Projet ZCOM

Simon GRANGIER – Valentin HORTANED

Table des matières

1 Objectif

1.1 Présentation générale

L'objectif est de vaincre les unités adverses en faisant progresser ses unités sur une carte quadrillée et en les faisant évoluer à chaque unité adverse détruite.

1.2 Règles du jeu

Le joueur commence une partie en arrivant sur une carte quadrillée, avec des *zones d'obstacles*, vue d'en haut avec un nombre d'unité équivalent à son adversaire. A chaque tour, une nouvelle unité apparaît dans la *zone d'apparition* puis chaque unité peut être déplacée et attaquer. Une unité possède un nombre de points de vie, un nombre de cases de déplacement et un type d'arme faisant plus ou moins de dégâts. Le type et la puissance d'une arme change lorsqu'une unité détruit une unité adverse. Le but est de détruire la(les) tour(s) de l'adversaire représentant sa vie.

1.3 Conception Logiciel

Cmake, SFML, DIA, Boost, gcc, Microhttpd, JSON

2 Description et conception des états

2.1 Description des états

Un état du jeu est formée par un ensemble d'éléments fixes (le terrain) et un ensemble d'éléments mobiles (les unités). Tous les éléments possèdent les propriétés suivantes :

— Coordonnées (x,y) dans la grille

— Identifiant de type d'élément : ce nombre indique la nature de élément (ie classe)

2.1.1 État éléments fixes

Le terrain est formé par une grille d'éléments nommé « cases ». La taille de cette grille est fixée au démarrage du niveau. Les types de cases sont :

Cases « Obstacle ». Les cases « Obstacle » sont des éléments infranchissables pour les éléments mobiles et bloquent leur champ de tir. On considère les types de cases « Obstacle » suivants :

— Les espaces « montagne », qui permettent de se mettre à couvert

— Les espaces « rivière », par dessus lesquels il est possible de tirer

— Les « Objectif », qui ont une barre de vie. La destruction d'une de ces cases entraîne la victoire de l'équipe adverse.

Cases « Sol ». Les cases « Sol » sont les éléments franchissables par les éléments mobiles. On considère les types de cases « Sol » suivants :

— Les espaces « plaine » ou « pont »

— Les espaces « Apparition », sur lesquels apparaissent les nouvelles unités.

2.1.2 État éléments mobiles

Les éléments mobiles (unités) possède une direction (gauche, droite, haut ou bas), une position, des points de déplacements, des points de vie, une attaque et un pouvoir unique comparé aux autres unités. À chacun de leur tour, les joueurs pourront déplacer une par une leurs unités, dans l'ordre souhaité. Chaque tour, une unité peut être déplacé d'un nombre de cases égal à ses points de déplacements et peut choisir entre attaquer, utiliser son pouvoir ou terminer son action.

— Status « immobile » : animation d'attente jusqu'à que le status change.

— Status « en mouvement» : après sélection d'une unité et de la case où l'on souhaite et où il est possible de la déplacer, une animation de mouvement et un déplacement vers la case est effectué.

— Status « attaque » : après sélection d'une cible, une animation d'attaque se lance selon le type d'arme.

— Status « pouvoir » : lorsqu'une unité détruit une unité adverse, elle évolue, changeant alors ses ses propriétés, son attaque et son pouvoir.

— Status « apparition » : chaque tour, une nouvelle unité apparaît dans la « zone d'apparition ».

— Status « mort » : cas où l'unité n'a plus de vie, elle quitte le terrain.

2.1.3 Etat général

A l'ensemble des éléments statiques et mobiles, nous rajoutons les propriétés suivantes :

— Tour : définie le joueur pouvant effectuer des actions avec ses unités.

— Temps écoulé: le nombre d'époque depuis le début du tour d'un joueur.

2.2 Conception logiciel

Le diagramme des classes pour les états est présenté en Figure 5, dont nous pouvons mettre en évidence les groupes de classes suivants :

Classes GameObject Toute la hiérarchie des classes filles de GameObject (en orange) permettent de représenter les différentes catégories et types de game object. C'est un exemple très classique d'utilisation du Polymorphisme. Etant donné qu'il n'y a pas de solution standard efficace[1] en C++ pour faire de l'introspection, nous avons opté pour des méthodes comme isStatic() qui indiquent la classe d'un objet. Ainsi, une fois la classe identifié, il suffit de faire un simple `static_cast<Classe>(gameObject)`.

Conteneurs d'élément. Viennent ensuite les classes Player, World et GameState qui permettent de contenir des ensembles de game object. GameState contiendra tous les objets permettant de récupérer une partie en cours. La classe World est le conteneur principal, à partir duquel on peut accéder à toutes les données de l'état initial d'une partie, permettant ainsi de relancer une partie. Et enfin, la classe Player les données et objets des joueurs qui s'affrontent.

Note 1 : Actuellement, la classe World est génère la carte dans le constructeur à partir d'un string inscrit dans le code. A terme, celui-ci appellera une fonction loadMap qui chargera les informations depuis un fichier.

Note 2 : Les classes GameState et Character ont des méthodes non implémentés. Dû au prototype généré par dia2code, nous n'avons pas réussi à attribuer une valeur aux attributs de type [Classe]* depuis les arguments de type const [Classe]&.

1 Il existe « Run-Time Type Information (RTTI)", cependant cela est peu portable et souffre de sérieux problèmes de performance.

2.3 Conception logiciel : extension pour le rendu

2.4 Conception logiciel : extension pour le moteur de jeu

2.5 Ressources

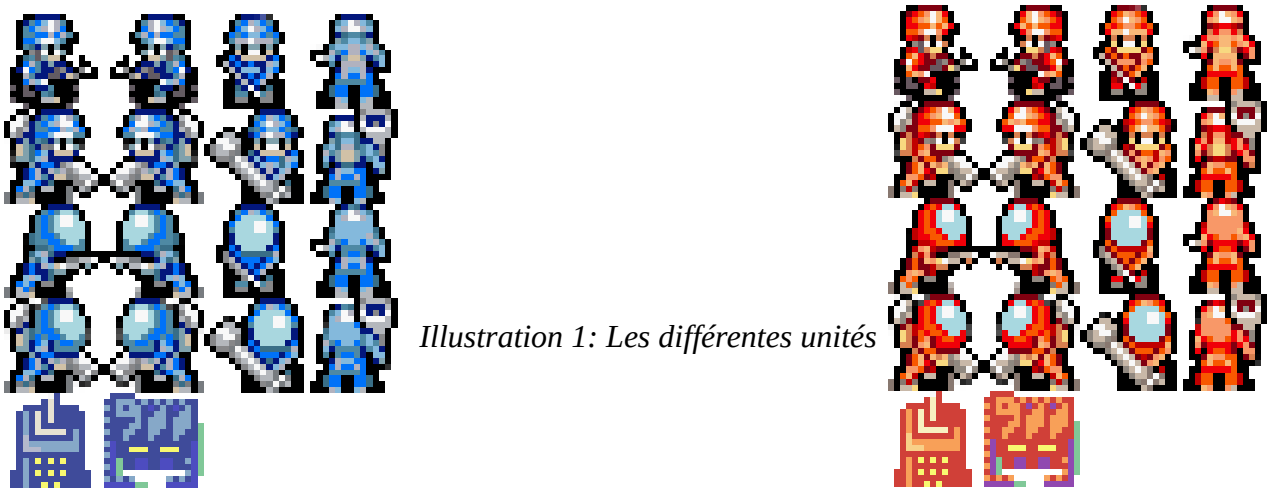


Illustration 1: Les différentes unités

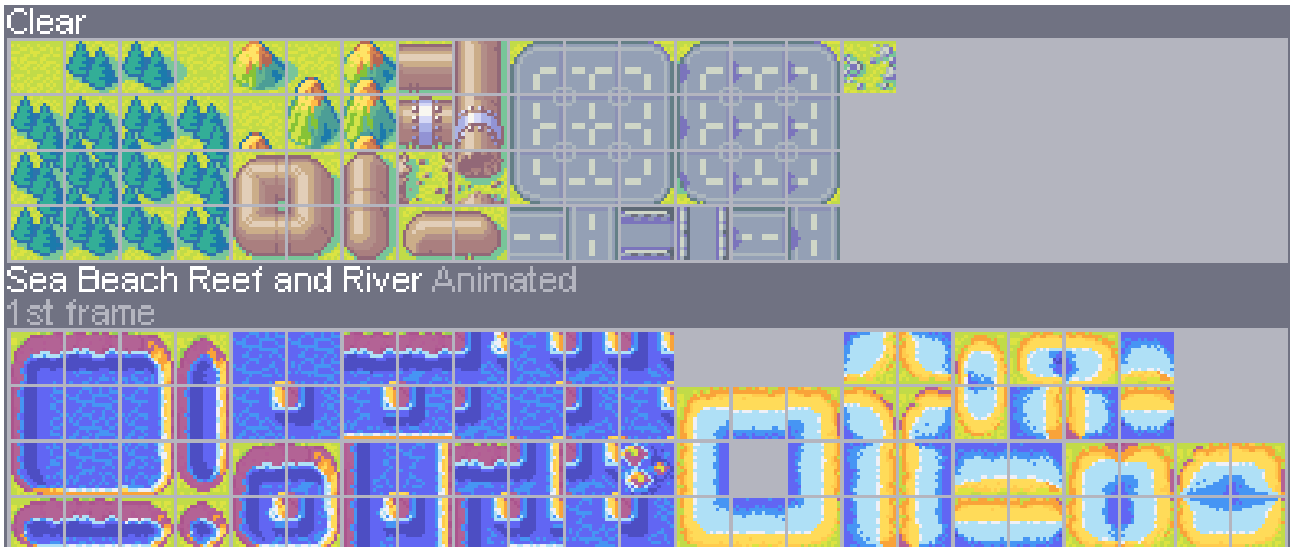


Illustration 2: Éléments du décor

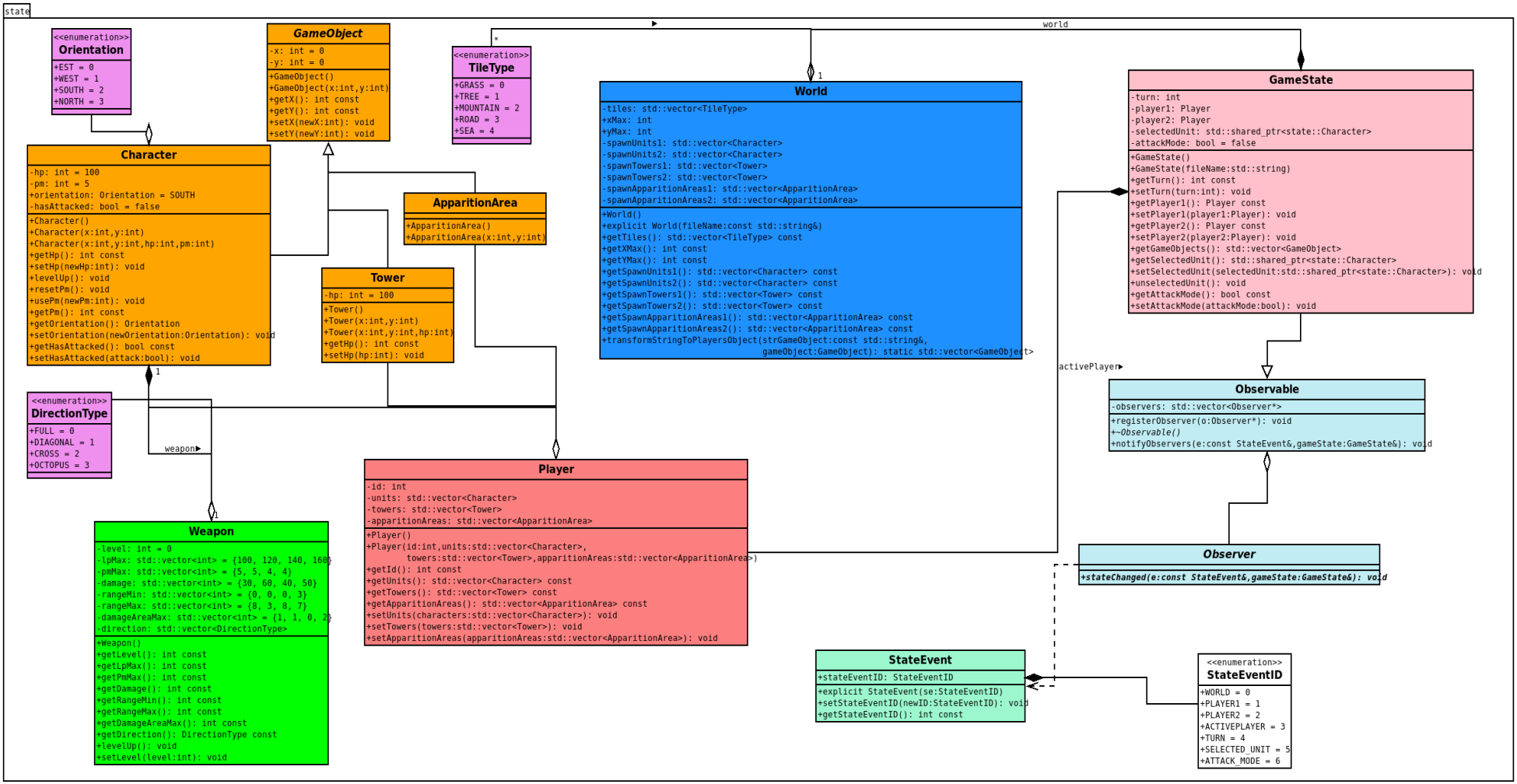


Illustration 3: Diagramme des classes d'état

3 Rendu : Stratégie et Conception

Présentez ici la stratégie générale que vous comptez suivre pour rendre un état. Cela doit tenir compte des problématiques de synchronisation entre les changements d'états et la vitesse d'affichage à l'écran. Puis, lorsque vous serez rendu à la partie client/serveur, expliquez comment vous aller gérer les problèmes liés à la latence. Après cette description, présentez la conception logicielle. Pour celle-ci, il est fortement recommandé de former une première partie indépendante de toute librairie graphique, puis de présenter d'autres parties qui l'implémente pour une librairie particulière. Enfin, toutes les classes de la première partie doivent avoir pour unique dépendance les classes d'état de la section précédente.

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 utiliser 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, la carte avec les éléments décoratifs (mur, sol, escalier) et les éléments venant à changer à savoir les gameObject (unités, tours et zones d'apparitions) qui se superposent sur la couche précédente. On transmet l'état du jeu ainsi que les textures de toute la carte avec leurs coordonnées et la texture des unités avec leurs coordonnées et leur orientation à afficher.
La carte étant très grande, nous l'avons entièrement chargé dans la mémoire pour l'affichage.
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 unités, seul le rendu des unités 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.
Nous avons rajouté sur la scène finale, certains attributs des unités (point de vie actuelle et id des unités) en bas de la fenêtre.

3.2 Conception logiciel

Pour afficher un état on crée une scène qui génère un ensemble de pointeurs sur des objets graphiques de types LayerRender pour la carte. Pour les unités on instancie un Sprite par unité. Ce Sprite contient à la fois les coordonnées de l'unité sur la fenêtre et sa position sur le Tileset utilisé. Puis, on utilise la méthode Scene : :update() pour déclencher l'ouverture de la fenêtre et l'affichage de l'état. Il est à noter que l'état contient d'ores et déjà toutes les informations nécessaires à l'affichage de la carte. En effet, à la création de l'état celui-ci parse un fichier texte (ici un string dans main.cpp)contenant toutes les informations nécessaires à l'affichage de la carte.
classe Scene est un des Observer de GameState. Ainsi, lorsque l'état subit un changement, Scene est notifiée et met à jour le rendu en fonction du type d'évènement qui lui est transmis :
— Si un PLAYER_EVENT (correspondant à changement d'état pour l'un des joueurs) la méthode update-Players() responsable de la mise à jour du rendu des unités est appelée.
— Si un WORLD_EVENT (correspondant changement d'état lié à l'environnement ou à la partie) la méthode updateMap().
Par ailleurs, les points de vie de chacunes des unités sont, pour le moment, affichés et mis à jour en temps réel à l'intérieur de la méthode draw() de la classe Scene.
Chaque objet LayerRender parcourt l'ensemble des tuiles d'un étage de la carte, détecte la position de la tuile sur la ressource graphique (i.e : l'image du tileset) et calcule sa position sur la fenêtre. Pour optimiser les performances seule l'image du tileset est chargée dans une Texture, l'objet

LayerRender conserve seule ment la position de chaque tuile de la carte sur cette texture. Les unités sont chacun rendus à l'intérieur d'un Sprite. Lorsqu'un mouvement de l'un des unités est détecté le la classe scène met à jour l'affichage de l'ensemble des unités.

3.3 Conception logiciel : extension pour les animations

Les animations concernant notre seront :

- Animation de déplacement
- Animation d'explosion/dégâts

https://github.com/SFML/SFML/wiki/Source:-AnimatedSprite

3.4 Ressources

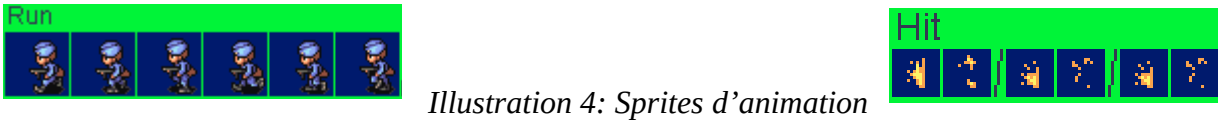


Illustration 4: Sprites d'animation

3.5 Exemple de rendu

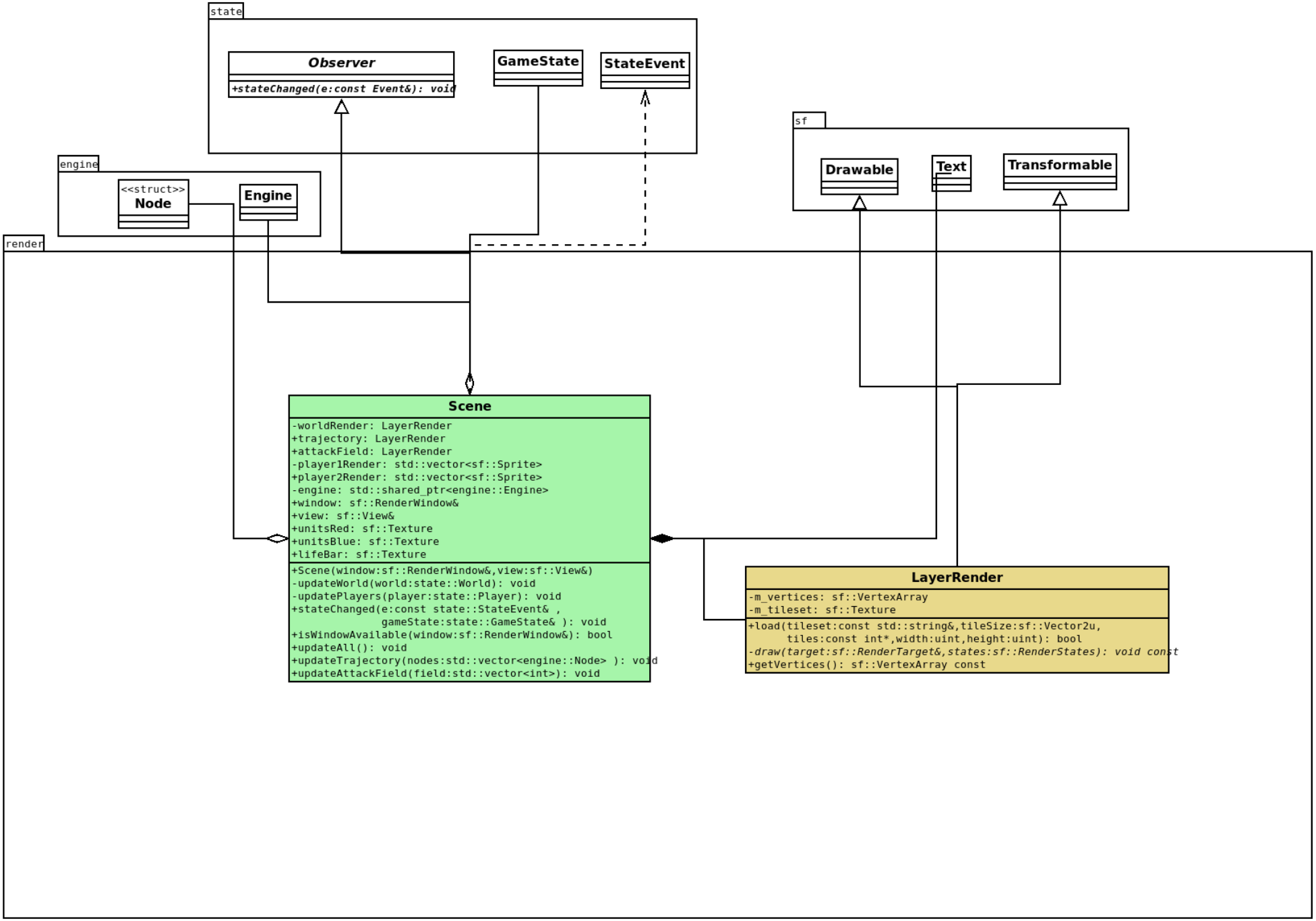


Illustration : Diagramme de classes pour le rendu

4 Règles de changement d'états et moteur de jeu

Dans cette section, il faut présenter les événements qui peuvent faire passer d'un état à un autre. Il faut également décrire les aspects lié au temps, comme la chronologie des événements et les aspects de synchronisation. Une fois ceci présenté, on propose une conception logiciel pour pouvoir mettre en œuvre ces règles, autrement dit le moteur de jeu.

4.1 Horloge globale

Les changements d'états sont liés aux commandes exécutées par le moteur. Les commandes extérieures
sont :

— Commande de déplacement
— Commande d'attaque
— Commande passer son tour

Un personnage ne peut se déplacer que d'une case par une case (sur des terrains praticables et inoccupés) . Il possède des points de mouvements qui correspondent au nombre de déplacement maximum qu'il lui est possible de faire. Lorsque tous ses points de mouvement ont été utilisés, il lui est impossible de continuer son déplacement.

Un personnage ne peut attaquer un autre personnage que lorsque celui ci se trouve dans son champ d'attaque et appartient au camp adverse (les attaques alliées ne sont pas autorisées).

Lorsqu'un personnage termine ses actions, son statut devient ATTENTE. Cela signifie qu'il ne pourra plus effectuer d'actions avant le prochain tour du joueur.

L'action « Terminer son tour d'action » doit obligatoirement être effectuée par chaque personnage encore actif à chaque tour du joueur. Il existe 4 enchaînements d'actions possibles :

- Attaquer directement (lorsque cela est possible) ce qui termine automatiquement le tour d'un personnage
- Effectuer un ou plusieurs déplacements , attaquer puis terminer son tour automatiquement
- Effectuer un ou plusieurs déplacements puis terminer son tour manuellement
- Terminer son tour directement sans avoir effectuer aucune autre action

Le tour de jeu d'un joueur est terminé lorsque tous ses personnages sont en ATTENTE. C'est alors le tour du joueur adverse.

Si un personnage perd tous ses points de vie, son statut évolue et prend la valeur MORT.

Si tous les personnages ou tours d'un joueur meurent, la partie est terminée à la fin du tour adverse et le joueur adverse gagne.

Chaque action effectuée modifie l'état. Le choix du personnage sélectionné et des actions effectuées est défini par des commandes.

Les commandes du joueur sont provoquées par une pression sur une touche.

- Sélectionner un personnage : « Clique gauche » (lorsque le curseur encadre le personnage).
  - Déplacer un personnage : « Mouvement souris + clique gauche » (le personnage doit avoir été sélectionné au préalable).
  - Mode Attaque : « Double Clique gauche » (lorsque le curseur encadre le personnage avec une projection en bleu de sa portée d'attaque). Puis « Mouvement souris + clique gauche » pour confirmer l'attaque (le personnage doit avoir été sélectionné au préalable et en mode attaque).
    - Annuler une selection : Appuyer sur « Clique droit » ou « Clique gauche sur une autre unité » ou « Clique gauche sur une case hors de portée de

déplacement/attaque » .
- Terminer le tour d’actions du joueur : « T».

4.2 Changements extérieurs

Les changements extérieurs sont provoqués par des commandes extérieurs, comme la pression sur une touche ou un ordre provenant du réseau :
— Commandes principales : « Passer son tour » : on exécute la commande
— Commandes « Sélection unité » : effectué par un clique gauche avec la souris

4.3 Changements autonomes

Les changements autonomes interviennent à la fin de chaque changement d’état lié à une commande extérieure et s’exécutent dans l’ordre suivant :
1. Si le joueur n’a plus d’unités ou de tours, « game over »
3. On met à jour les statistiques (point de vie) du joueur en fonction des règles d’attaque
4. On applique les règles de déplacement du joueur.

4.4 Conception logiciel

On utilise ici le pattern Command : un ensemble d’utilisateur peut ajouter des "commandes" à l’objet Engine puis lui demander de les exécuter à l’aide de la méthode runCommands().
Les commandes hérite de la classe abstraite Command contient une méthode execute() virtual pure qui est appelée par le moteur lorsqu’un utilisateur appelle la méthode runCommands(). Cette méthode applique en fait directement des changements à l’objet GameState conservé par la classe Engine.
On a implémenté, pour le moment, 3 types de commande :
— MoveCommand qui correspond à un changement de position d’une des unités.
— AttackCommand qui correspond à l’attaque d’une unité.
— NewTurnCommand qui est déclenché lorsqu’un joueur passe son tour.
Les trois commandes MoveCommand, AttackCommand et NewTurnCommand sont déclenchés sur un évènement de lecture de clavier ou souris, plus précisement sur un évènement d’appui sur une touche de clavier pour plus de fluidité lors des déplacements.

La commande MoveCommand met à jour la position et l’orientation d’une unité sur la map.
La commande AttackCommand agit sur l’état des unités et sur le rendu de la scène en gardant à jour les points de vie actuels représentés par une barre de vie au-dessus des unités.
La commande MoveCommand remet à jour le nombre de points de mouvements des unités et leur possibilité d’attaquer.

4.5 Conception logiciel : extension pour l'IA

4.6 Conception logiciel : extension pour la parallélisation

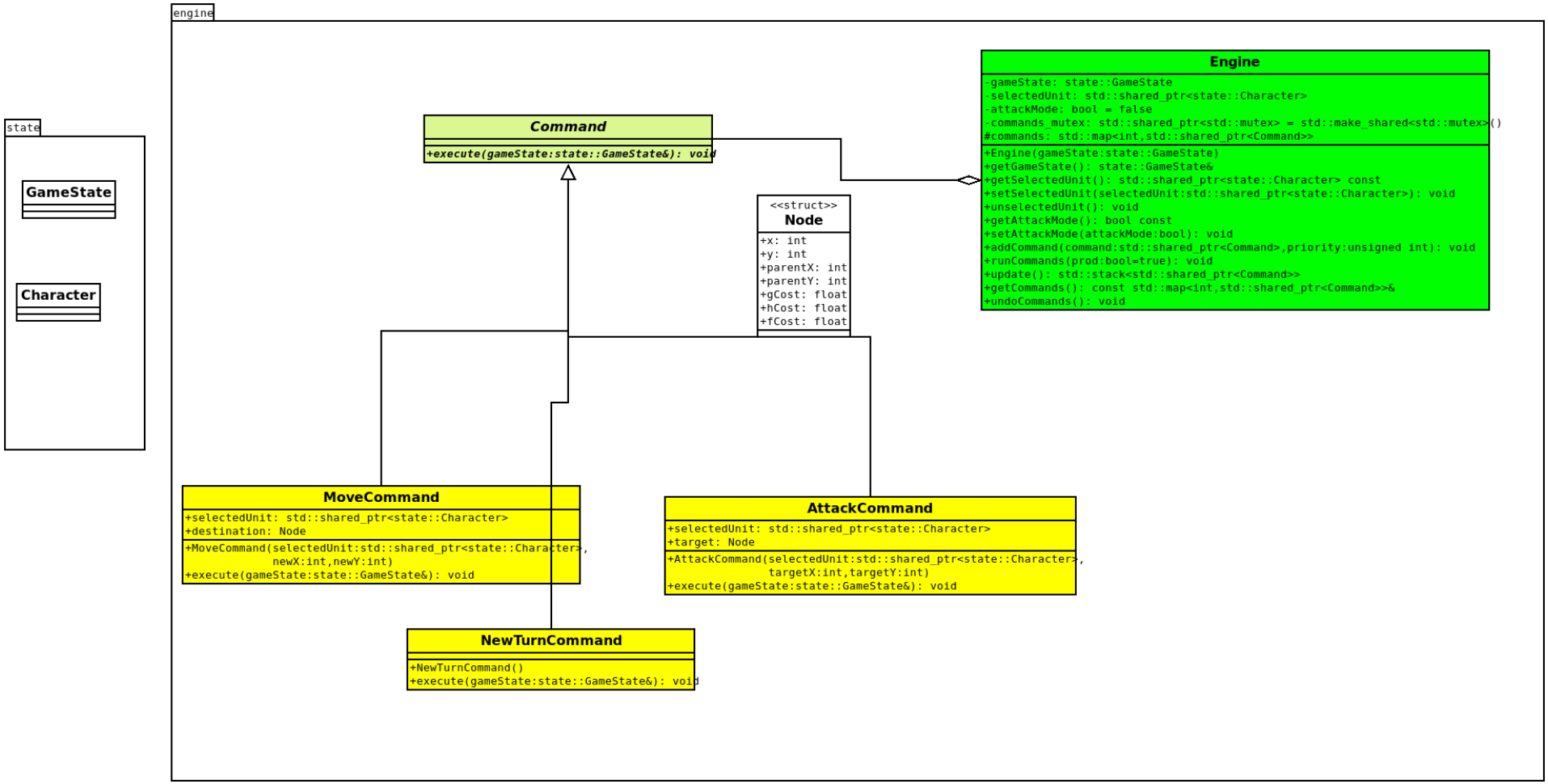


Illustration 5: Diagrammes des classes pour le moteur de jeu

5 Intelligence Artificielle

Cette section est dédiée aux stratégies et outils développés pour créer un joueur artificiel. Ce robot doit utiliser les mêmes commandes qu'un joueur humain, ie utiliser les mêmes actions/ordres que ceux produit par le clavier ou la souris. Le robot ne doit pas avoir accès à plus information qu'un joueur humain. Comme pour les autres sections, commencez par présenter la stratégie, puis la conception logicielle.

5.1 Stratégies

5.1.1 Intelligence minimale

L' IA contrôle une liste de personnages qu'elle va sélectionner les uns après les autres lors de son tour de jeu. Une fois un personnage sélectionné, un entier correspondant à une action précise est choisi au hasard parmi 3 : 0 (se déplacer), 1 (attaquer), 2 (terminer son tour d'actions).

Si « se déplacer » est choisi et que le personnage sélectionné possède encore des points de mouvement, une destination va être choisie aléatoirement parmi une liste de destinations possibles. Si la liste n'est pas vide, le personnage se déplacera. Sinon il ne pourra pas se déplacer. Dans les deux cas, il restera SÉLECTIONNÉ et l'IA pourra donc continuer à le manipuler (une nouvelle action aléatoire sera tentée après celle-ci).

Si « attaquer » est choisi, une cible va être déterminée aléatoirement parmi une liste de cible possibles (qui diffèrent en fonction de la position du personnage et de son champ d'attaque). Si la liste n'est pas vide, le personnage va attaquer puis terminer directement son tour, comme défini dans les règles du jeu. Il passera donc en ATTENTE ou MORT (s'il est décédé des suites de son attaque), l'IA ne pourra plus le manipuler et sélectionnera le personnage suivant. Si la liste d'attaque est vide, l'attaque n'aura pas lieu, le personnage restera SÉLECTIONNÉ et l'IA pourra donc continuer à le manipuler (une nouvelle action aléatoire sera tentée après celle-ci).

Si « terminer son tour d'actions » est choisi, le personnage va terminer son tour d'actions directement. Il passera donc en ATTENTE. L'IA ne pourra plus manipuler ce personnage et sélectionnera le personnage suivant.

Le tour de l'IA est terminé lorsque tous ses personnages ont été sélectionnés et sont en ATTENTE ou MORT. L'IA pourra de nouveau sélectionner ses personnages (à condition qu'ils ne soient pas morts) au tour suivant et ce schéma se répétera jusqu'à la fin de la partie.

5.1.2 Intelligence basée sur des heuristiques

L'IA va définir un score sur chaque mouvement et attaque possible de chaque unité, et effectuer la meilleur action :
D'abord -> si une unité peut faire des dégâts à un ennemi, le score est égal aux dégâts.
Par exemple: si vous pouvez faire -30 à un ennemi, le score de l'action est 30. Bonus de +10 si l'ennemi peut être tuer ou à moins de vie qu'un autre ennemi qui peut également être touché,

Deuxièmement -> si une unité ne peut pas faire de dégât à un ennemi, le score pour se déplacer est alors proportionnel à la distance de l'ennemi le plus proche. Le score est plus élevé si vous pouvez être proche d'un ennemi.
Par exemple: l'ennemi est 9 cases devant vous, si vous vous déplacez de 5 cases derrière vous, le score est de -5, si vous avancez vers l'enemi, il est de 5.
Mais si l'unité a attaqué, son score de mouvement est encore plus grand s'il peut trouver un mouvement où il ne peut pas être attaqué.

5.1.3 Intelligence basée sur les arbres de recherche

Nous proposons en dernier lieu une intelligence avancée basée sur l'algorithme minmax à deux joueurs. Dans notre implémentation, l'IA avancée va explorer toutes les issues possibles et va tenter de maximiser ses chances de victoire face au joueur. Pour cela, nous avons implémenté les règles suivantes :
1. le fait de se déplacer quand il ne voit pas d'ennemi autour, caractérisé par la distance entre l'IA et le Joueur.
2. le fait d'attaquer l'ennemi lorsqu'il est à portée.
3. le fait de se mettre hors de portéé si possible.
Ainsi à chaque tour, l'IA avancée va choisir le coup optimal parmi l'ensemble des actions qu'il peut effectuer et qui va maximiser ses chances de gagner grâce à un score qui mettra en avant ces actions tour à tour. Pour cela, l'algorithme va parcourir l'ensemble des issues possibles à chaque tour. Pour effectuer cette tâche, on a choisi de dupliquer la partie de l'état nécessaire au bon fonctionnement de l'algorithme (état des tours et des unités) à chaque noeud de l'arbre de recherche. Pour ce qui est du déplacement sur la carte, nous avons gardé le même algorithme que celui de l'heuristique, à savoir le A *, afin que l'IA puisse se diriger et s'orienter vers le joueur sur la carte de façon optimum.

5.2 Conception logiciel

Les classes filles de la classe AI implémente 3 stratégie d'IA suivi par l'ennemi :

1. RandomAI : Intelligence Artificielle aléatoire
2. HeuristicAI : Intelligence Artificielle Heuristique
3. DeepAI : Intelligence Artificielle sur plusieurs niveau

Algorithme AStar : Cet algorithme permet de trouver le chemin le plus cours entre deux cases de notre carte. Déjà implémenté pour les déplacements, cet algorithme est utilisé par l'IA pour déterminer sa distance aux ennemis et aux alliés.

Algorithme Ray Tracing : Cet algorithme permet de vérifier si la vue entre deux points est obstruée. Ce code a d'abord été utilisé pour altérer la portée des unités selon le terrain. L'IA en a une utilisation inverse afin de déterminer si ses unités sont à portée des unités du joueur.

DeepAI : C'est une IA basée sur la résolution de problème à état finis. On utilise l'algorithme de parcours d'arbres minmax. Afin d'achever cela, un système de retour en arrière qui permet aussi d'enregistrer et rejouer une partie est implémenté.

Stratégie : Les critères de "jugement" de l'IA sont :
— Sa vie qu'elle veut maximiser
— La vie de l'ennemi qu'elle veut minimiser
— Sa distance à l'ennemi qu'elle veut minimiser

DeepAI : Malheureusement, une partie peut rassembler jusqu'à quelques dizaines d'unités et l'environnement étant relativement grand il est impossible pour l'IA de parcourir l'arbre de recherche en profondeur à chaque coup. En effet chaque noeud de l'arbre possède d'innombrable branches et la carte, bien que possédant des murs et des obstacles, contient des centaines de cases. Ainsi, on comprend qu'il est impossible de parcourir l'ensemble de l'arbre dans un temps acceptable. Dans la pratique, l'algorithme parcourt au maximum 2 niveaux de l'arbre. Malheureusement, cela ne semble pas suffisant pour obtenir une intelligence plus performante que l'IA Heuristique. Pour améliorer la performance de l'algorithme nous pourrions mettre en place des

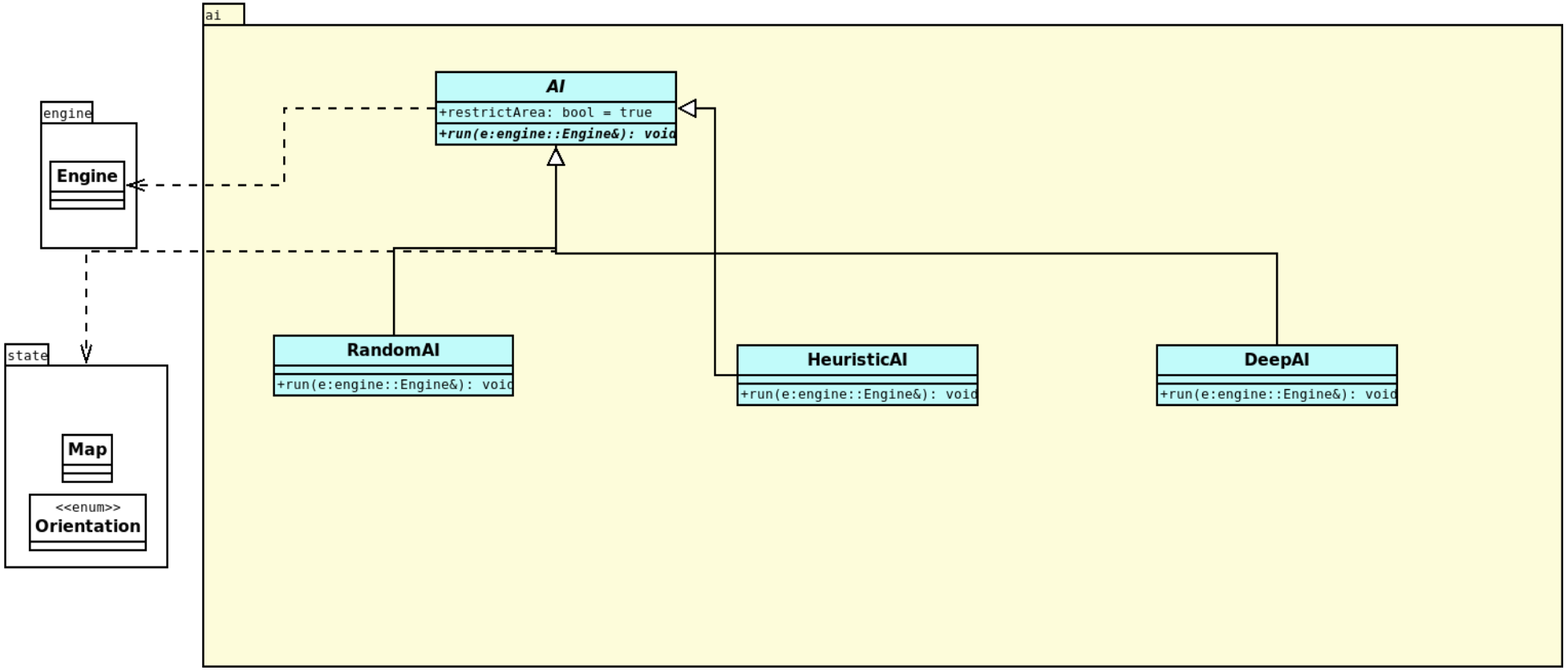
conditions pour le parcours des noeuds de l’arbre de recherche pour ainsi éviter le parcours de noeuds inutiles (algorithme alpha bêta).

5.3 Conception logiciel : extension pour l'IA composée

5.4 Conception logiciel : extension pour IA avancée

5.5 Conception logiciel : extension pour la parallélisation

Figure 8 – Diagramme des classes d'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 un 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

On veut ici lancer le moteur et le rendu dans deux threads différents. Le thread principal sera celui du rendu (contrainte matérielle) et le secondaire celui du moteur. Deux types de données transitent entre le moteur et le rendu :
— Les commandes, qui transistent du rendu vers le moteur
— Les évènements notifiant le rendu d'un changement d'état

6.1.1 Répartition sur différents threads

Premièrement, les commandes peuvent arriver à tout moment et l'ajout d'une commande à la map contenant les commandes lors d'une mise à jour de l'état de jeu pourrait engendrer des conflits. Pour palier à ce problème, nous avons utilisé un mutex qui bloque l'exécution de handleInputs pendant toute la durée du tour de l'adversaire. Cette méthode n'est pas viable à long terme car cela empêche des actions comme le redimensionnement de la fenêtre.

Lors d'une mise à jour, l'état notifie le rendu par le biais d'events. Pour le moment, le traitement de ces notifications bloque le rendu et inversement, le rendu du jeux bloque le traitement des notifications. Ce mécanisme de verrou est implémenté à l'aide d'un mutex et est coûteux en termes de performances. Pour remédier à ce problème on pourra à l'avenir utiliser une liste stockant l'ensemble des notifications ainsi qu'un signal permettant de signaler la présence d'un évènement qui doit être traité au rendu. Ainsi, lorsque le rendu a fini sa dernière tâche et que le signal est actif, il vide la liste, traite les notifications et réarme le signal. Ce mécanisme devrait permettre d'améliorer notablement les performances.

6.1.2 Répartition sur différentes machines

Nous avons crée une base de données d'utilisateurs sur notre serveur. Le serveur reçoit les commandes à travers une API REST et envoie à son tour les commandes et les notifications.

6.2 Conception logiciel

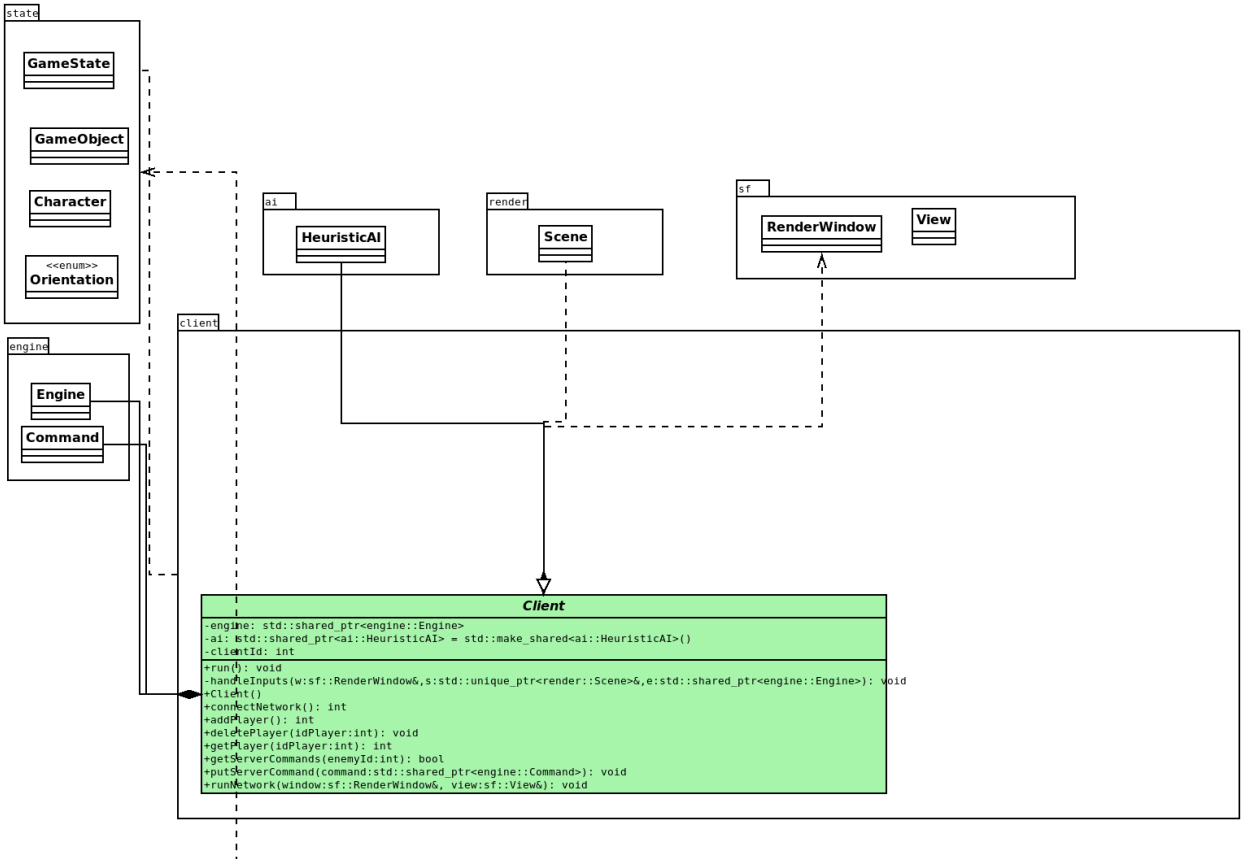
Le diagramme des classes pour la modularisation est présenté en Figure 7. Client La classe Client contient le moteur de jeu, les intelligences artificielles et les rendus. Elle utilise ces différents éléments dans deux threads distincts. Dans le thread principal gérant l'affichage on procède selon l'ordre suivant :
1. La méthode handleInputs gère l'ajout des commandes au moteur lorsqu'un utilisateur appuie sur un touche.
2. On affiche le rendu à l'aide de la méthode Scene::draw. Dans le thread secondaire on gère principalement les IA et le moteur. En particulier, on utilise l'IA pour ajouter les commandes automatisé au moteur avant d'éxecuter la méthode Engine::runCommands qui applique les changements d'état. La méthode Client::run permet d'exécuter le client. Les services REST implémentés dans le serveur héritent tous de la classe abstraite AbstractService. La classe

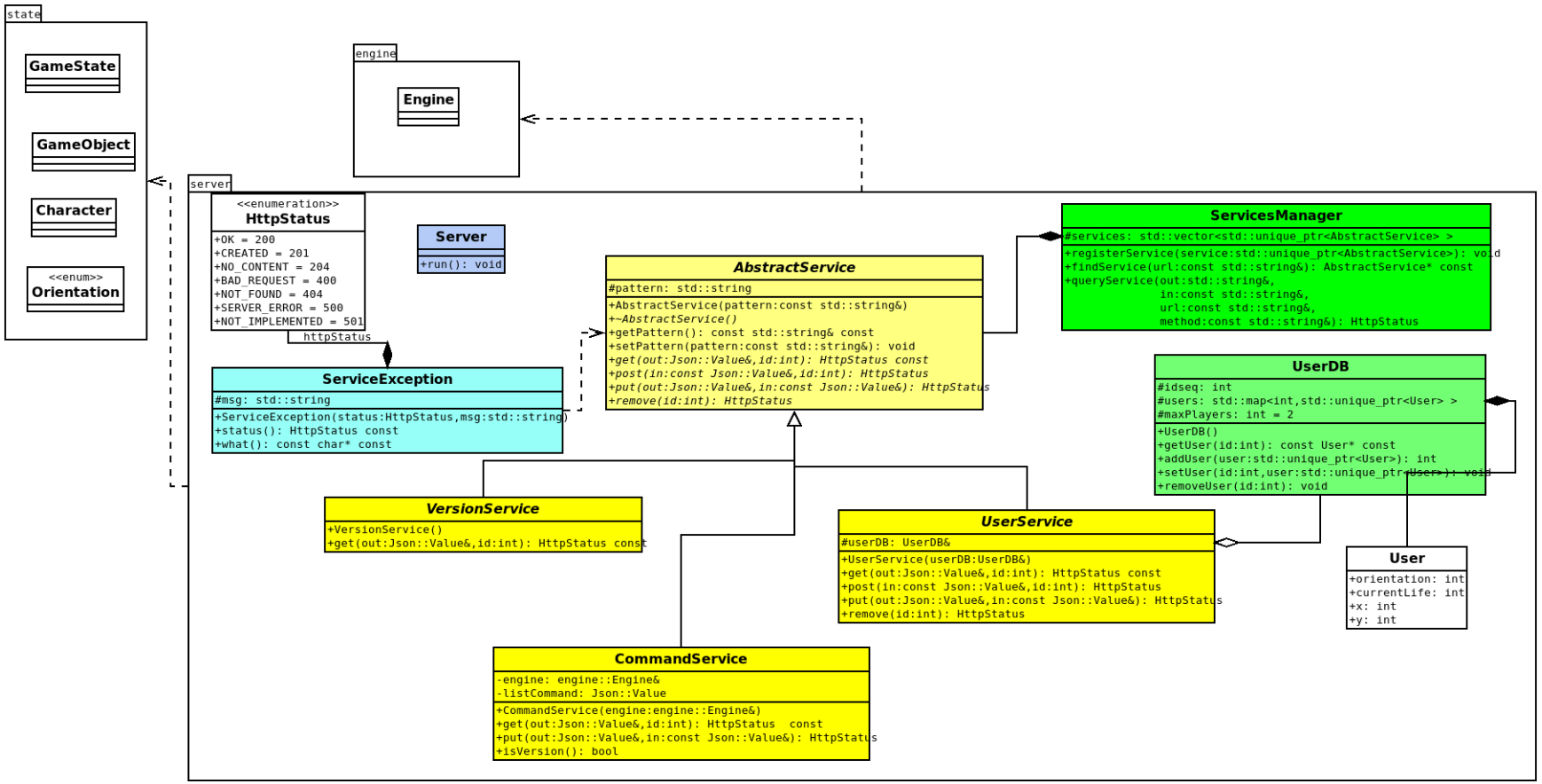
ServiceManager permet de gérer l’ensemble des services disponibles sur le server :
— elle permet d’ajouter de nouveaux services REST grace à la méthode registerService
— d’accéder à la méthode GET du service spécifié (i.e. en fonction de son url) en paramètre de la méthode findService
— d’accéder à une méthode d’un des services en fonction des paramètres du requête reçue par le serveur grâce à la méthode queryService
Pour l’instant nous utilisons deux services :
— Un service UserService qui permet de gérer les utilisateurs du jeu
— Un service VersionService qui renvoie la version du serveur.

6.3 Conception logiciel : extension réseau

6.4 Conception logiciel : client Android

Illustration 6: Diagrammes de classes pour la modularisation




state
GameState
GameObject
Character
<<enum>>
Orientation
engine
Engine
server
<<enumeration>>
HttpStatus
+OK = 200
+CREATED = 201
+NO_CONTENT = 204
+BAD_REQUEST = 400
+NOT_FOUND = 404
+SERVER_ERROR = 500
+NOT_IMPLEMENTED = 501
httpStatus
Server
+run(): void
AbstractService
#pattern: std::string
+AbstractService(pattern:const std::string&)
+~AbstractService()
+getPattern(): const std::string& const
+setPattern(pattern:const std::string&): void
+get(out:Json::Value&,id:int): HttpStatus const
+post(in:const Json::Value&,id:int): HttpStatus
+put(out:Json::Value&,in:const Json::Value&): HttpStatus
+remove(id:int): HttpStatus
ServicesManager
#services: std::vector<std::unique_ptr<AbstractService> >
+registerService(service:std::unique_ptr<AbstractService>): void
+findService(url:const std::string&): AbstractService* const
+queryService(out:std::string&,
in:const std::string&,
url:const std::string&,
method:const std::string&): HttpStatus
ServiceException
#msg: std::string
+ServiceException(status:HttpStatus,msg:std::string)
+status(): HttpStatus const
+what(): const char* const
UserDB
#idseq: int
#users: std::map<int,std::unique_ptr<User> >
#maxPlayers: int = 2
+UserDB()
+getUser(id:int): const User* const
+addUser(user:std::unique_ptr<User>): int
+setUser(id:int,user:std::unique_ptr<User>): void
+removeUser(id:int): void
VersionService
+VersionService()
+get(out:Json::Value&,id:int): HttpStatus const
UserService
#userDB: UserDB&
+UserService(userDB:UserDB&)
+get(out:Json::Value&,id:int): HttpStatus const
+post(in:const Json::Value&,id:int): HttpStatus
+put(out:Json::Value&,in:const Json::Value&): HttpStatus
+remove(id:int): HttpStatus
User
+orientation: int
+currentLife: int
+x: int
+y: int
CommandService
-engine: engine::Engine&
-listCommand: Json::Value
+CommandService(engine:engine::Engine&)
+get(out:Json::Value&,id:int): HttpStatus const
+put(out:Json::Value&,in:const Json::Value&): HttpStatus
+isVersion(): bool