Projet OUAW

Figure 1 - Logo du jeu

Léo GIMENEZ

Raphaël PICHON

Luc WERNER

Charlie XANH

Table des matières

1- Présentation

Dans le cadre de notre Projet Logiciel Transversal, il nous est demandé de réaliser un jeu vidéo tour par tour permettant d'exploiter les modules étudiés au cours de l'année : Génie Logiciel, Algorithmique, Programmation Parallèle et Web Services.
Notre groupe a décidé de baser son jeu sur ***For the King*** qui est un RPG -Roguelite alternant déplacement et combat, les deux en tour par tour.

1.1 Définition de notre jeu

RPG - "Role Play Game" : Traduit en "Jeu de Rôle", cela définit un genre de jeu vidéo où les joueurs incarnent un ou plusieurs personnages qui évoluent au fil d'une quête.

Roguelite : Ce terme définit un *sous-genre* du jeu vidéo. Ce sous-genre est dérivé du terme *Roguelike*, qui définit un jeu vidéo avec les caractéristiques suivantes :

- Une mort permanente : le personnage joué n'est pas sauvegardé d'une partie à l'autre.
- Des niveaux procéduraux : les niveaux sont générés de façon procédurale.
- Des combats tour par tour.
- Pas de fin.

Dans notre cas plusieurs de ces caractéristiques ne sont pas respectées, comme par exemple **les niveaux procéduraux** ou **l'absence de fin**, ce qui fait de notre jeu un *Roguelite*.

1.2 Règles de notre jeu

Nous avons choisi d'appeler notre jeu ***OUAW : "Once Upon A WEI"***, les joueurs y incarnent des étudiants fictifs de l'ENSEA qui affrontent des épreuves dans le but de valider leurs années et retrouver le WEI perdu depuis des temps immémoriaux. Les règles sont les suivantes :

- n-joueurs jouent chacun un élève au tour par tour
- A chaque tour le groupe lance un dé pour se déplacer
- A chaque étape des PO est donnée comme monnaie
- Possibilité de récupérer de l'équipement à la K-fet en échange de PO –
- Le groupe perd lorsqu'il ne valide pas son année
- Chaque joueur à des caractéristiques et des équipements qui lui permettent de passer les épreuves lors d'une phase de combat.

Nous n'avons pas, pour le moment, définie un système d'équilibre concernant les caractéristiques, les dégâts, etc… Cependant les classes sont prévues pour recevoir un tel équilibrage.

1.3 Ressources

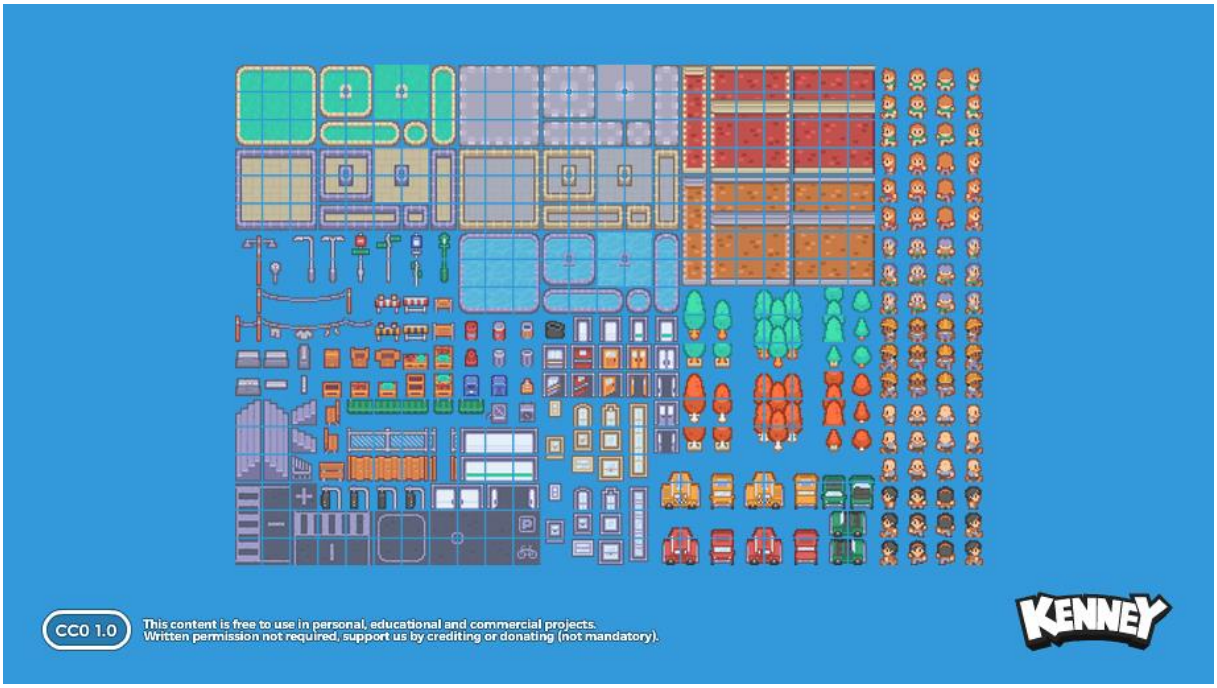


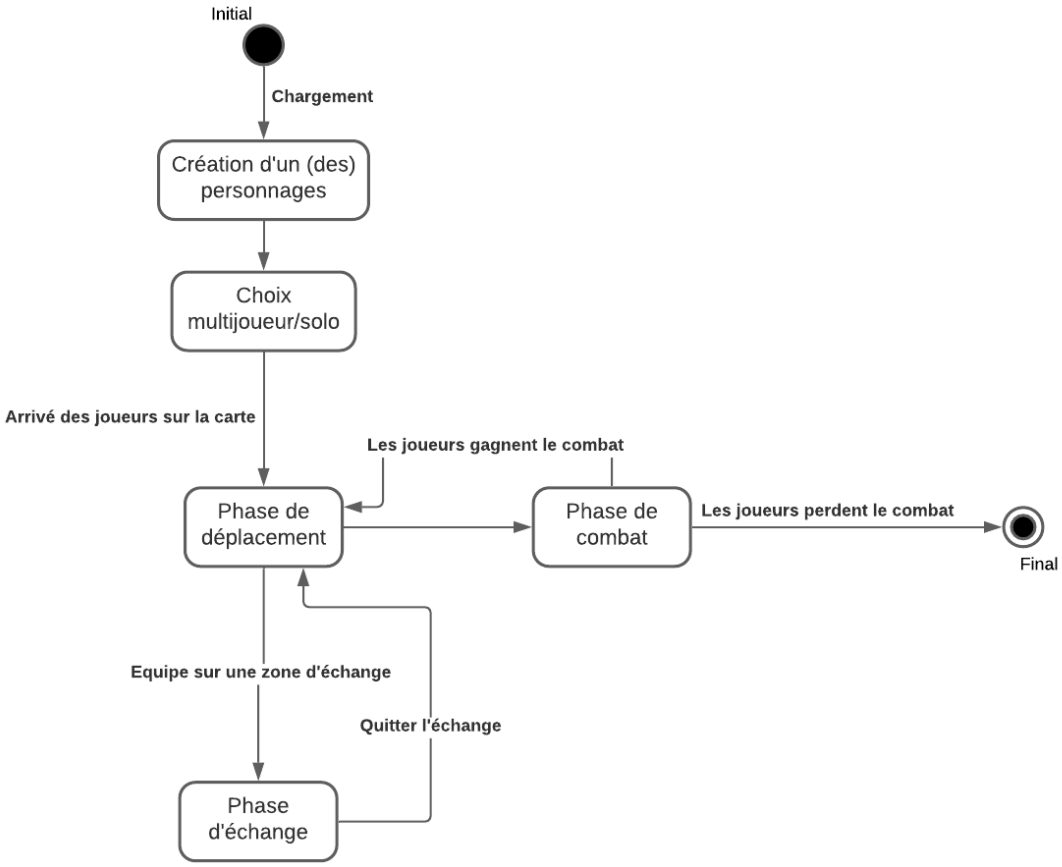
Figure 2 - Textures pour le jeu

Source : https://kenney.nl/assets/rpg-urban-pack

2- Description et états

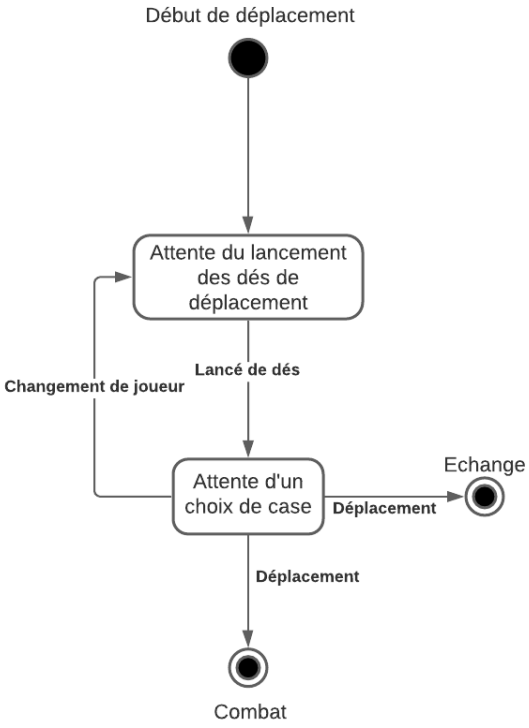
La partie se déroule donc lors de 4 phases de jeu, elles sont décrites dans les diagrammes d'états ci-dessous :

2.1 Déroulement global d'une partie



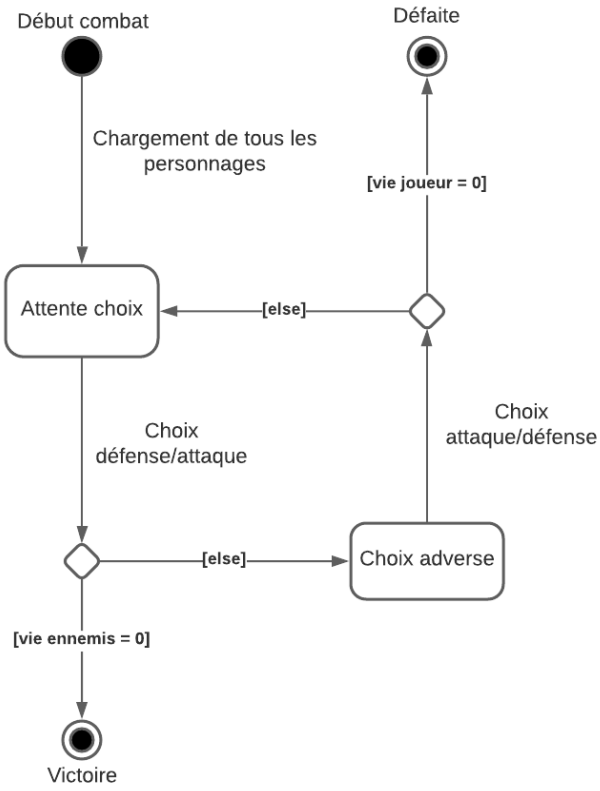
La partie commence par deux menus permettant de choisir son personnage et de jouer en multi joueurs. Puis la partie tourne autour de trois phases, celle de déplacement, de combat et d'échange. On note que la partie se termine uniquement lors d'une phase de combat.

2.2 Phase de déplacement



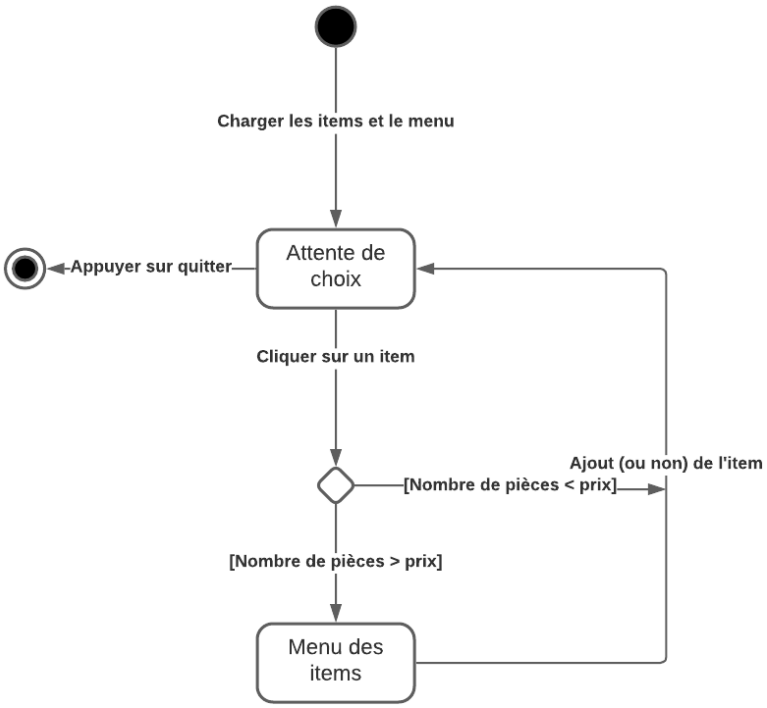
La phase de déplacement tourne en boucle tant que le groupe n'est pas arrivé à une phase de combat ou une phase d'échange.

2.3 Phase de combat



La phase de combat est une phase classique de combat au tour par tour. Tant que le joueur ou l'adversaire est en état de combattre, le combat continu.

2.4 Phase d'échange



La phase d'échange permet d'échanger de l'argent contre des items.
Cette phase se termine quand le ioueur le décide.

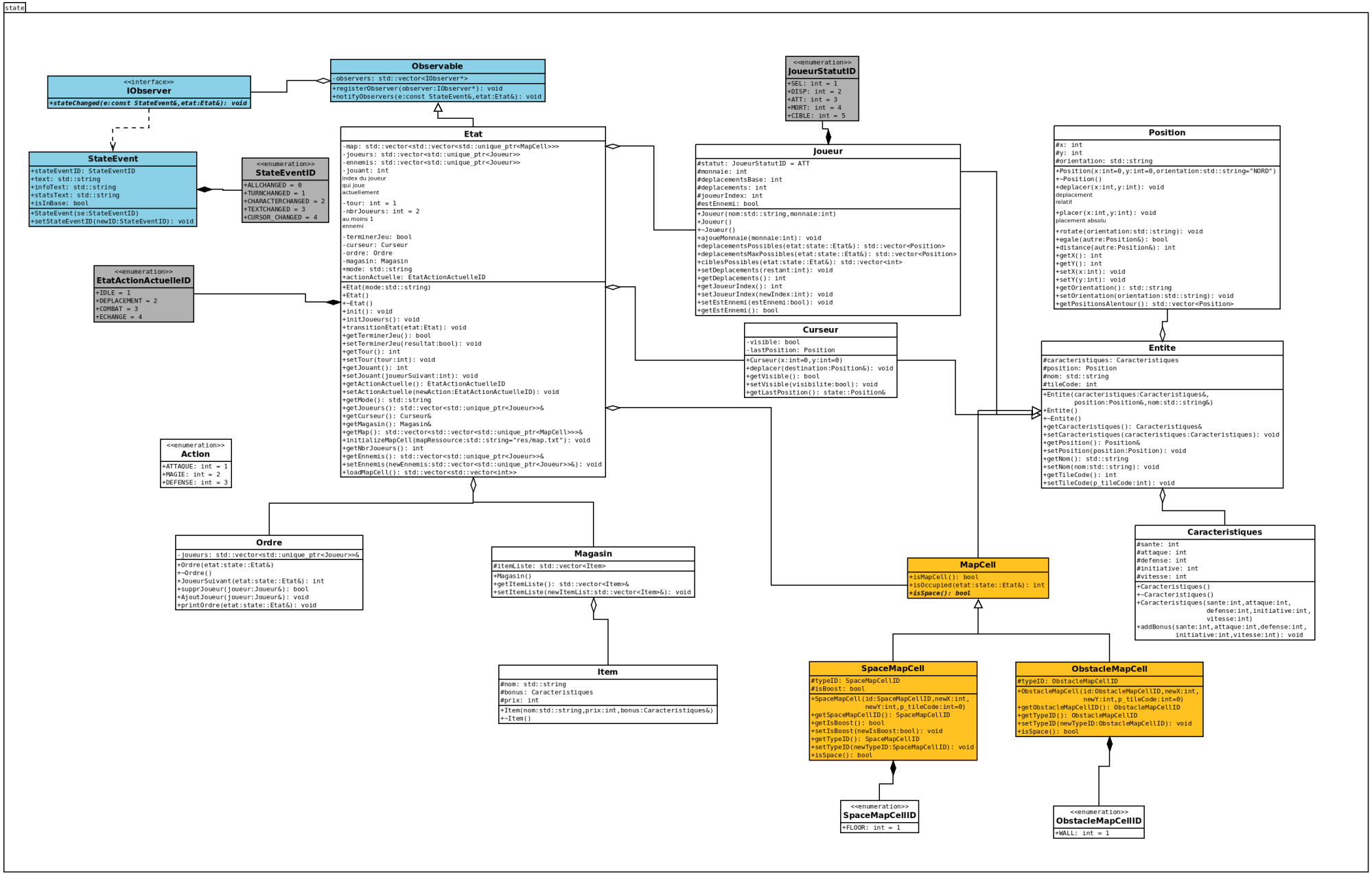


Figure 3 - Diagramme d'état

3- Rendu : Stratégie et conception

3.1 Stratégie de rendu

Le rendu correspond à la partie qui va gérer l'aspect graphique du jeu. Nous devons récupérer l'état de la partie et afficher les informations nécessaires à l'écran.

Nous utiliserons la librairie SMFL2.0 pour toute la partie rendue.

La stratégie de rendu est de découper une scène à afficher en plusieurs plans superposés. La stratégie reste identique suivant la situation de l'état (map, échange ou combat). Nous allons afficher un ensemble de textures (Tuiles) suivant une certaine position appartenant toutes à un plan (layer).

Le dernier plan à afficher sera celui du décor de fond stocké (ou non) sous la forme d'un fichier texte suivant l'état, chargé et mis à jour au centre du joueur. Le second plan est celui des personnages. Le premier plan est celui des menus et interactions de l'utilisateur.

Pour le dernier plan :

- Lors d'un déplacement sur la carte, la stratégie est d'observer la position du joueur sur la carte et de centrer l'affichage en conséquence. Le fond ne changera que lorsque le joueur se déplace. Pour une situation de combat ou d'échange, le fond est fixe et sera inchangé.

Pour le second plan :

- Les personnages seront affichés directement sur la surface précédente sous forme de sprites. Une animation sera possible pour rendre le rendu plus dynamique lorsque le joueur doit prendre une décision. Une fréquence de rafraichissement des personnages (environ 30Hz) suivant une routine de sprites à afficher sera mis en place.

Pour le premier plan :

- Le premier plan est l'ensemble des menus avec lesquels l'utilisateur peut interagir. Ils seront donc rafraîchis si l'utilisateur souhaite effectuer une action. Les textures seront donc modifiées suivant la position de la sourie et de son interaction avec l'interface.

3.2 Conception du rendu

Voir figure … pour le die de rendu

Le cœur de notre rendu se situe autour des layer, chaque layer (3) est spécialisé dans son rendu et possède des attributs uniques. Par exemple le layer charger de la map n'aura que des tiles de type MapTile. Chaque tile possède une texture et une position. Ainsi à l'aide d'un appel successif sur l'ensemble des textures des tiles appartenant aux layers, il est possible de créer la surface finale à afficher lors d'un appel de getScene(). Les différentes classes sont les suivantes :

Scene :

Contient la boucle d'affichage et les textures finales qui seront affiché à l'écran. La Scene vient synthétiser la partie rendue en proposant d'actualiser la texture finale à l'aide de setLayer() et getLayer. La fonction draw scene vient donc afficher les textures qui ont été actualisées.

Layer :

Contient un ensemble de tiles, chaque layer correspond à une couche à afficher et sera synthétisé sous forme d'une texture par Scene. getLayer() permet d'obtenir la surface associée à l'ensemble des tiles qu'elle contient.

Tile :

Un tile est un élément de layer qui possède des coordonnées pour être positionné sur l'écran. Ainsi pour l'afficher la carte une matrice de tile sera nécessaire. Seul un vecteur de tile sera nécessaire pour les Player et les Menus. Ces Tile pouvant posséder des animations, un vecteur de texture est disponible dans les menus et les players. Ainsi chaque Texture de ce vecteur correspond à une orientation pour notre personnage, ou à une interaction avec l'utilisateur.

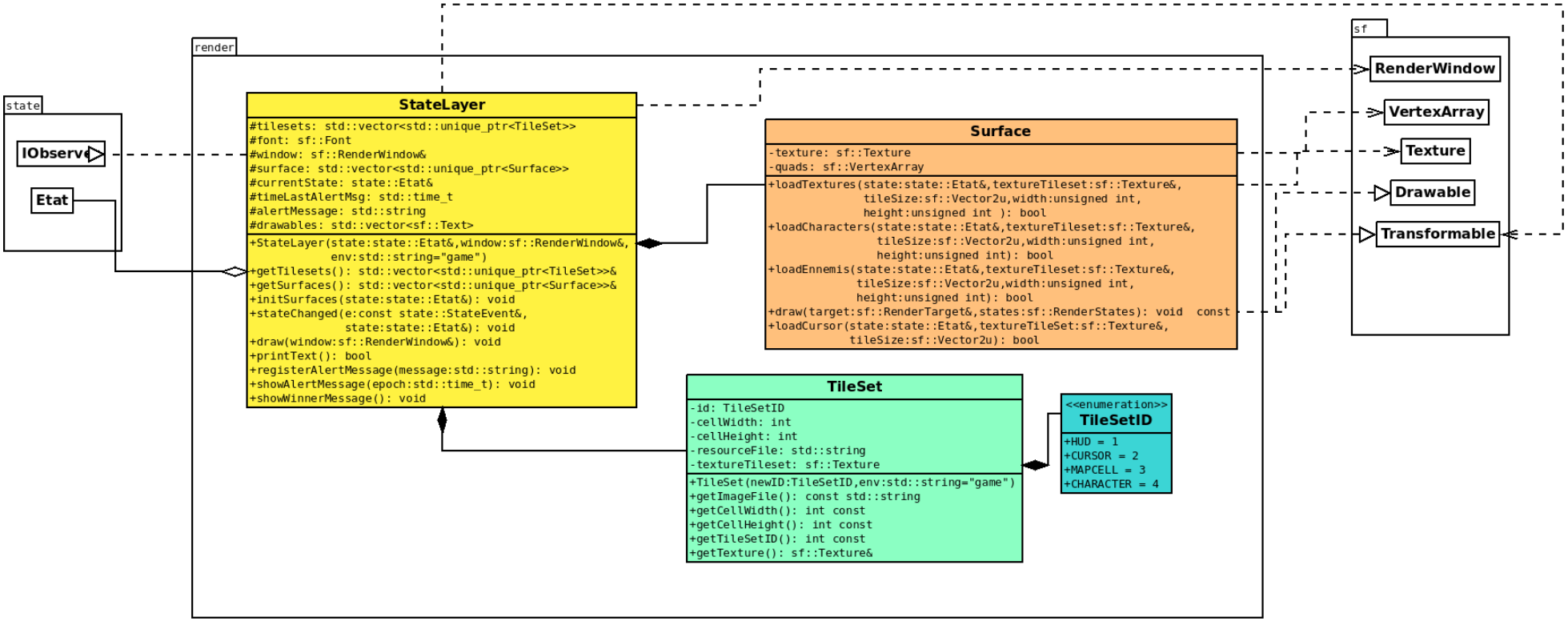


Figure 4 - Diagramme de rendu

4- Règles de changements d'états et moteur de jeu

4.1 Changement extérieur

Changements provoqués par le joueur.

- En mode déplacement : Déplacement du joueur.
- En mode combat : Sélection dans un menu, Utilisation d'une action,
  Action fuite -> Passage en mode déplacement.
- En mode échange : Sélection dans un menu, Confirmation d'un achat,
  Quitter l'échange -> passage en mode déplacement.

4.2 Changements autonomes

Changements effectués de manière automatique par le système (création/mise à jour d'un état) suite à une action extérieure.

- En mode déplacement :
  - Si le joueur est sur une case en bord de map -> Changement de map.
  - Si le joueur est sur une case ennemie -> Passage en mode combat.
  - Si le joueur est sur une case « magasin » -> Passage en mode échange.
- En mode combat :
  - Mise à jour des caractéristiques des personnages en fonction des règles (Pv, attaque…).
  - Si le combat est terminé perdant -> Game Over.
  - Si le combat est terminé gagnant -> Passage en mode déplacement.
  - Action de l'IA (sélection de cible et action).
- En mode échange :
  - A définir si besoin.

1. Conception Logiciel

L'ensemble du moteur de jeu repose sur un patron de conception de type Commande.

Le rôle de classe Commande est de représenter une commande. A cette classe est associée un CommandeId pour identifier le type de commande.
Les différentes commandes sont adaptées en fonction du mode de jeu. Elles héritent toutes de la classe Commande.

En mode combat :
La classe AttaquerCommande correspond à l'attaque d'un personnage et la classe.
La classe VictoireCommande permet de savoir le joueur gagnant.
La classe TerminerTourCommande permet la fin d'un tour.

En mode déplacement :
La classe DeplacementCommande représente le déplacement d'un personnage.

En mode échange :
La classe SelectionnerItemCommande permet de savoir quel objet est choisi dans le magasin.
La classe AcheterItemCommande permet d'acheter un objet.
La classe Engine est le coeur du moteur. Elle stocke les commandes dans une std::map avec une clé entière. Ce mode de stockage permet d'introduire une notion de priorité : on traite les commandes dans l'ordre de leurs clés, de la plus petite à la plus grande. Lorsqu'un nouveau tour démarre, Lorsque l'on a appelé la méthode update() après un temps suffisant, le moteur appelle la méthode execute() de chaque commande, incrémente le nombre de tours, met à jour les statistiques des personnages puis supprime toutes les commandes.

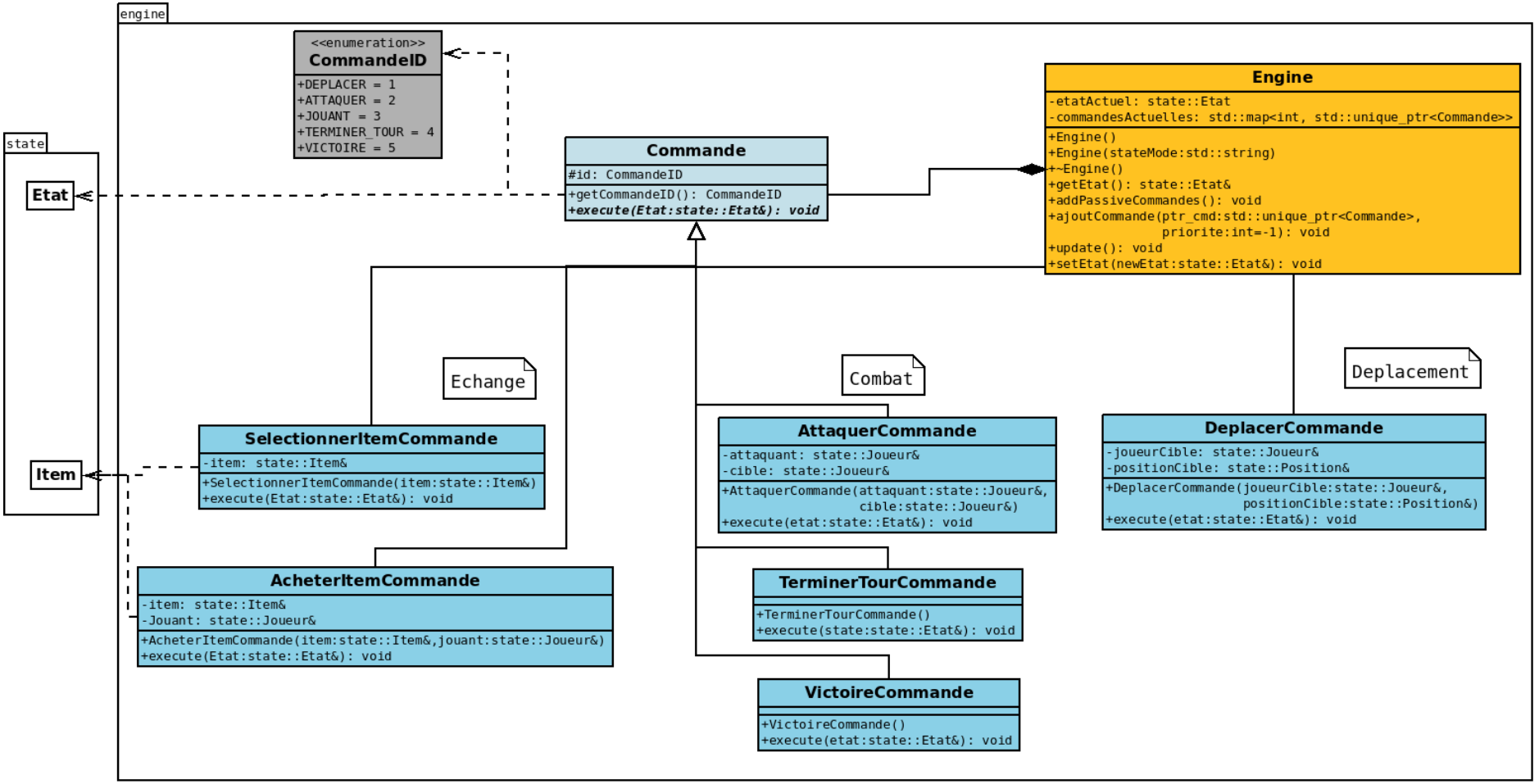


Figure 5 - Diagramme du moteur de jeu

5- Intelligence artificielle

5.1 Stratégie simple et intelligence aléatoire

La stratégie simple de notre intelligence artificielle ce base sur les paternes suivants :

En déplacement :

- Au début de son tour de déplacement, l'IA se déplacera aléatoirement dans une zone prédéfinie et restrictive. Le nombre de cases que parcourra l'IA lors de son déplacement sera déterminé aléatoirement en fonction de la portée maximale du personnage incarné. L'IA sera bloquée par les éléments infranchissables et donc cela réduira ses possibilités.

En combat :

- Au début de son tour de combat, l'IA va déterminer l'action qu'elle va effectuer de façon aléatoire parmi les actions possibles. Si l'IA doit choisir une cible, lors d'une attaque par exemple, la cible sera choisie aléatoirement aussi.

5.2 Conception Logicielle

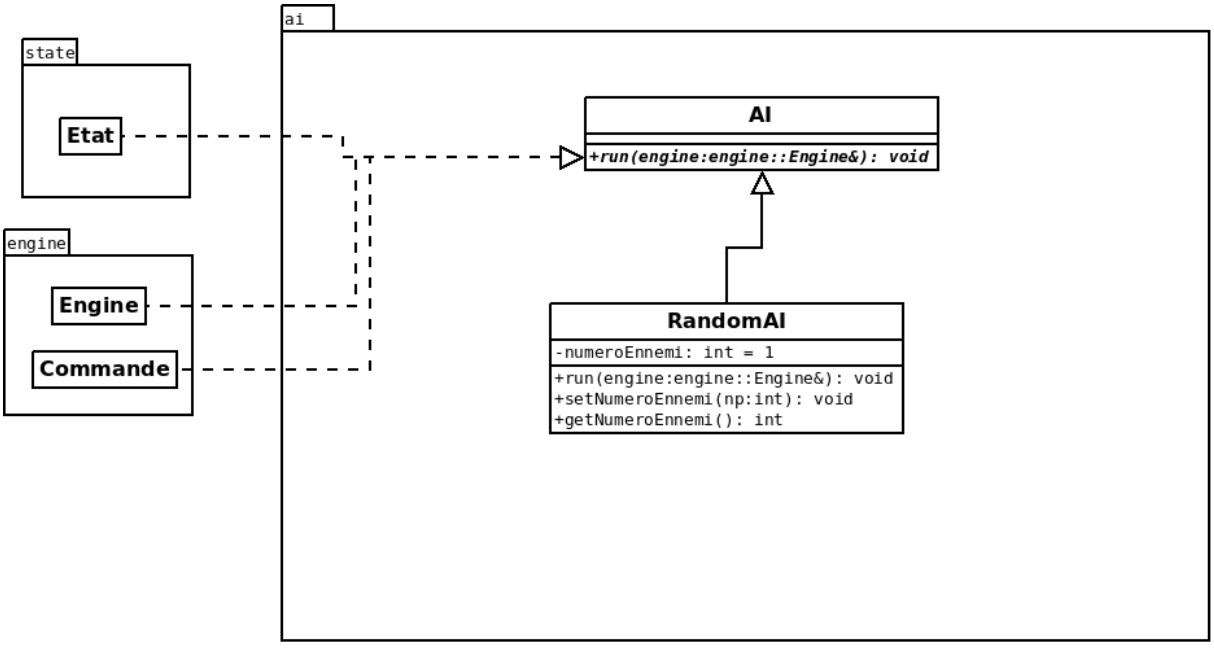


Figure 6 - Diagramme de l'IA Simple

5.3 IA de déplacement

5.3.1 A étoile

L'algorithme de A étoile (A star) est basé sur un algoritme de Dijkstra, il est plus rapide et plus efficace que ce dernier. Nous avons besoin d'un point de départ et d'arriver dans une carte ayant des murs et des zones accessibles, ici un vecteur de vecteur d'entiers de 0 (accessibles) et 1 (murs). Nous

avançons dans la carte à l'aide de nœuds. Un nœud sera défini par un score, une position et un nœud parent. Dans l'implémentation de l'algorithme, l'estimation du score et son utilisation sont laissés à la liberté de la personne qui l'implémente. Le score sera ici une map, comprenant la distance au point d'arrivé (« end », int) et la somme des 'end ' précédents (« sum », int). L'algorithme contient 2 listes (vector dans notre code). La liste fermée contient les noeuds qui ont été testé et seront parcourus dans le chemin. La liste ouverte est la liste qui contient les noeuds qui sont à étudier. A chaque itération de l'algorithme nous procédons aux étapes suivantes :

- Selection du noeud ayant la plus faible valeur de « end » (le noeud le plus proche de la fin) dans la liste ouverte

- Pour chaque point situé autour de ce noeud :

- Si le point n'est un mur, et qu'il n'est pas déjà situé dans la liste fermée :

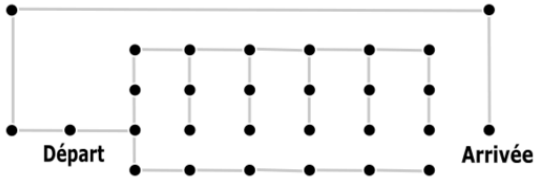
- S'il est dans la liste ouverte et que son score « sum » est meilleur, on remplace le noeud de la liste ouverte pas le noeud étudié

- Sinon, on ajoute le noeud à la liste ouverte, de noeud parent le noeud

L'opération est à réitérer tant que la liste ouverte n'est pas vide (aucun chemin ne mène au point d'arrivé), ou lorsque le point d'arrivé est dans la liste ouverte.

Ainsi, en remontant tous les noeuds de la liste fermée par chaque parent, on obtient le chemin jusqu'au point d'arrivé.

L'algorithme n'est pas efficace si les noeuds les plus proches de l'arrivé ne sont pas connectés à l'arrivé :



5.4 Conception Logicielle

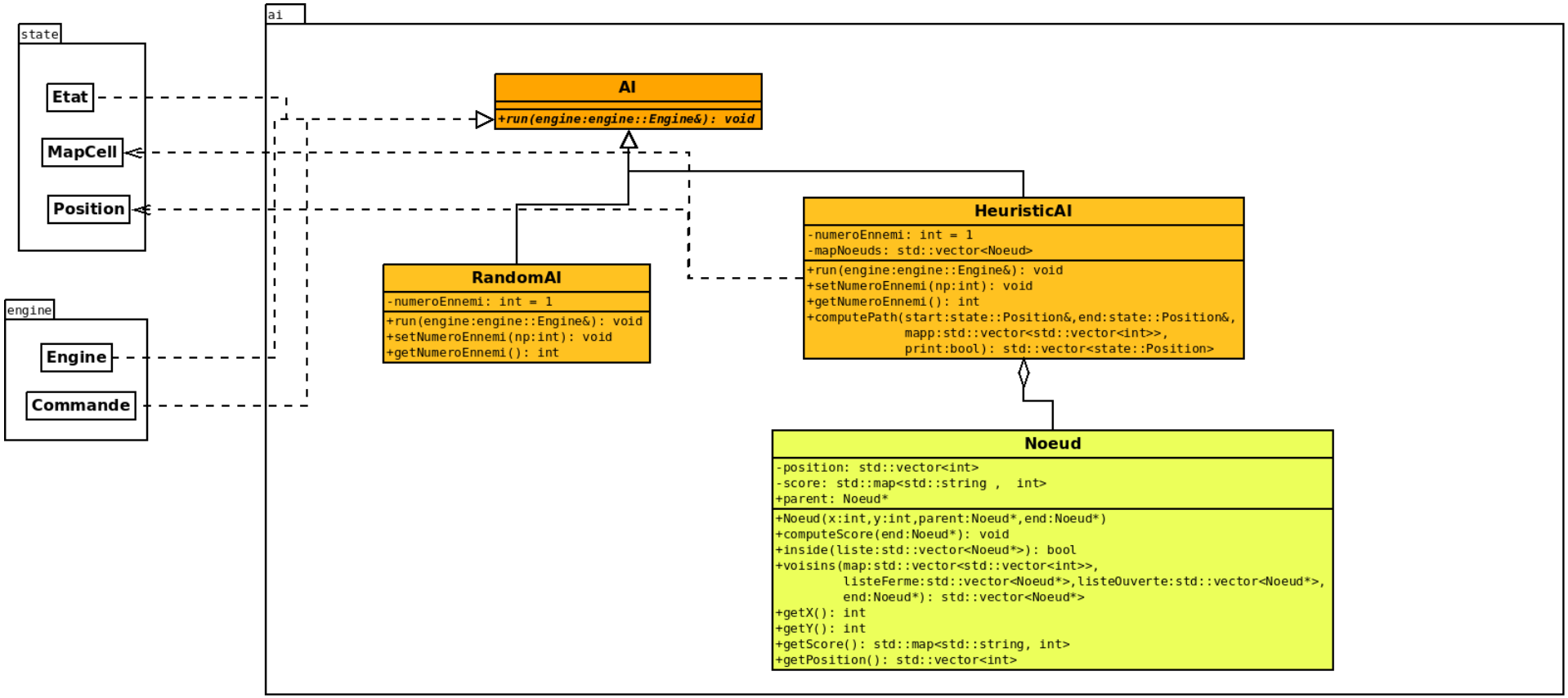


Figure 7 - Diagramme de l'IA Heuristique

6- Modularisation

6.1 Client

L'objectif est ici de séparer notre jeu en différents modules sur différents threads afin de facilité la conception Client/Serveur et d'optimiser l'utilisation des ressources. Du côté client, nous avons choisi de faire l'affichage du rendu et l'écoute/envoi de commandes.

Pour se faire nous disposons d'une classe Client qui contient les informations provenant du Moteur de jeu ainsi que du rendu. Nous avons aussi une classe KeyListener chargé de transmettre les commandes utilisateurs vers le Moteur de jeu.

6.2 Conception Logicielle

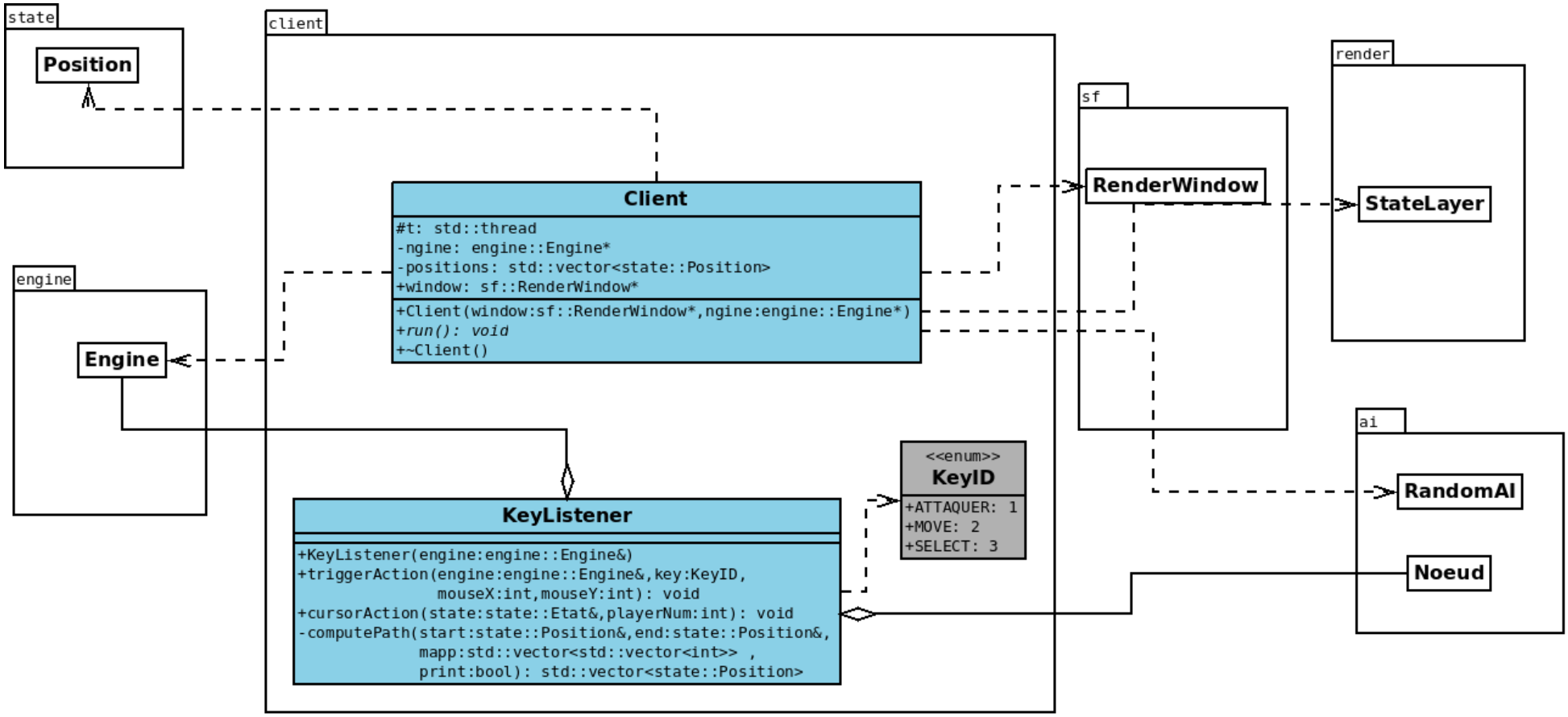


Figure 8- Diagramme du module client