeu est formé d'un ensemble d'éléments mobiles (les combattants) dont l'état varie au cours de la partie et d'un décor statique.

2.1.1 Etats des combattants

Un combattant est soit contrôlé par un joueur soit contrôlé par une Intelligence Artificielle (I.A), il possède un nombre de points de vie à l'état t, un nom et un ensemble de statistiques (Points de vie, Points de magie, Combo) qui lui sont attribués en début de partie. Lorsqu'un combattant effectue un nombre de coups réussi dépassant un seuil, il entre dans un statut *Spécial* qui lui donne droit à des attaques plus puissantes. Le combattant est considéré comme étant "mort" si son nombre de points de vie atteint 0.

2.1.1 Etats de l'arrière plan (BackgroundManager)

L'arrière plan représente un terrain attribué à chaque combattant. Les combattants sont donc favorisés lorsque le combat se déroule sur le terrain qui leur ait attribué. Les statistiques d'attaques et de défense sont améliorées.

Le passage d'un terrain à un autre peut représenter le passage à un niveau supérieur lors d'une partie.

2.2 Conception logiciel

Le package état peut se diviser en trois sous-partie:

- Une partie gérant les personnages, en rouge
- Une partie gérant l'environnement, bleu
- Une classe représentant l'état global du jeu, en orange

La classe Player contient l'ensemble des éléments permettant de caractériser l'état d'un joueur. Chaque joueur est lié à un combattant par une relation de composition : un combattant ne peut pas exister sans joueur. Dans le cas où le combattant est contrôlé par l'IA, l'IA est considérée comme un joueur et contrôle donc son combattant.

Pour notre implémentation du jeu, nous avons prévu d'utiliser les quatre combattants suivant :

- Thork
- Flint
- Seku
- Kuro

Le constructeur Fighter permet d'avoir les quatre types de combattants en utilisant les différents identifiants qu'il a à sa disposition.

2.3 Conception logiciel : extension pour le rendu et moteur de jeu

On utilise le pattern Observer pour notifier les éléments dépendant de l'état lorsque celui-ci change. La classe State hérite de la classe Observable qui contient une liste d'objets héritant de la classe abstraite Observer.

Lorsque la classe State subit un changement, l'utilisateur utilise la méthode notifyObservers() pour notifier ses observers avec l'évènement correspondant au changement appliqué.

Il existe différents types d'Events:

- Les FighterEvent qui représente un changement d'état lié au statut des combattants (En vie, mort, en mode super) .
- Les environnement Event qui correspondent à des changement liés à l'environnement (changement de niveau) ou à des changements liées aux statistiques de la partie.

- Dans le cas où les fighters et le terrain change l'évènement est appelé :
ALLCHANGED

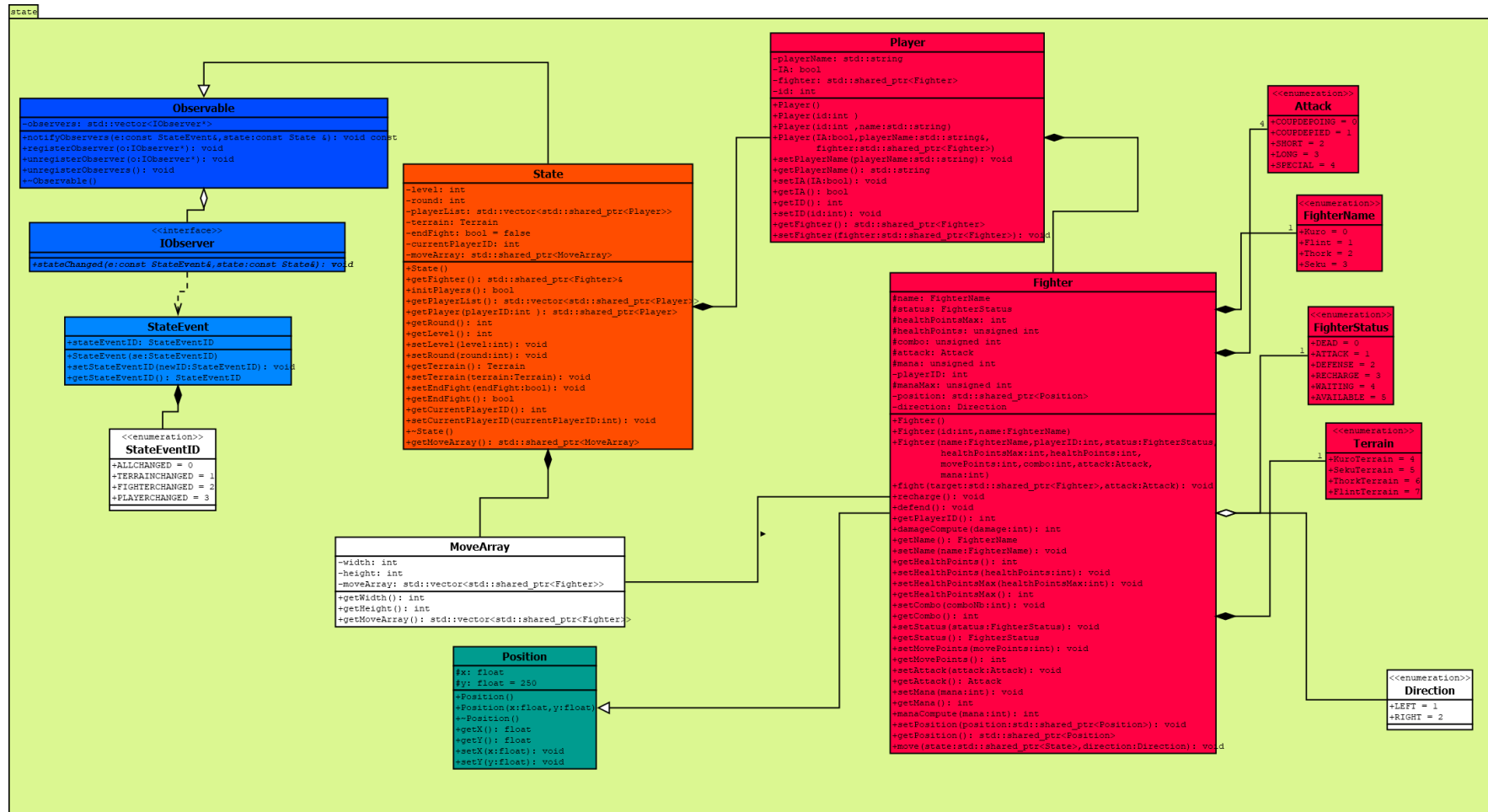


Figure 5 -Diagramme des classes d'état

3 Rendu : Stratégie et Conception

3.1 Stratégie de rendu d'un état

Notre stratégie de rendu d'un état se base sur l'utilisation de l'interface SFML qui s'appuie sur OpenGL afin de générer un rendu en 2D. Nous avons utilisé les fonctions de SFML afin de charger dans le processeur, la liste des éléments à afficher par le processeur graphique.

Nous avons découpé notre affichage graphique sur deux niveaux, l'arrière plan qui représente le terrain de combat et les éléments mobiles à savoir les combattants (Thork, Flint, Seku, Kuro) qui se superposent sur la couche précédente. On transmet l'état du jeu ainsi que les textures des différents terrains et la texture des combattants à afficher.

Lorsque qu'un changement d'état se produit, la vue est modifiée en fonction du changement appliqué à l'état. Si le changement modifie uniquement la position des combattants, seule le rendu des combattants est mis à jour, si ce changement s'applique à l'environnement l'ensemble du rendu de la carte est mis à jour. Lorsque l'ensemble de l'état est modifié le rendu de l'état est entièrement mis à jour. Dans le cas où le jeu est fini la fenêtre est automatiquement fermée.

3.2 Conception logiciel

Pour afficher un état, on utilise les cinq classes suivantes: **TileSet**, **TextureManager**, **BackgroundManager**, **FighterRender** et **StateLayer**.

La classe **TextureManager** est une classe singulière qui va nous permettre de récupérer une instance du Tile que l'on va dessiner (soit un Tile de terrain ou de fighter) à partir de la classe **TileSet** qui charge les textures. Cette instance est ainsi appelée dans les classes **BackgroundManager** et **FighterRender** qui nous permettent de dessiner les bonnes textures. Enfin, la classe **StateLayer** effectue un travail de synthèse en positionnant et dessinant les différentes textures récupérées dans **BackgroundManager** et **FighterRender** constituant ainsi le rendu final.

3.3 Exemple de rendu

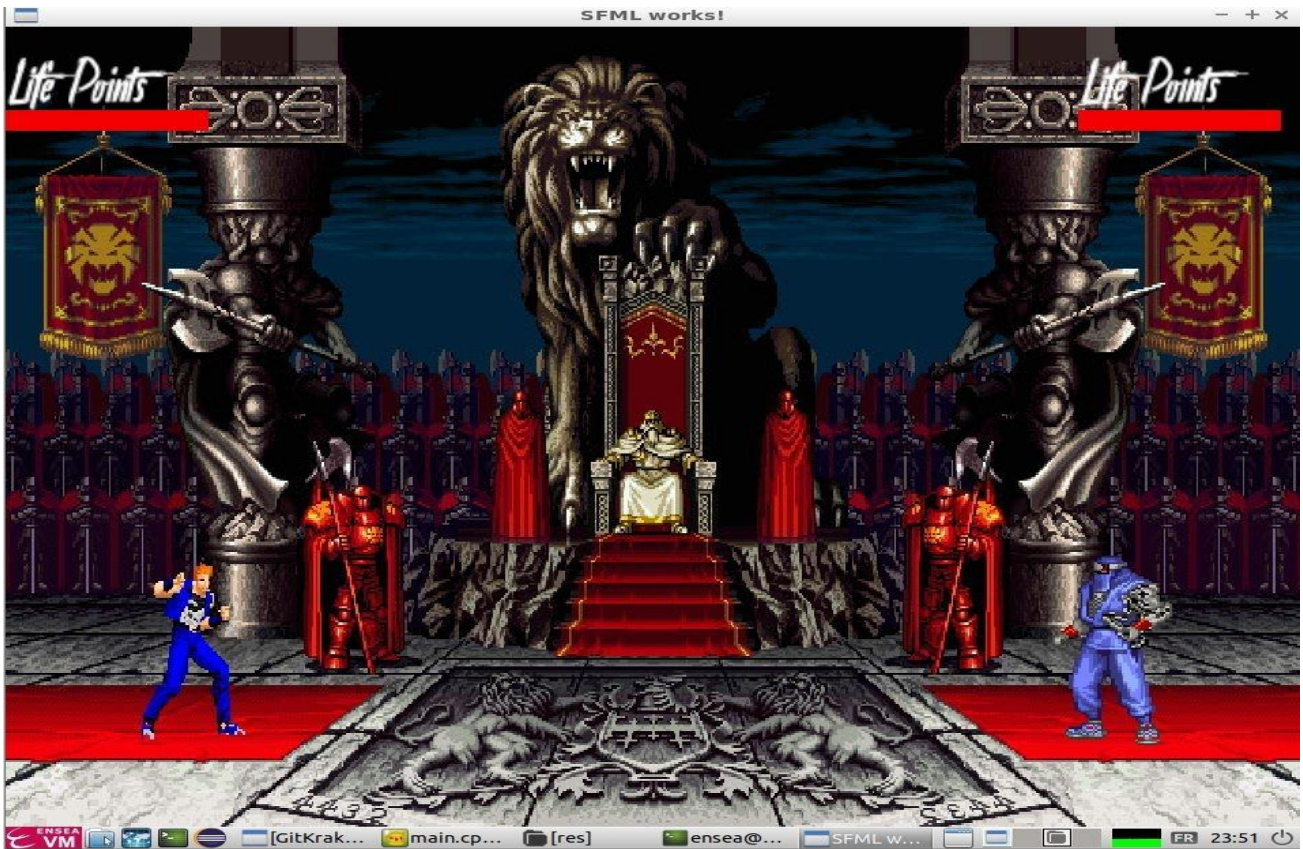


Figure 6 - Exemple de rendu

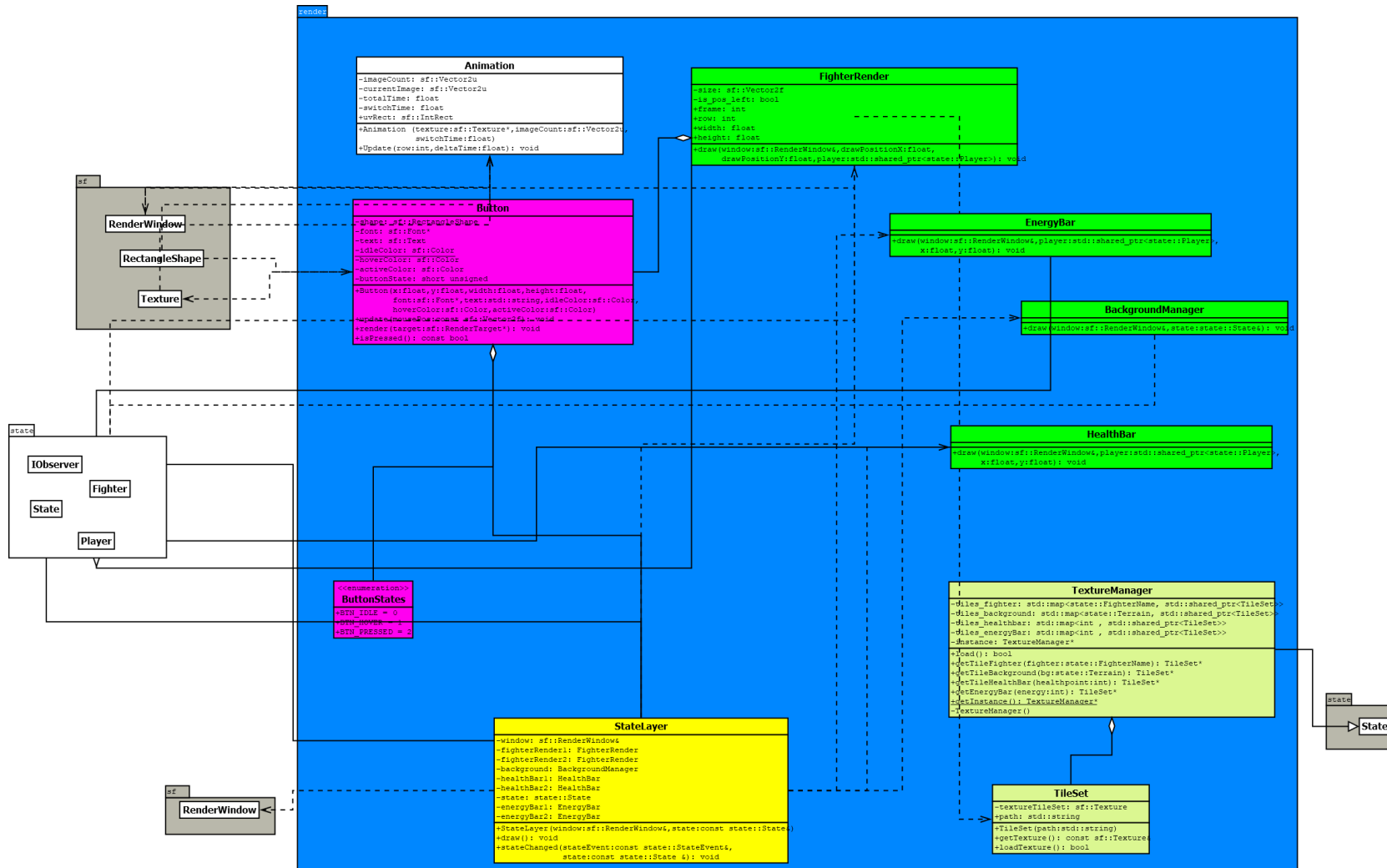


Figure 7 -Diagramme de classes pour le rendu

4 Règles de changement d'états et moteur de jeu

4.1 Stratégie de conception du moteur de jeu et règles de changement d'états

En début de partie, tous les fighters ont le statut "AVAILABLE". Ils peuvent donc attaquer, défendre ou recharger en cas de points d'énergie (mana) insuffisant. Une fois qu'un fighter a fait une action, c'est la fin de son tour et c'est le tour du fighter adverse.

Chaque fighter peut effectuer 3 types d'action lorsque son état est AVAILABLE c'est-à-dire disponible :

- attaquer : le fighter peut infliger des dégâts à un fighter. Le fighter aura le statut ATTACK. La touche "A" permet d'exécuter cette commande.
- défendre : le fighter peut choisir de se mettre en position de défense pour anticiper une potentielle attaque de l'adversaire. Ce statut va lui permettre de sauver des points de points de vie. Son statut à l'issue de cette action est DEFENSE. La touche "D" permet d'exécuter cette commande.
- recharger : le fighter peut choisir de recharger ses points d'énergie (mana) pour effectuer des attaques aux tours suivants. Son statut est RECHARGE. La touche "R" permet d'exécuter cette commande.
- terminer son tour: cette action permet au joueur adverse de jouer. La touche "T" permet d'exécuter cette commande.

4.2 Conception logiciel

Le diagramme des classes pour le moteur du jeu est disponible à la page suivante. Ce diagramme est constitué de deux classes principales Engine et Command.

Classe Engine: C'est le moteur du jeu en lui-même. Cette classe contient l'état du jeu et la liste des commandes qui devront être exécutées. Cette liste est un

tableau associatif `std::map` car un nombre entier est associé à chaque action pour pouvoir indiquer son niveau de priorité.

Une fonction `"addCommand"` permet d'enregistrer chaque nouvelle commande avec sa priorité. La méthode `"update"` va permettre lorsqu'un tour complet est terminé d'exécuter les commandes enregistrées dans le tableau associatif. Il va aussi prévenir le `"render"` qu'il y a eu des changements sur l'état en appelant le `"notifyObserver"` de `state`.

Cette méthode fait appel à `stateChanged`, fonction abstraite d'`Observer` dont le fils est `stateLayer` de `render`. Ainsi dans `stateLayer` la fonction `stateChanged` est appelée. Cette fonction lance des fonctions de `render` pour modifier l'affichage selon le `state` passé en argument. `"update"` va aussi vider le tableau associatif de toutes ses données pour que celui-ci soit prêt à accueillir un nouveau tour d'actions.

Classe Command: Cette classe va permettre de définir un type commun pour pouvoir enregistrer toutes les actions dans une même `std::Map`.

Les actions héritent donc toutes de cette classe et ont un format très similaires.

Chacune des classes **AttackCommand**, **RechargeCommand**, **ChangeCommand** et **DefenseCommand** ont une méthode `"execute"` qui va modifier l'état actuel.

Chacune de ces classes va enregistrer les informations dont il a besoin dans des attributs sur lesquels `"execute"` pourra agir.

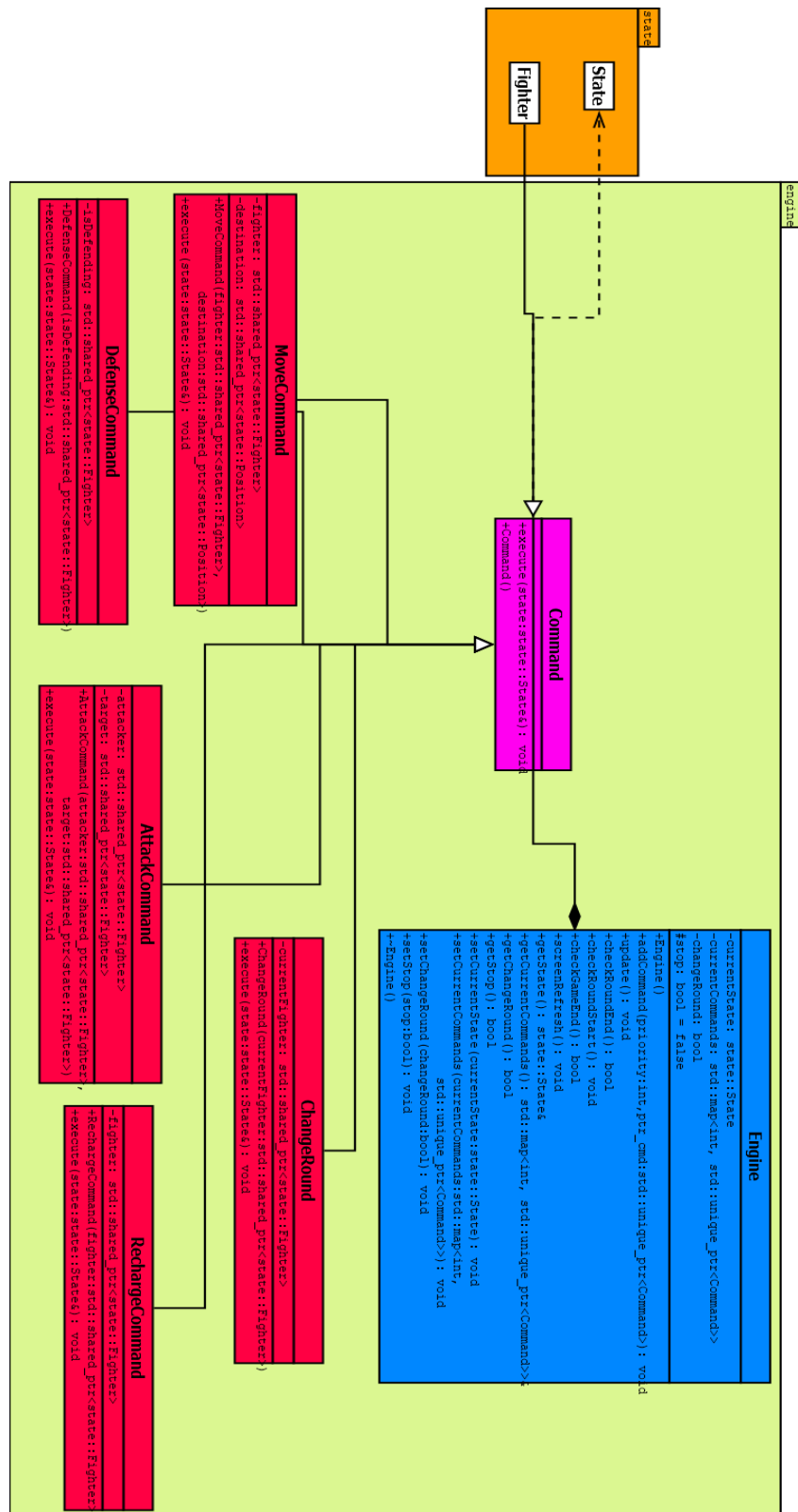


Figure 8 - Diagramme de classes pour le moteur

5 Intelligence Artificielle

L'IA est considérée comme un joueur avec une liste de commandes à exécuter. La commande à exécuter est choisie selon l'algorithme de l'intelligence Artificielle appliquée.

5.1 Stratégies

5.1.1 Intelligence minimale

Cette stratégie est la même pour tous les personnages. De manière aléatoire, on choisit une action parmi les trois possibles et l'IA l'exécute. Cette IA sera comparée par la suite à une IA basée sur des heuristiques puis à une autre basée sur une autre basée sur des arbres.

5.1.2 Intelligence basée sur des heuristiques

Nous proposons ensuite un ensemble d'heuristiques pour offrir un comportement meilleur que le hasard et donner une chance à l'IA de remporter la partie.

On privilégie presque toujours l'attaque à la recharge d'énergie. Et ce dans la mesure où le « fighter » a assez de points d'énergie pour attaquer.

L'IA défend lorsque que ses points de vie sont inférieurs au quart des points de vie initiale.

L'IA recharge lorsque ses points d'énergie sont insuffisants pour lui permettre de faire une attaque.

L'IA attaque si :

- elle n'est pas en danger
- si elle est en danger mais elle dispose d'assez de points de vie pour faire une attaque qui peut lui permettre de remporter la partie.

Si l'IA peut attaquer mais que ses points de vie sont trop faibles, elle va prioriser la défense. Dans le cas où, l'IA ne peut pas attaquer, la défense est prioritaire.

5.1.3 Intelligence basée sur les arbres de recherche

Cette IA est basée sur la théorie des arbres de recherche. Dans notre cas, l'algorithme minimax est plus adapté pour établir une IA plus efficace que l'IA heuristique précédente. Un système de « scoring » est mis en place pour évaluer l'impact de chaque action sur le jeu. Le score est calculé en fonction des points de vie et des points d'énergie. Plus le score est important, plus cette action sera prioritaire.

5.2 Conception logiciel

L'IA va d'abord vérifier les actions possibles du fighter en tenant compte de ses points de mana et de ses points de vie.

Classe Random_ai

C'est la classe qui implémente l'IA Random.

Pour le choix de l'action, un entier va prendre une valeur aléatoire entre 0 et 2.

Selon le résultat, l'IA effectuera une action:

- 0 pour l'attaque,
- 1 pour la défense
- 2 pour la recharge.

Classe Heuristic_ai

C'est la classe qui implémente l'IA Heuristic.

Le choix des actions de l'IA tient compte des points de vie de l'IA et de ses points d'énergie(mana). Les points de vie et les points d'énergie du fighter adverse sont aussi pris en compte.

Classe Deep_ai

C'est la classe qui permet d'évaluer les différentes actions possibles de l'IA en calculant un score. L'algorithme minimax est appliqué en utilisant les méthodes maximiseScore and minimiseScore.

La fonction GetCommand permet de récupérer la commande choisie par l'utilisateur et de l'évaluer.

Classe NodeDeep_ai

Cette classe se charge de construire l'arbre en ajoutant les différents nœuds et els différents niveau de celui-ci. Une copie de l'état est faite afin d'évaluer les différents score à chaque nœud.

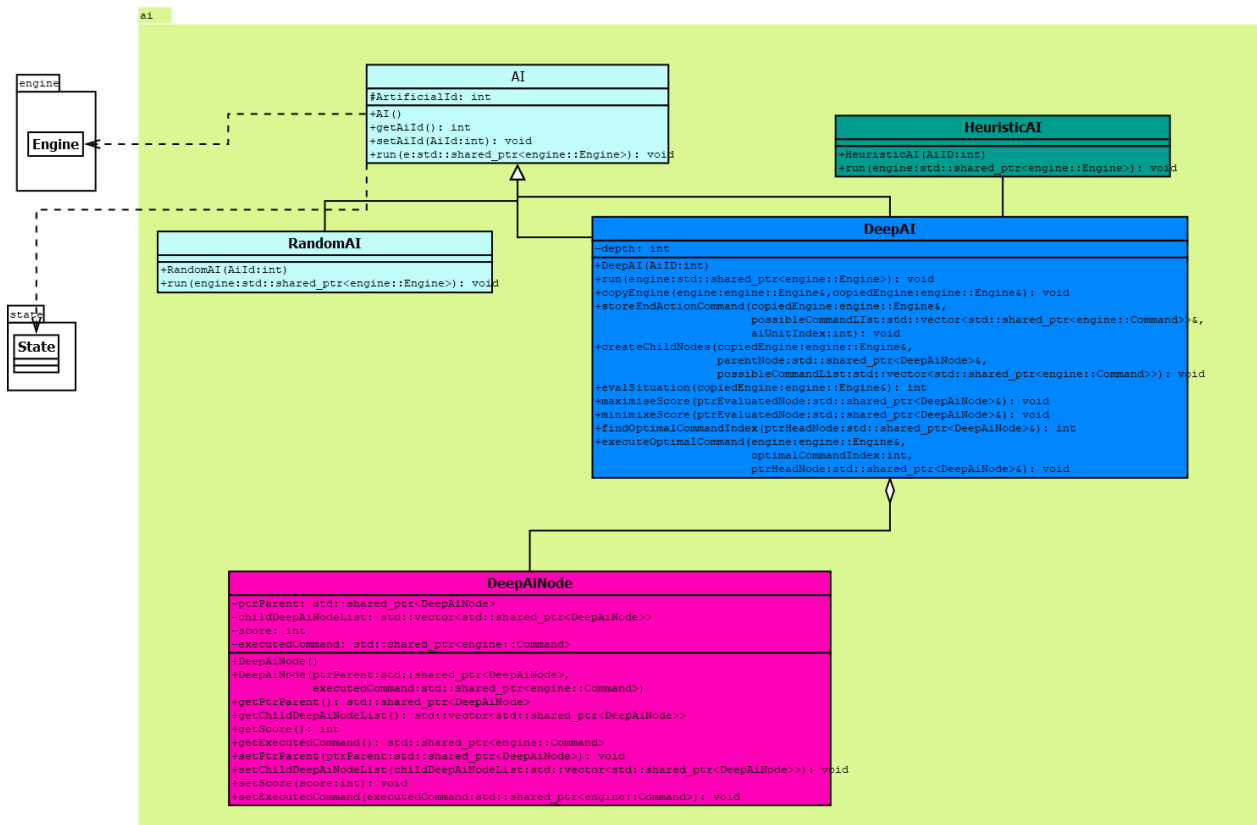


Figure 9 - Diagramme de classes pour l'Intelligence Artificielle

6 Modularisation

Cette section se concentre sur la répartition des différents modules du jeu dans différents processus. Deux niveaux doivent être considérés. Le premier est la répartition des modules sur différents threads. Notons bien que ce qui est attendu est une parallélisation maximale des traitements: il faut bien démontrer que l'intersection des processus communs ou bloquant est minimale. Le deuxième niveau est la répartition des modules sur différentes machines, via une interface réseau. Dans tous les cas, motivez vos choix, et indiquez également les latences qui en résulte.

6.1 Organisation des modules

6.1.1 Répartition sur différents threads

6.1.2 Répartition sur différentes machines

6.2 Conception logiciel

6.3 Conception logiciel : extension réseau

6.4 Conception logiciel : client Android

Illustration 4: Diagramme de classes pour la modularisation

