Projet Advance Wars

Ahmed BEN HAMOUDA – Benjamin FRANÇOIS

FIGURE 1 – Exemple du jeu Advance Wars

Table des matières

1 Objectif....3
1.1 Archétype....3
1.2 Règles du jeu....3
1.3 Ressources....3
2 Description et conception des états....5
2.1 Description des états....5
2.1.1 Couche inférieure....5
2.1.2 Couche Intermédiaire....5
2.1.3 Couche Supérieure....5
2.1.4 État général....5
2.2 Conception Logiciel....6
3 Rendu : Stratégie et Conception....8
3.1 Stratégie de rendu d'un état....8
3.2 Conception logiciel....8
4 Règles de changement d'états et moteur de jeu....10
4.1 Changements extérieurs....10
4.2 Changements autonomes....10
4.3 Conception logiciel....10
5 Intelligence Artificielle....12
5.1 Stratégies....12
5.1.1 Intelligence aléatoire....12
5.1.2 Intelligence basée sur des heuristiques....12
5.1.3 Intelligence avancé....13
5.2 Conception logiciel....13

1 Objectif

1.1 Archétype

L'objectif de notre projet est de développer une version simplifiée du jeu Advance Wars. Afin de personnaliser notre projet, nous avons décidé de créer une variante des règles originales du jeu.

1.2 Règles du jeu

On dispose d'un terrain composé de deux bases avec plusieurs unités. Chaque joueur contrôle les unités de sa base. Contrairement au jeu original où le but est de capturer la base ennemie, l'objectif ici va être de capturer un drapeau situé dans la base ennemie pour la ramener dans sa propre base. Le vainqueur est celui qui aura ramené le drapeau ennemi dans sa propre base le premier.
Le jeu se déroule au tour par tour : lorsqu'un joueur attaque, il peut déplacer chacune de ses unités individuellement. Si une de ses unités rencontre une unité ennemie, le joueur a la possibilité de lancer une offensive. Enfin, pour éviter de perdre toutes ses unités au combat, chaque joueur possède une solde qui augmente à chaque tour. De cette manière, il peut acheter des unités supplémentaires. Il est à noter qu'il existe différents types d'unités, et que les unités plus puissantes coûtent plus cher que les unités de base.

1.3 Ressources

L'affichage du jeu repose principalement sur trois textures :

- une texture de 464 x 48 pixels correspondant aux tuiles du terrain, présentée dans la figure 2.
- une texture de 287 x 80 pixels correspondant aux tuiles des différentes unités, présentée dans la figure 3.
- une texture de 97 x 195 pixels correspondant aux tuiles des bâtiments, présentée dans la figure 4.

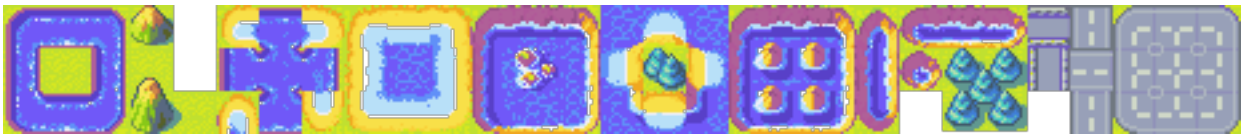

FIGURE 2 – Texture pour les tuiles du terrain

FIGURE 3 – Texture pour les tuiles des unités

FIGURE 4 – Texture pour les tuiles des bâtiments

2 Description et conception des états

2.1 Description des états

L'état du jeu est formée par trois couches d'éléments :
— La couche inférieure constituée du terrain.
— La couche intermédiaire constituée des bâtiments,
— La couche supérieure formée par un ensemble d'éléments mobiles (les unités).

2.1.1 Couche inférieure

La couche inférieure correspond à un tableau en 2D d'éléments de « type de terrain ». Un type de terrain correspond à un élément de décor présent sur chaque case de la grille de jeu. Il existe 5 types de terrain :
— Les routes. Celles-ci peuvent être pratiquées par toutes les unités, elles favorisent le déplacement des véhicules.
— Les plaines. Toutes les unités peuvent se déplacer sur celles-ci.
— Les forêts. Les véhicules peuvent difficilement se déplacer sur ces types de terrains.
— Les montagnes. Les véhicules lourds ne peuvent pas s'y déplacer.
— Les villes. Les unités sont affectés de la même manière que les forêts.

2.1.2 Couche Intermédiaire

La couche intermédiaire est constituée par des bâtiments, des éléments qui sont fixes sur le terrain. Ces bâtiments possèdent les propriétés suivantes :
— Position (x,y) dans la grille.
— Color (c) représentant la couleur des équipes qui s'affrontent.
— Id_b(id_b) représentant l'identité de l'unité pour l'affichage dans le rendu.
Les bâtiments sont composé de deux types d'éléments :
— **Le bâtiment principal.** Cet élément contient l'objet à récupérer par le joueur adverse (le drapeau).
— **Les bâtiments secondaires.** Ces éléments permettent la construction des unités sur la même position où le bâtiment se trouve à condition qu'aucune autres unité, allié ou ennemie, ne se trouve sur le bâtiment.

2.1.3 Couche Supérieure

La couche supérieure est constituée par les unités, les éléments mobiles sur le terrain. Ces unités possèdent les propriétés suivantes :
— Position (x,y) dans la grille.
— Color (c) représentant la couleur des équipes qui s'affrontent.
— Vie(v) représentant la vie de l'unité.
— Mvt (m) représentant les points de mouvement de l'unité.
— Puissance (p) représentant la puissance de l'unité.
— Prix (prix) représentant le prix d'achat de l'unité.
— Id(id) représentant l'identité de l'unité pour l'affichage dans le rendu.

2.1.4 État général

L'état général du jeu, en plus du terrain composé des trois couches décrites précédemment, est également défini par deux variables :
— Le tour. Cette variable sert à savoir combien de tours ont été joués depuis le début du jeu et à

déterminer à quel joueur c’est le tour de jouer.
— La fin. Cette variable sert à déterminer si la partie est terminée ou non.

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 :

Classe Unite. Cette classe, étant polymorphe, sert à définir les types d’unités différentes, représentées par les classe filles. Chaque classe dérivée de la classe **Unité** possède des valeurs prédéfinies pour ses différents attributs, correspondant aux caractéristiques du type d’unité. Pour placer les unités sur la grille de jeu, on les stocke dans un tableau à deux dimensions de type vector<vector<Unite*>> (on stocke les pointeurs pour profiter du polymorphisme de la classe). Les coordonnées des unités dans le tableau correspondent à leur position sur la grille. Cette position est définie par la classe **Position**.

Classe Batiment. Cette classe représente les bâtiments servant à fabriquer les unités. Cette classe est donc développé selon le design pattern **Factory**. Comme pour la classe **Unite**, on stocke les instances de **Batiment** dans un tableau 2D.

Classe TerrainTab. Cette classe correspond à un tableau en 2D des éléments de **TerrainTypeId**, une structure de données de type énumération qui sert à représenter les types de terrain.

Conteneurs d’éléments. On retrouve les classes **Terrain** et **Jeu**. La première classe contient les trois tableaux 2D correspondant aux trois couches d’éléments de la grille de jeu que nous avons décrite précédemment. Elle contient également des méthodes de type « getter » qui permettent de renvoyer les éléments présents sur la grille en fonction de leur position. Enfin, la classe **Jeu** contient une occurrence de la classe **Terrain**.

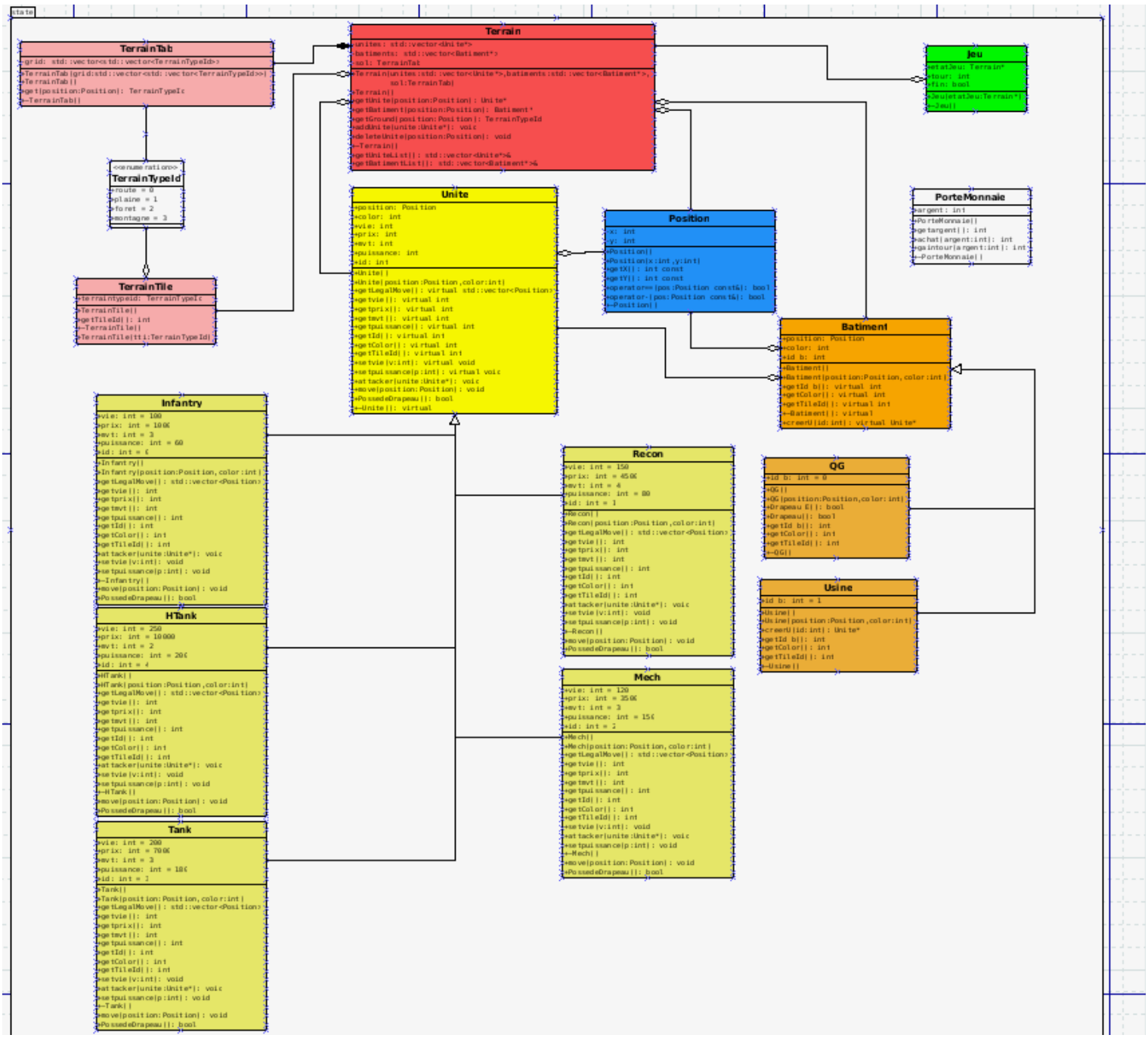

FIGURE 5 – Diagramme UML de l'implémentation de l'état du jeu

3 Rendu : Stratégie et Conception

3.1 Stratégie de rendu d'un état

Pour le rendu d'un état, nous avons opté pour une stratégie assez bas niveau, et relativement proche du fonctionnement des unités graphiques.
Plus précisément, nous découpons la scène à rendre en plans (ou « layers ») : un plan pour le décors (plaine, foret, etc.), un plan pour les bâtiments (QG, usines) et un plan pour les éléments mobiles (les unités). Chaque plan contiendra deux informations bas-niveau qui seront transmises à la carte graphique : une unique texture contenant les tuiles (ou « tiles »), et une unique matrice avec la position des éléments et les coordonnées dans la texture. En conséquence, chaque plan ne pourra rendre que les éléments dont les tuiles sont présentes dans la texture associée.
Pour la formation de ces informations bas-niveau, la première idée est d'observer l'état à rendre, et de réagir lorsqu'un changement se produit. Si le changement dans l'état donne lieu à un changement permanent dans le rendu, on met à jour le morceau de la matrice du plan correspondant.

3.2 Conception logiciel

Le diagramme des classes pour le rendu général est présenté en Figure 6.
Plan :
Le cœur du rendu réside dans la classe Layers. Le principal objectif d'une instance de Layer est de donner les informations basiques pour former les éléments bas-niveau à transmettre à la carte graphique. Ces informations sont données à une instance de Surface, et la définition des tuiles est contenu dans une instance de TileSet. Les classes filles de la classe Layer se spécialise pour l'un des plans à afficher. Pour chaque plan, on a une méthode nommée setSurface() qui fabrique une nouvelle surface, lui demande de charger la texture, puis initialise la liste des sprites. Par exemple, pour afficher le niveau, elle demande un nombre de quads/sprites égal aux nombre de cellules dans la grille avec initQuads(). Puis, pour chaque cellule du niveau, elle fixe leur position avec setSpriteLocation() et leur tuile avec setSpriteTexture().
Surface :
Chaque surface contient une texture du plan et une liste de paires de quadruplets de vecteurs 2D. Les éléments texCoords de chaque quadruplet contient les coordonnées des quatre coins de la tuile à sélectionner dans la texture. Les éléments position de chaque quadruplet contient les coordonnées des quatre coins du carré où doit être dessiné la tuile à l'écran.
Tuiles :
Le patern Factory TileSet regroupent toutes les définitions des tuiles utilisé pour chaque plan à part. Par exemple, il sait que les coordonnées de la tuile avec un QG vert qui est (0,93) et de taille (16,31). Pour obtenir une telle information, un client de ces classes utilise la méthode getTile(). Par exemple si on passe à cette méthode une instance de la classe QG, alors elle va renvoyer une instance de Tile qui correspond à cet élément.

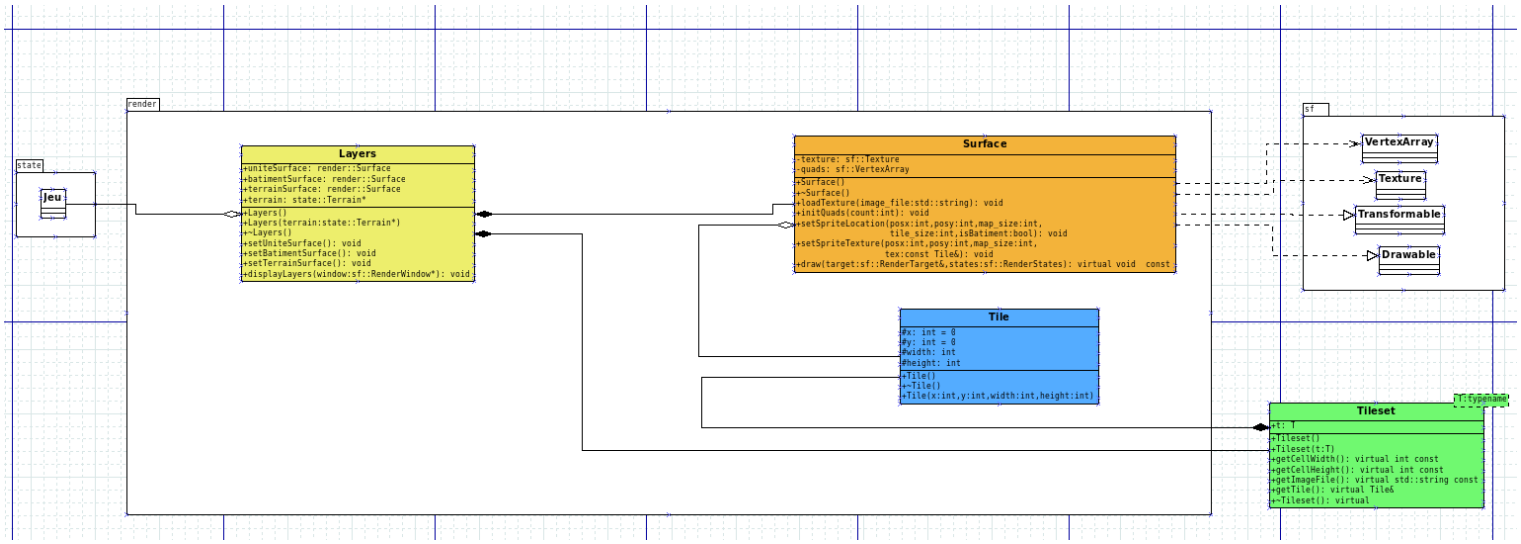

FIGURE 6 – Diagramme des classes de rendu.

4 Règles de changement d'états et moteur de jeu

4.1 Changements extérieurs

Les changements extérieurs sont provoqués par des commandes extérieurs, comme un clic de souris ou un ordre provenant du réseau :
— Commandes principales : « Charger un niveau » : On fabrique un état initial à partir d'un fichier
— Commandes « Move Unit », « Attack Unit », « Creat Unit » et « Capture Flag » : Si cela est possible (pas d'unité bloquantes ou de terrain inaccessible), une unité se déplace toujours selon le nombre de mouvement possible.
Ces changements seront implémenté ultérieurement.

4.2 Changements autonomes

Les changements autonomes sont appliqués à chaque création ou mise à jour d'un état, après les changements extérieurs. Ils sont exécutés dans l'ordre suivant :
1. Si aucune unité ne se trouve sur le le bâtiment de sa couleur, le joueur peut le choisir afin de créer une nouvelle unité.
2. Si une unité est sélectionné, afficher les cases de déplacement possible.
3. Si une unité est adjacente à une unité ennemie, on donne le choix de l'attaque. Si elle attaque :
(a) Si l'unité ennemie est morte, elle disparaît du terrain.
(b) Si l'unité ennemie n'est pas morte, elle contre-attaque.
l'unité ne peut plus réaliser d'autre action pendant ce tour si elle a attaqué.
4. Si une unité se trouve sur le QG ennemie, elle peut récupérer le drapeau de l'équipe adverse.
5. Une unité ne peut réaliser une seul fois les action de chaque type (déplacement, attaque,…).
6. Si le drapeau de l'équipe ennemie se trouve sur sur le QG de l'équipe, le joueur a gagné la partie.

4.3 Conception logiciel

Le diagramme des classes pour le moteur du jeu est présenté en Figure 7. L'ensemble du moteur de jeu repose sur un patron de conception de type Command, et a pour but la mise en œuvre différée de commandes extérieures sur l'état du jeu.
Classes Command. Le rôle de ces classes est de représenter une commande, quelque soit sa source (automatique, souris, réseau, ...). Notons bien que ces classes ne gère**nt** absolument pas l'origine des commandes, ce sont d'autres éléments en dehors du moteur de jeu qui fabriqueron**t** les instances de ces classes.
A ces classes, on a défini un type de commande :
— AttackUnitCommand. Permet a une unité d'attaquer une autre unité adjacente ;
— MoveUniteCommand. Déplace une en fonction de ses points de mouvement ;
— Creat**e**UnitCommand. Permet de créer une unité sur une usine si elle n'est pas occupée et si le prix de l'achat est suffisant ;
— DeleteUnitCommand. Supprime une unité morte (0 point de vie) ;
— CaptureFlagCommand. Permet de capturer le drapeau de l'équipe ennemie.
Engine : C'est le cœur du moteur. Elle stocke les commandes dans une std : :vector avec clé entière. Lorsqu'une nouvelle époque démarre, ie lorsqu'on a appelé la méthode update() après un temps suffisant, le moteur appelle la méthode execute() de chaque commande, incrémente l'époque, puis

supprime toutes les commandes.

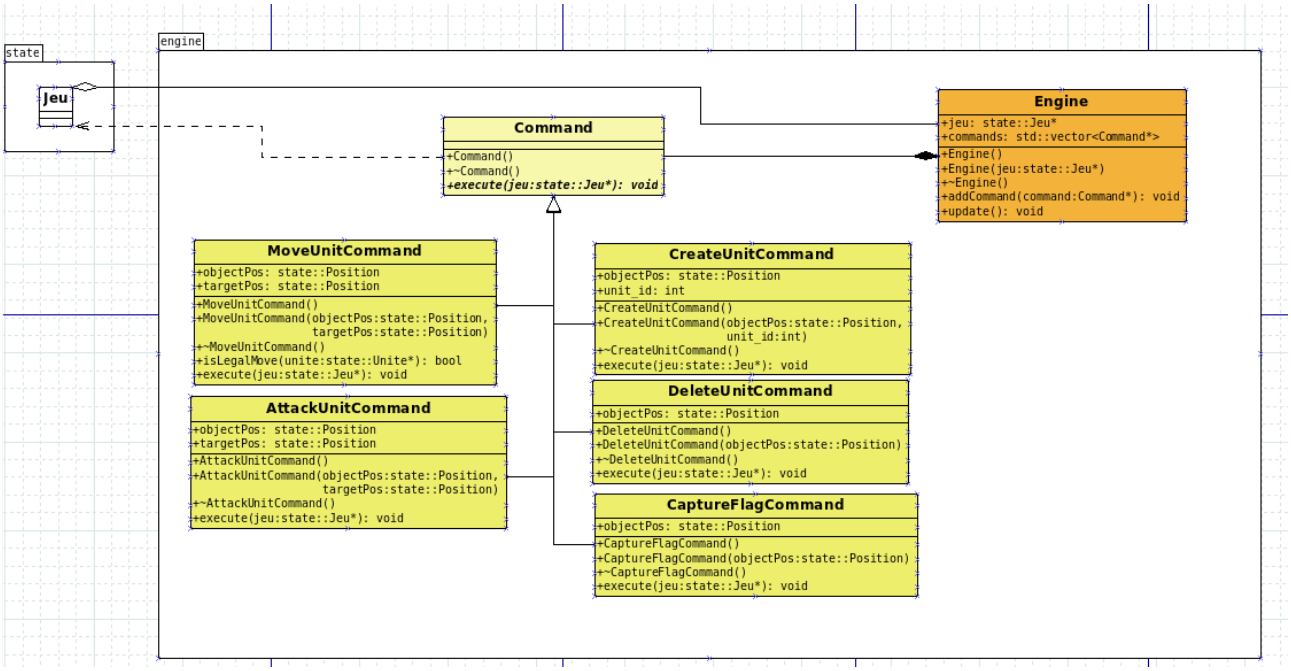

FIGURE 7 – Diagramme des classes de moteur de jeu.

5 Intelligence Artificielle

5.1 Stratégies

5.1.1 Intelligence aléatoire

Cette stratégie est la même pour tous les personnages : lorsque vient le tour d'une IA de jouer, elle liste l'ensemble des actions possibles à chaque état de jeu :
— Créer une unité via une usine
— Sélectionner une unité : l'IA ne peut choisir qu'une unité qui peut encore se déplacer (chaque unité ne peut se déplacer qu'une fois)
— Se déplacer : si une unité est sélectionnée, alors on liste toutes les cases vers lesquelles elle peut atterrir
— Attaquer : valable uniquement si un ennemi est à côté de l'unité sélectionnée
— Passer son tour
Ensuite, l'IA choisit une action aléatoirement parmi toutes celles qu'elle a listées, puis on passe à un nouvel état de jeu.

5.1.2 Intelligence basée sur des heuristiques

Afin d'intégrer dans l'IA une notion d'objectif à atteindre, nous décidons de pondérer chaque action qu'il lui est permis d'effectuer, et de choisir parmi les actions de poids le plus élevé. Les actions possibles ainsi que leur importance diffèrent selon l'état du jeu.
Lorsqu'une IA commence à jouer :
— Si elle possède au moins une unité, elle en sélectionne prioritairement une de disponible pour jouer.
— Moins l'IA possède d'unités, plus elle aura de chances de sélectionner une usine pour créer des unités, et inversement. Ainsi, plus l'IA a d'unités, plus elle préférera économiser pour créer des unités plus puissantes.
— En dernier recours, elle passe son tour.
Si l'IA a sélectionné une usine, alors elle choisira en priorité l'unité la plus puissante qu'elle peut créer.
Si l'IA a sélectionné une unité :
— Si une unité ennemie se trouve à proximité, alors l'unité sélectionnée attaquera en priorité l'unité ennemie.
— Sinon, on effectuera un déplacement.

Pour déplacer une unité, on lui attribue un objectif à poursuivre. Cet objectif est choisi selon plusieurs cas de figures :
— L'objectif principal est le drapeau ennemi qu'il faut capturer. Cependant, si le drapeau de l'IA a été capturé ou qu'il se trouve en dehors du QG, alors il devient la cible prioritaire.
— Si l'unité sélectionnée a réussi à capturer le drapeau ennemi ou qu'elle a retrouvé son drapeau, alors son objective est le QG de son propre camp.
— Si le drapeau ennemi est dans les mains d'une unité, alors cette unité devient la cible, que ce soit un allié à escorter ou un ennemi à attaquer.
Une fois l'objectif attribué, trois déplacements sont envisagés :
— Si un ennemi se dresse entre l'unité sélectionnée et l'objectif, se déplacer vers l'ennemi devient prioritaire.
— En deuxième choix, on privilégiera les déplacement qui rapprochent l'unité de son objectif.

— Tout autre déplacement aura une importance faible.

5.1.3 Intelligence avancé

Nous proposons une intelligence plus avancée en suivant les méthodes de résolution de problèmes à états finis. Dans cette configuration, un état est un état du jeu à une époque donnée, tel que défini en section 2. Les arcs entre les sommets du graphe d'état sont les changements d'états, définis en section 4. Passer d'un sommet du graphe d'état à un autre reviens à passer d'une époque à une autre du jeu, fonction de l'ensemble des commandes reçues (clavier, réseau, IA,. . .). Le score d'un état du jeu est déterminé par la variation de distance entre l'unité sélectionnée et son objectif respectif lorsque celle-ci effectue un déplacement. Cet objectif est choisi en sélectionnant l'entité la plus proche parmi les suivantes :
— Le drapeau ennemi le plus proche de l'unité
— Le plus proche ennemi possédant un drapeau
— Si l'unité sélectionnée possède un drapeau, son objectif sera alors son propre QG.
Le score peut aussi changer avec la possibilité d'attaquer un ennemi voisin sur le chemin. Le meilleur choix de mouvement pour une unité est donc celui du plus court chemin dans le graphe d'état qui mène vers un score maximal.
Pour trouver ce chemin, nous suivons des méthodes basées sur les arbres de recherche, avec une propriété importante. En effet, nous n'envisageons pas de dupliquer l'état du jeu à chaque sommet du graphe d'état : compte tenu du nombre de nœuds que nous allons traiter, nous aurions rapidement des problèmes de mémoire. Nous n'allons considérer qu'un seul état que nous modifions suivant la direction choisie par la recherche. Si le sommet suivant est à une époque suivante, i.e. on descend dans l'arbre de recherche, on applique les commandes associées, et notre état gagne une époque. Si le sommet suivant est à une époque précédente, i.e. on remonte dans l'arbre de recherche, on annule les commandes associées, et notre état retrouve sa forme passée.

5.2 Conception logiciel

Le diagramme des classes pour l'intelligence artificielle est présenté en Figure 8.
Classes AI. Les classes filles de la classe AI implante différentes stratégies d'IA, que l'on peut appliquer pour un personnage :
— RandomAI : Intelligence aléatoire
— HeuristicAI : Intelligence basée sur des heuristiques
— DeepAI : Intelligence avancée

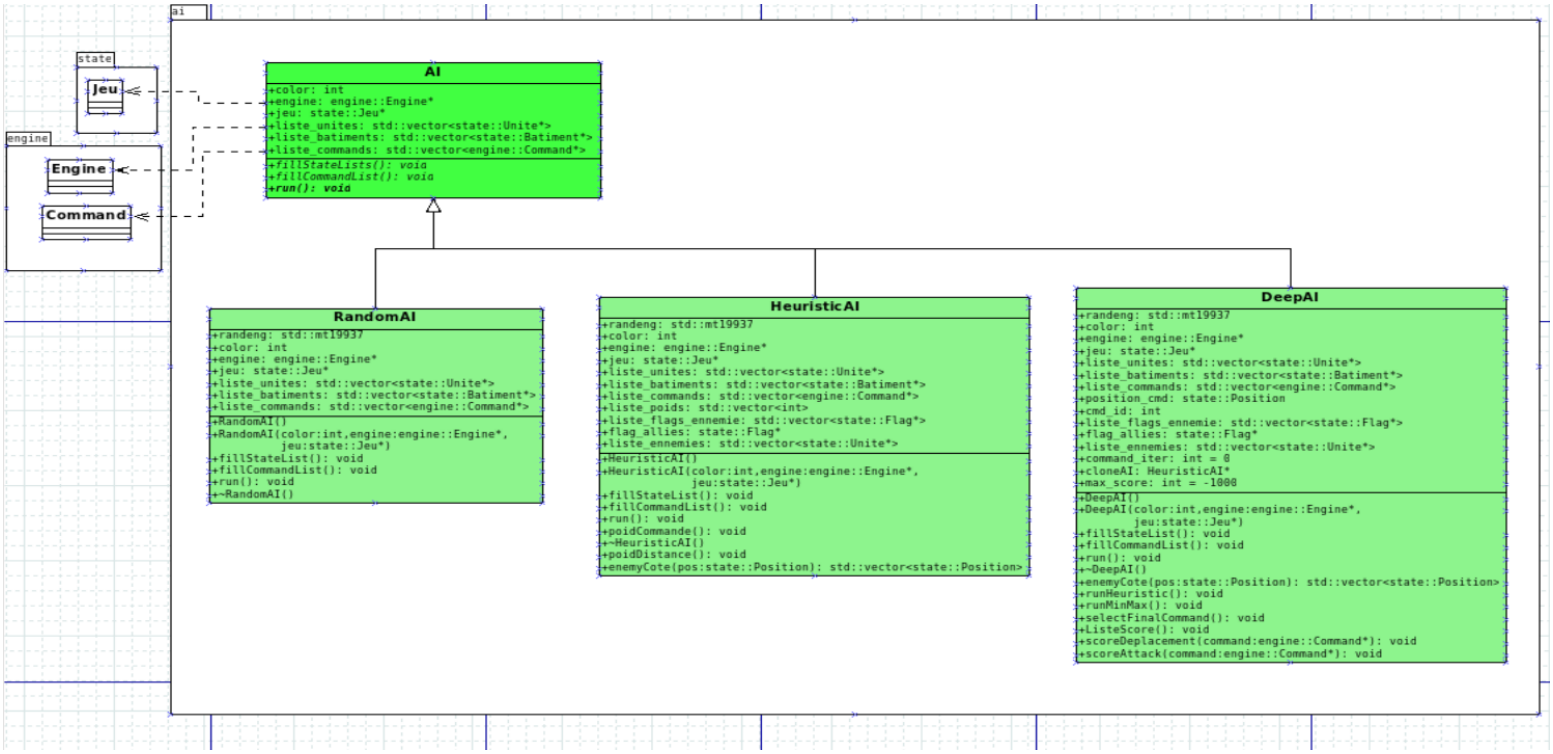

FIGURE 8 – Diagramme des classes d’intelligence artificielle.

6 Modularisation

6.1 Organisation des modules

6.1.1 Répartition sur différents threads

Notre objectif ici est de placer le moteur de jeu avec les IA et le moteur de rendu graphique sur deux threads séparés. Le thread principal sert à initialiser les différentes ressources, à savoir l'état de jeu, le moteur de jeu, les IA et le moteur de rendu, puis à lancer les deux threads secondaires.
Nous avons deux types d'information qui transite d'un module à l'autre : les commandes et les notifications de rendu.

Commandes. Lorsque le moteur de jeu se met à jour, il notifie le client afin que celui-ci fasse fonctionner les IA et que celles-ci envoient des commandes au moteur (lorsque c'est à leur tour de jouer). Une fois les commandes reçues, le moteur exécute une à une toutes les commandes qu'il reçoit. En effet, il est possible que le moteur reçoive plusieurs commandes avant de les traiter, voire même qu'il reçoive des commandes pendant qu'il en traite. C'est pourquoi la liste de commandes qu'il traite est considérée comme une FIFO.

Notifications de rendu. Une fois que le moteur de jeu a fini, il notifie le client qu'il a fini de se mettre à jour, puis il vide la liste des commandes.
Par manque de temps, nous n'avons pas fait en sorte que le moteur graphique soit à l'écoute des notifications du moteur afin d'optimiser le rafraîchissement de l'écran.

Note : Nous avons mis en place un système de sauvegarde et de chargement des commandes :
— Lorsqu'on choisit d'enregistrer une partie, le moteur sauvegarde chaque commande exécutée puis à la fin de la partie, il les stocke dans un fichier au format JSON.
— Il est alors possible de rejouer une partie précédemment enregistrée : le moteur recharge alors les commandes stockées au format JSON puis les convertit en commandes exécutables avec lesquelles il remplit sa liste.

6.2 Conception logicielle

Le diagramme des classes pour l'intelligence artificielle est présenté en Figure 9.

Client. La classe Client contient toutes les informations pour faire fonctionner le jeu : Moteur de jeu (avec état intégré), intelligences artificielles, et rendu. Cette classe est un observateur du moteur de jeu :
— Lorsque le moteur est sur le point d'exécuter ses commandes (méthode engineUpdating()), il appelle les IA pour ajouter les commandes des robots.
— Lorsque le moteur a terminé d'appliquer les commandes (méthode engineUpdated()), il vide le cache des commandes exécutées.

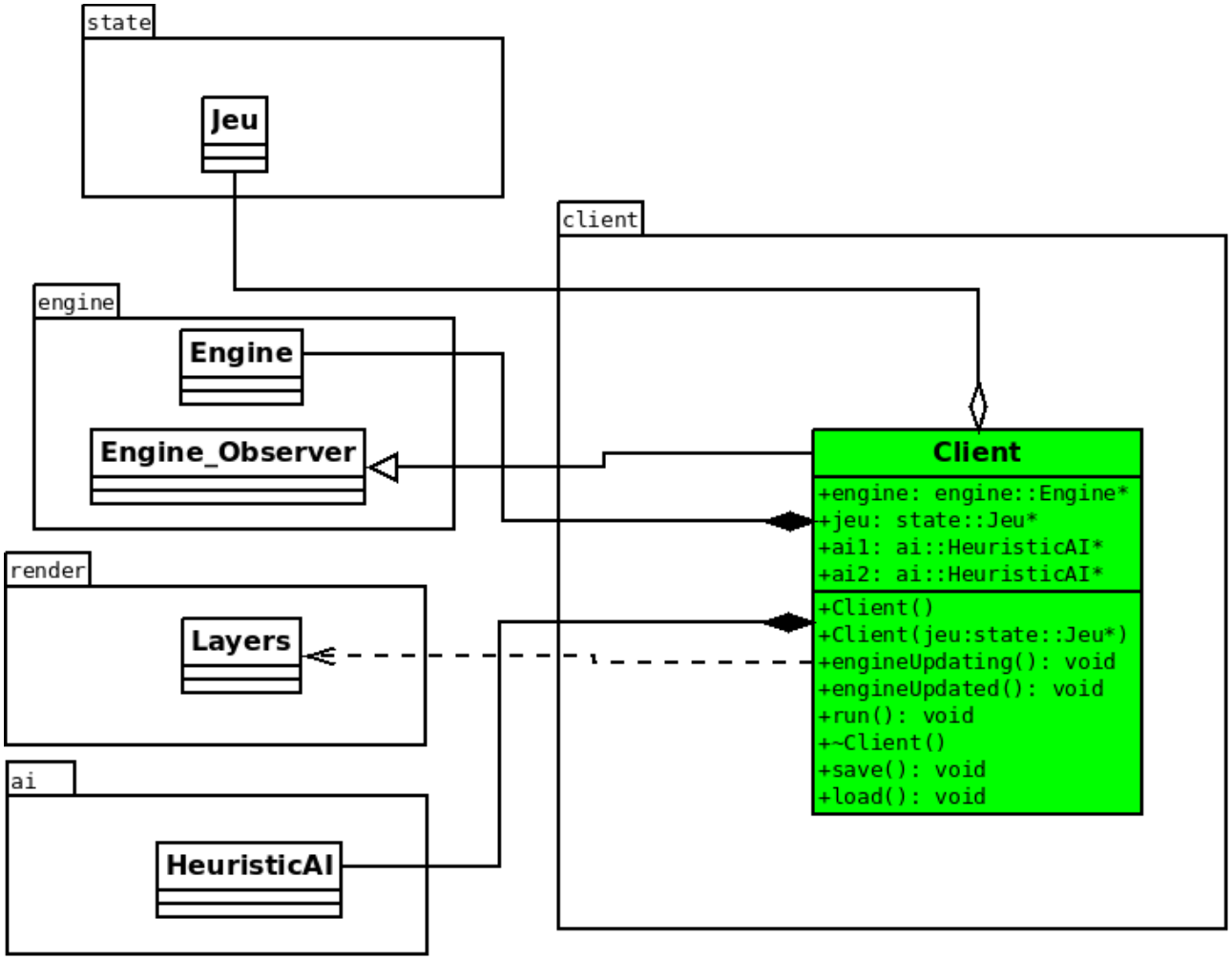

FIGURE 9 – Diagramme des classes pour la modularisation.