Projet Logiciel Transversal

Pierre CHABAUD – Alexandre CHAMINAS

FIGURE 1 – Capture d'écran du jeu robostrike.com

Table des matières

1 Objectif

1.1 Présentation générale

L'objectif de ce projet consiste en la réalisation du jeu multijoueur robostrike(Figure 1) lui même inspiré du jeu de société RoboRally(1994). Les joueurs doivent manœuvrer de manière stratégique un robot sur un quadrillage comprenant obstacles, pièges et bonus afin d'atteindre en premier des balises.

1.2 Règles du jeu

Lors d'une manche, les joueurs choisissent six actions ordonnées (avancer, reculer, droite, gauche, attaquer, …) avant la fin du temps imparti. Une fois que les joueurs ont tous entré leurs ordres ou que le temps est écoulé, le jeu exécute les actions des joueurs tour par tour.

Les manches s'enchaînent jusqu'à la victoire finale de l'un des joueurs. Le gagnant est le joueur qui atteint le premier tous les objectifs ou qui est le dernier survivant.

A l'exception du départ, plusieurs joueurs ne peuvent pas se trouver sur la même case. Si deux joueurs se retrouvent sur une même case, une collision a lieu et le joueur présent initialement sur la case disputée se fait pousser sur une case adjacente.

Chaque joueur commence la partie avec cinq points de vie et toute attaque entraîne la perte d'un point de vie pour le joueur touché.

Les joueurs obtiennent des bonus tels que des soins, des pièges ou des améliorations d'attaque en passant sur des cases bonus.

1.3 Conception Logiciel

Les dépendances et paquets utilisés sont communs à tous les projets 3IS et sont disponibles sur le dépôt github « github.com/cbares/plt ». La bibliothèque graphique retenue est la SFML. Le schéma ci dessous (Figure 2) montre l'aspect final du projet avec une partie serveur et une partie client. Le serveur sera basé sur micro_httpd qui est un serveur HTTP minimaliste.

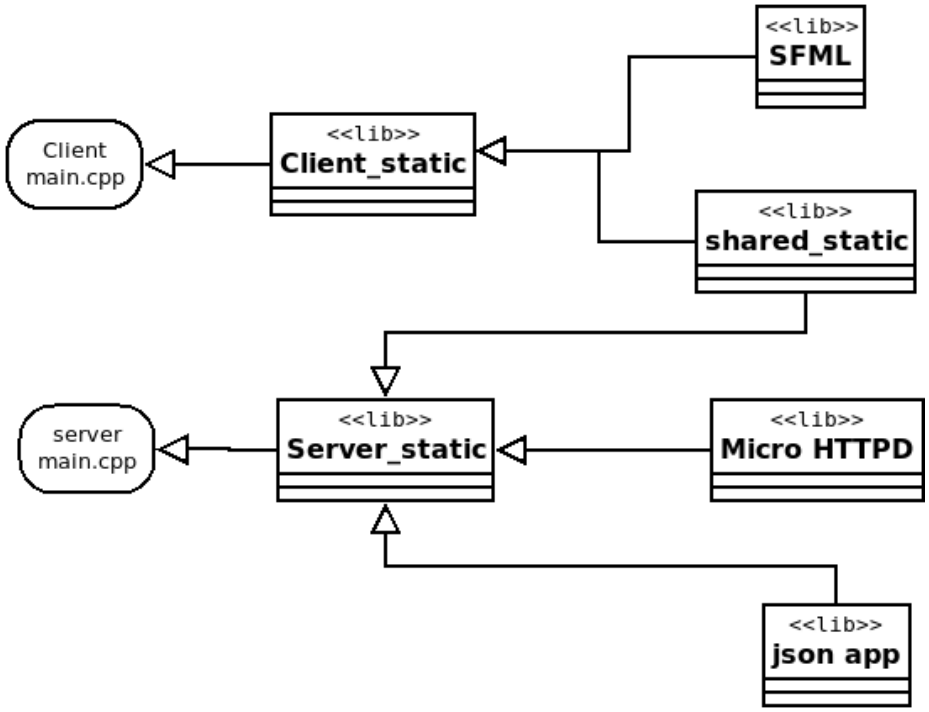

FIGURE 2 – Schéma de la structure logicielle du projet

2 Description et conception des états

2.1 Description des états

Un état du jeu est formée par un ensemble d'éléments fixes (la carte de jeu) et d'éléments mobiles (les robots). Tous les éléments possèdent les propriétés suivantes :

- Coordonnées (x,y) dans la grille
- Identifiant de type d'élément : ce nombre indique la nature de l'élément (ie classe)

2.1.1 État éléments fixes

Le terrain de jeu est formé par un ensemble d'éléments « tuiles » composant la carte de jeu. La taille de cette carte est fixée au démarrage du niveau et ne changera pas au cours de la partie. Les types de tuiles sont :

Tuile « Normale ». Les tuiles « normales » sont des éléments neutres et donc franchissables sans risque par les robots.

Tuile « Objectif ». Les tuiles « objectif » correspondent aux objectifs que le joueur doit atteindre en totalité pour gagner la partie. Ces éléments sont numérotés de 1 à N. La tuile objectif dont le numéro est le plus élevé sert de point de réapparition après la mort d'un joueur et le départ de la partie est donné sur l'objectif n°1. De manière purement esthétique, le joueur voit les objectifs d'une couleur différente une fois qu'il est passé dessus.

Tuile « Trou ». Les tuiles « trou » sont des éléments franchissable qui causent la mort instantanée

du robot qui s’y est engagé ou qui a été poussé dedans.

Tuile « Mur ». Les tuiles « mur » sont des éléments infranchissable pour les éléments mobiles.

Tuile « Bonus ». Les tuiles « bonus » sont des éléments franchissables pour les éléments mobiles. Cette tuile peut contenir :

— Une bombe : explose à la fin de la manche en infligeant deux points de dégât aux joueurs touchés dans un rayon de deux cases et les immobilise pour le tour suivant

— Une trousse de soin : régénère toute la vie de vie dès utilisation

— Un tir en croix : inflige deux points de dégât aux joueurs les plus proches dans les directions Nord, Sud Est et Ouest dès utilisation.

— Un bouclier : protège le joueur dès activation et jusqu’à la fin de la manche

— Une attaque automatique : inflige un point de dégât au joueur en face a chaque action de manière automatique et augmente d’un point la puissance des attaques de face et en croix jusqu’à la fin de la manche

Tuile « Rotation ». Les tuiles « rotation » sont des éléments franchissables pour les éléments mobiles et on pour capacité de modifier l’orientation du robot d’un quart de tour dans le sans horaire ou anti-horaire selon le type.

Tuile « Tapis roulant ». Les tuiles « tapis roulant » sont des éléments franchissables pour les éléments mobiles et on pour capacité de modifier l’orientation du robot ainsi que sa position sur la grille.

2.1.2 État éléments mobiles

Dans ce projet un seul type d’élément mobile est nécessaire : le type « Robot ». Il possède une orientation (nord, sud, est, ouest), une couleur, une liste d’objectifs visités, un nombre de points de vie, un nombre de vies et des indicateurs de possession et d’utilisation de bonus.

Élément mobile « Robot ». Cet élément est dirigé par les joueurs ou une IA, qui commande la propriété de déplacement, d’attaque et d’utilisation des bonus. De plus il possède la propriété « couleur », qui est purement esthétique et dont la seule règle est qu’elle est unique. On utilise également une propriété que l’on nommera « ActionStatus », et qui peut prendre les valeurs suivantes :

— Status « normal » : cas le plus courant, où le robot peut se déplacer sur la carte et cherche à atteindre les objectifs

— Status « paralysé » : cas où le robot à été touché par l’explosion d’une bombe à la toute fin de la manche précédente. Il est alors incapable de faire des actions et subit l’influence des éléments de jeu de la carte et les attaques des autres joueurs

— Status « mort » : cas où le robot est tombé dans un piège ou a perdu son dernier point de vie. Cet état intervient entre deux manches avant la réapparition ou dans l’attente de la fin de la partie (le joueur n’a plus de vies mais que les adversaire toujours en jeu la finissent).

2.1.3 État général

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

— Epoque : représente « l'heure » correspondant à l'état, ie c'est le nombre de « tic » de l'horloge globale depuis le début de la partie

— Timer : représente le nombre de « tic » de l'horloge globale depuis le début de la manche afin de donner la priorité de jeu au joueur ayant validé ses actions le premier. Dans un premier temps l'ordre sera établi en fonction de l'identifiant du robot.

2.2 Conception logicielle

Le diagramme des classes pour les états est présenté en Figure 3, dont nous pouvons mettre en évidence les groupes de classes suivants :

Classes Element. Toute la hiérarchie des classes filles d'Element (en jaune) permettent de représenter les différentes catégories et types d'élément. Nous n'avons pas utilisé le polymorphisme pour faire de l'introspection ne maîtrisant pas encore parfaitement ce concept. Cependant la méthode isStatic() ou isReachable() s'y prêterait tout à fait et nous travaillons activement à leur implémentation.

Conteneurs d'élément. La classes State permet de contenir des ensembles d'éléments. A partir cette classe nous pouvons accéder à toutes les données de l'état. Par exemple, « twoDTab » est un tableau à deux dimensions d'éléments de type « MapTile » permettant de contenir la carte de jeu.

Fabrique d'élements. Dans le but de pouvoir fabriquer facilement des instances d'Element et plus particulièrement de « MapTile » (« Robot » étant créé une seule au début de la partie et de manière simple) nous utilisons la classe MapFactory. Cette classe permet de créer n'importe quelle instance non abstraite à partir d'un caractère. L'idée est de l'utiliser, entre autres, pour créer un niveau à partir d'un fichier texte.

Observateurs de changements. La conception de ces classes suit le Design Pattern Observer. Le but de la classe Observable dont hérite « State » est d'enregistrer des observateurs puis de les notifier à chaque changement d'état. Par exemple la classe StateLayer (render.dia) est un observateur et met à jour les textures/sprites à chaque action d'un personnage

2.3 Conception logicielle : extension pour le rendu

2.4 Conception logicielle : extension pour le moteur de jeu

2.5 Ressources

Les ressources graphiques comprennent trois packs de textures de 64 par 64 pixels (figures 2, 3 et 4) ainsi que sur un fichier de police d'écriture « roboto.ttf ».

FIGURE 2 – Textures pour les tuiles du décor

FIGURE 3 – Textures pour les robots

FIGURE 4 – Textures pour les commandes de déplacement et d’attaque

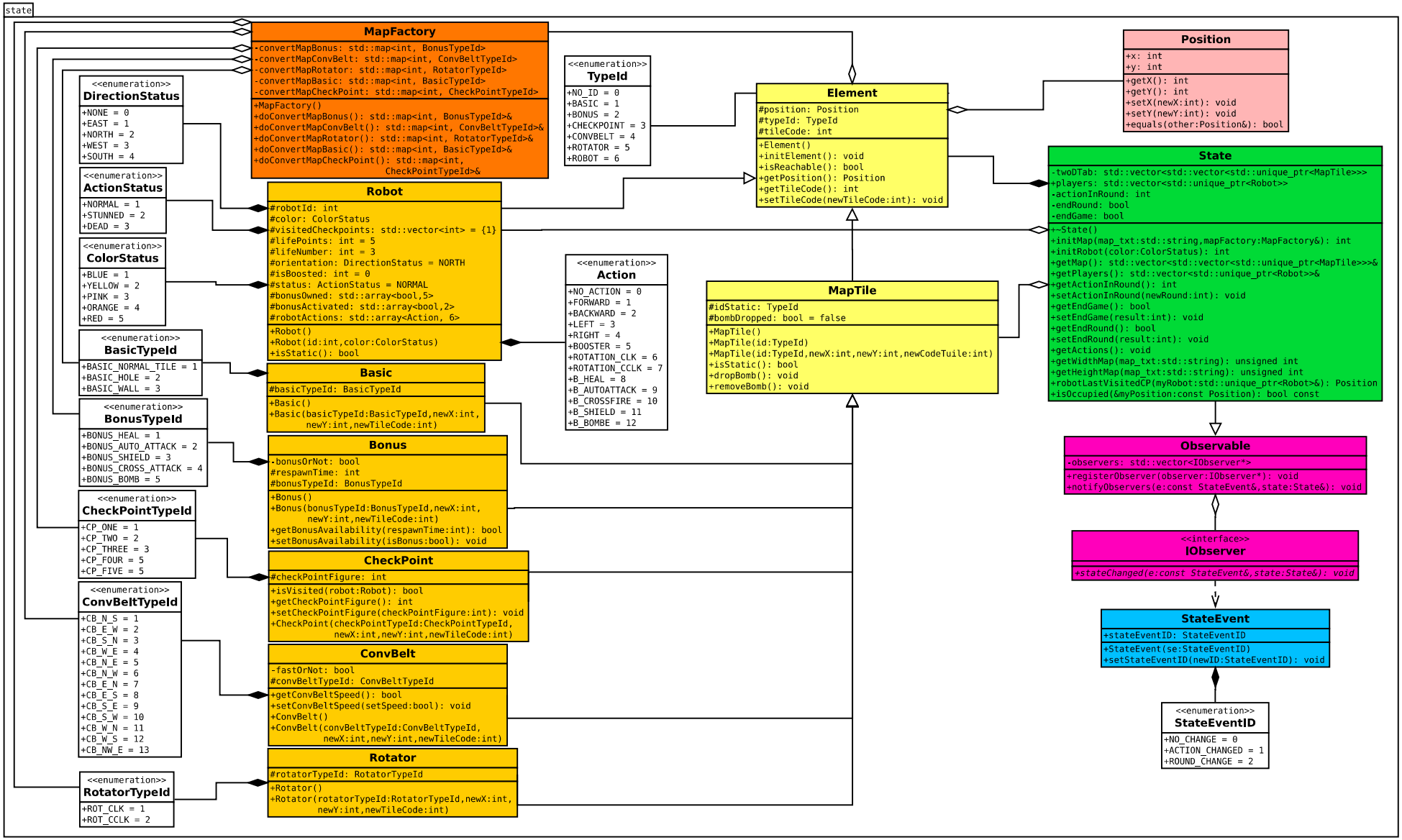

FIGURE 5 – Diagramme de classes du moteur d'états

3 Rendu : Stratégie et Conception

3.1 Stratégie de rendu d'un état

Le rendu de la carte de jeu est effectué sous forme de tuiles carrée générées à l'aide de la bibliothèque SFML.

La fenêtre de jeu est divisée en deux parties distinctes. L'aire de jeu se trouve à gauche et comporte les différentes tuiles. Un zoom est appliqué pour rester dans des valeurs de fenêtre raisonnables en cas de map trop grande ou trop petite. A droite est affichée la partie commande ainsi que les informations en textes sur les statistiques de la partie (niveau de vie des joueurs et nombre de vie des joueurs restants).

La grille de la carte est créée à partir d'unfichier texte « map.txt **»** situé dans les ressources et dont de choix de la taille est laissée à l'appréciation du joueur. Un tableau stocké dans « MapFactory » permettant de relier un entier présent dans le fichier « map.txt », un type d'élément et un nombre permettant de choisir la position du sprite sur l'image « map.png ». Cela est réalisé à l'aide d'un std::map qui associe à chaque code de tuile un ID de terrain spécifique.

La méthode initMap de la classe State (package state) permet de fabriquer pour chaque code tuile du fichier « map.txt **»** le terrain correspondant, de créer un pointeur unique vers cet objet et de l'ajouter à la grille (tableau à deux dimensions contenant des pointeurs de Terrain).

Le reste de l'affichage est codé en « dur » car ne sera pas modifié d'une partie à l'autre (seul le nombre d'objet varie).

3.2 Conception logiciel

Classe TileMap: cette classe possède deux attributs : une texture et un tableau de Vertex (quads) contenant la position des éléments et leurs coordonnées dans la texture (map.png). Elle possède les méthodes loadGrille, loadPersonnage et loadCurseur lui permettant d'initialiser ses attributs à partir d'un tableau de Terrain, d'une liste de Personnage ou d'un Curseur. La méthode Draw a pour but de dessiner une texture pour ensuite permettre son affichage dans une fenêtre. Elle est codés en virtuelle pour surcharger la fonction *draw* de la SFML.

Classe TileSet : Cette classe possède plusieurs attributs : un id de type TileSetID, des entiers cellWidth et cellHeight (qui représentent respectivement la largeur et la longueur en pixel d'une tuile) et une chaîne de caractères imageFile (chemin vers un « fichier.png »). L'ID peut prendre différentes valeurs :

- MAP_TILESET
- COMMAND_TILSET
- PAWN_TILESET

En fonction de l'ID passé en TileMap addapte ses attributs cellWidt, cellHeight ainsi que l'arborescence imageFile. La classe possède deux attributs : une texture et un tableau de Vertex (quads) contenant la position des éléments et leurs coordonnées dans la texture (map.png).

Classe StateLayer : Cette classe est la classe principale du moteur de rendu et centralise les informations au de la même manière que la classe « State » pour le moteur d'états. Cette classe prend en paramètre une référence à une instance de « State ». Elle possède également un tableau

de pointeurs de TileSet et un autre de pointeurs de Surface. Le but de cette classe est de créer deux surfaces grâce à la méthode initSurface (grille de jeu et joueurs) et d'initialiser leurs textures. Cette classe est un observateur, elle implémente l'interface IObserver pour être avertie des changements d'état. Elle réagit ensuite en actualisant les textures.

3.3 Exemple de rendu

s

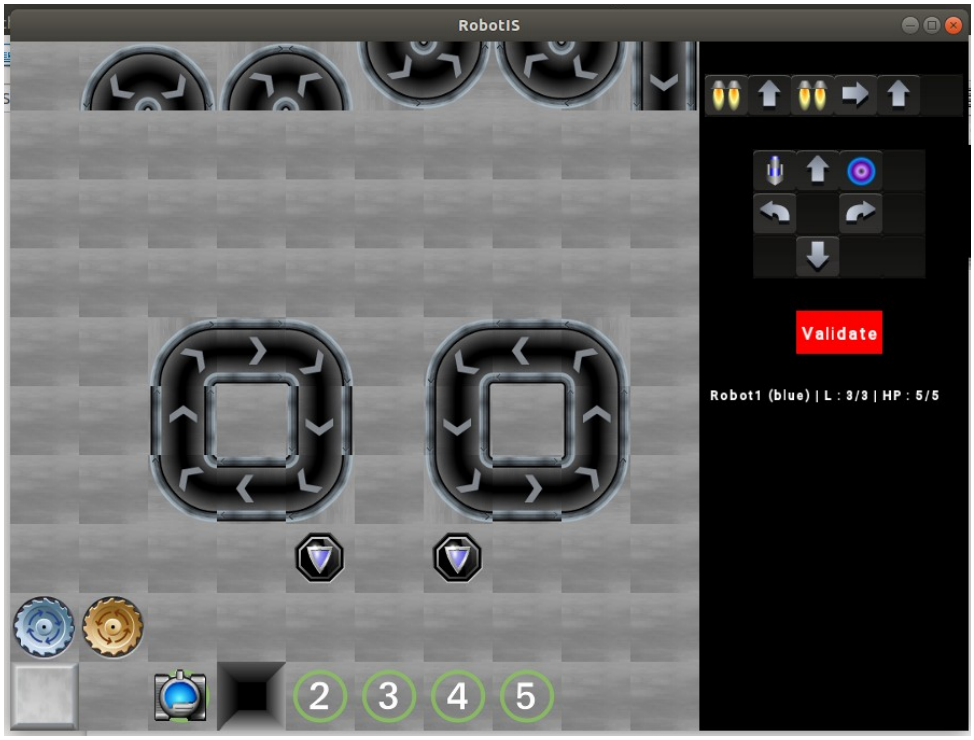

FIGURE 6 – Exemple d'affichage de la fenêtre de jeu

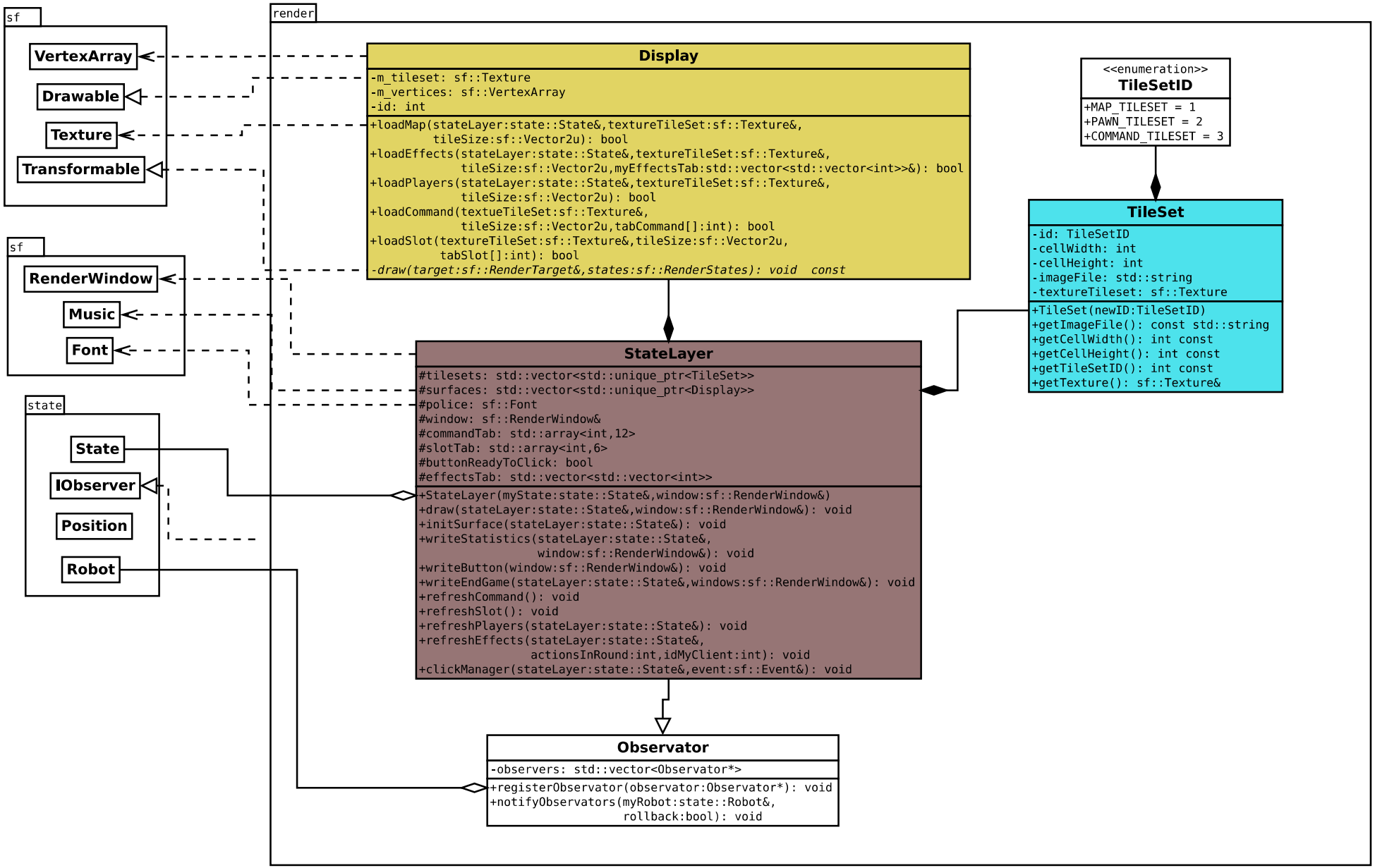

FIGURE 7 – Diagramme de classes du moteur de rendu

4 Règles de changement d'états et moteur de jeu

4.1 Changement d’état

Le jeu est découpé en manches où chaque robot effectue une série de six actions. Une manche débute lorsque tous les joueurs/robots ont validé leurs six actions ou que le temps imparti est écoulé (généralement 30 secondes). Les robots possèdent un statut qui définit leur capacité à effectuer les actions :

- NORMAL : le robot est capable d’effectuer ses actions sans contraintes particulières
- STUNNED : le robot a été touché lors de la manche précédente et ne peut donc pas effectuer d’action lors de la prochaine manche. Il reste cependant sur la carte et subi les actions des autres joueurs et de la carte de jeu
- DEAD : le robot n’a plus de points de vie. Ce statut peut survenir au cours de la manche jusqu’à la réapparition du joueur s’il lui reste des vies ou de manière plus durable jusqu’à la fin de la partie si ses vies sont consommées

Lorsque le robot a un statut NORMAL, le moteur de jeu prend en compte ses actions :

- Se déplacer : avancer, reculer, tourner, boost, ...
- Attaquer : attaque de face, attaque circulaire, attaques spéciales
- Utiliser un bonus : utilisation des cinq types de bonus. Ces bonus sont les suivants :
  - *Heal* : Sert à restaurer ses 5 points de vie au maximum
  - *Auto attack* : Le robot lancera une attaque à chaque action de la manche, en plus de son action habituelle
  - *Cross attack* : Attaque se lançant dans les 4 directions
  - *Bomb* : Pose une bombe qui explose en fin de manche
  - *Shield* Bouclier protégeant le robot des attaques adverses

Le moteur de jeu interprète les actions en fonction de l’ordre croissant des 6 actions entrées par les joueurs. Toutes les premières actions sont effectuées, puis le moteur traite les deuxièmes, etc. Les actions de même numéro sont effectuées par ordre de priorité suivant (1 étant la priorité maximale et 4 la minimale) :

1. Utilisation des bonus
2. Attaques (y compris issues du bonus *auto attack*)
3. Déplacements
4. Actions de la carte de jeu sur le robot (trou, plaque tournante, tapis roulant), …

Les actions de même priorité sont exécutées en même temps à l’exception des actions de déplacement qui sont effectuées en fonction de la rapidité à jouer des robots.

Chaque action traitée par le moteur de jeu modifie l’état du jeu.

4.2 Conception logicielle

Le diagramme des classes pour le moteur de jeu (« **engine.dia** ») est présenté en Figure 8.

Classe Engine : Cette classe est centrale et permet de stocker un Etat. Elle a accès à toutes les actions entrées par les joueurs ainsi qu'à leurs attributs. Pour chaque ordre créé par les joueurs, le moteur crée 6 commandes et les exécute action après action. Lorsque la méthode « update » est appelée, le moteur appelle la méthode « executeOrder » de chaque commande puis supprime toutes les commandes une fois exécutées.

Classes Commandes : Les classes Attack, Environment, Move et UseBonus héritent de la classe Command et réalisent une action pour un joueur. Elles possèdent chacune une méthode « executeOrder » qui fait effectuer à un personnage l'action correspondante.

Classes UseBonus, Rotation, Attack et Move : Ces classes exécutent une action et mettent à jour l'état du jeu.

4.3 Horloge globale

4.4 Conception logiciel : extension pour l'IA

4.5 Conception logiciel : extension pour la parallélisation

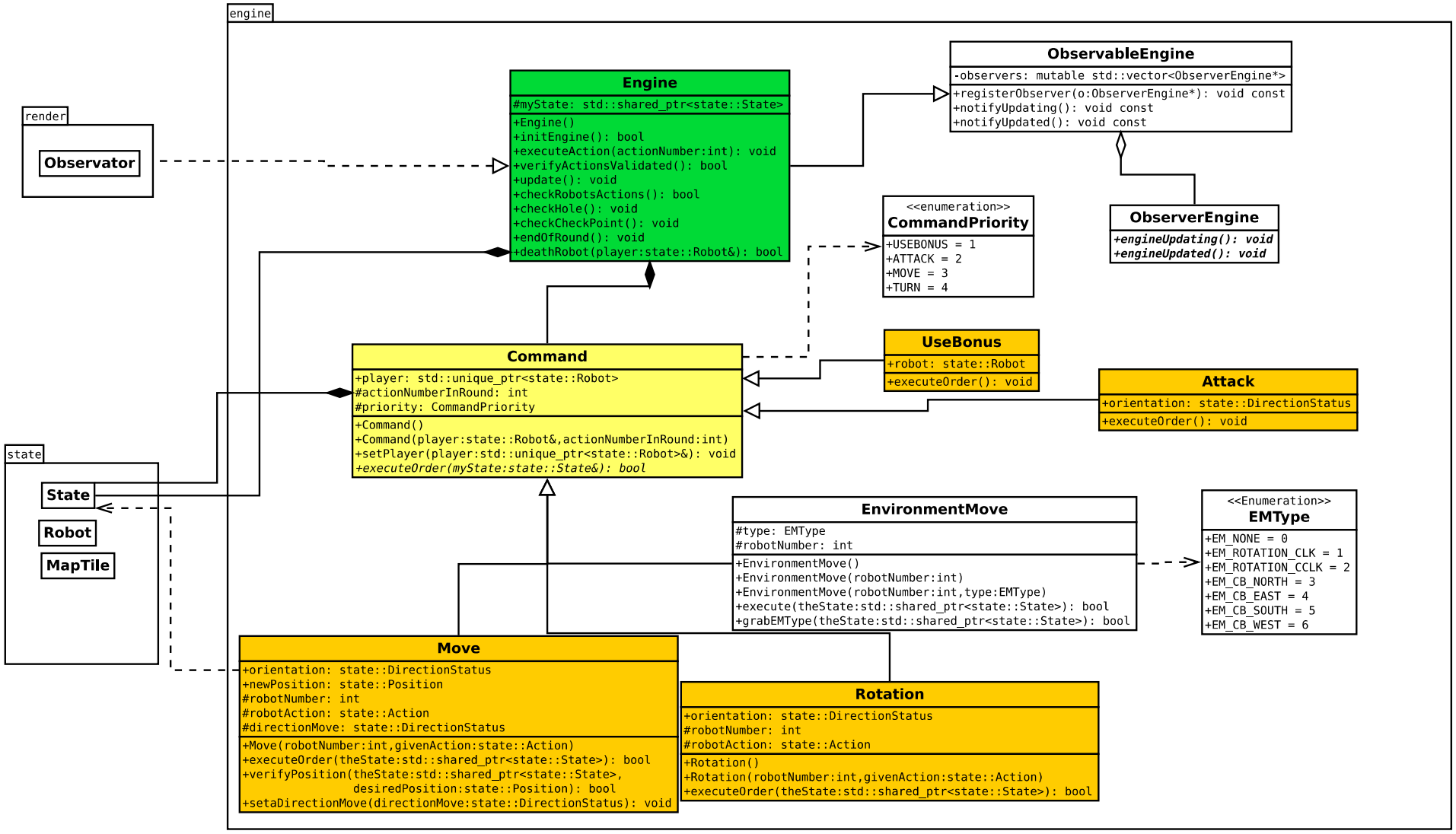

FIGURE 8 – Diagramme de classes du moteur de jeu

5 Intelligence Artificielle

5.1 Stratégies

Chaque IA contrôle un robot afin de lui faire effectuer les mouvements et actions permettant de gagner le jeu. Nous avons décidé que nos IAs aient deux caractéristiques importantes:

- Une IA se comporte de manière indépendante et ne communique pas avec les autres IA (le jeu étant un jeu individuel et ne comprenant pas d'équipes)

- Le choix d'actions d'une IA se fait sans mémoire : la série de six actions est uniquement calculée en fonction de l'état (disposition de la carte, bonus possédés, état des adversaires, incluant leur vie et leurs bonus possédés)

On en conclut que chaque IA sera une fonction qui prend en entrée l'Etat (en lecture seule) et le n° de robot , et sort un vecteur de 6 actions (stocké directement dans la liste d'actions du joueur).

5.1.1 Intelligence minimale/aléatoire

L'IA aléatoire (dans notre jeu notée «RandomAI») contrôle un robot en lui faisant effectuer de manière aléatoire des actions de déplacement ou de rotation. Pour cela, six entiers aléatoires entre 0 et 3 sont choisis et à chaque valeur est attribuée une action. Ainsi, 0 correspond à avancer, 1 à reculer, 2 à une rotation antihoraire, 3 à une rotation horaire. L'état du robot est auparavant vérifié pour vérifier qu'il n'est pas immobilisé ou mort. Une fois que les 6 actions sont choisies, la fonction renvoie *true* si le tour a bien été choisi au hasard.

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 un chance à l'IA de remporter la partie.

On privilégie presque toujours les objectifs et donc les déplacements à l'attaque. La victoire est accordé au joueur qui atteint tous les objectifs le premier. L'ia cherche donc le plus court chemin pour passer sur tous les objectifs. Dans ce jeu le plus court chemin n'est souvent pas celui qui necessite le moins d'actions. L'absence de téléportation latéral directe implique que le déplacement en diagonal n'est pas la meilleur option pour se déplacer efficacement (FIGURES 9 et 10).

FIGURE 9 – Plus court chemin pour atteindre le checkpoint n°5.

FIGURE 10 – Possibilités d’instructions pour atteindre le checkpoint n°5.

Notre stratégie de déplacement consiste à ce mettre à la hauteur du checkpoint puis à se téléporter latéralement dessus. Un système de contournement d’obstacle (murs et trous) ainsi que la prise en compte des tapis roulants et plaques tournantes sera pris en compte.

L’ia calcule la probabilité de chaque joueur de gagner en fonction des objectifs déjà visités, le la position sur la carte et du nombre de vie et points de vie restants. Si un joueur est proche de la victoire l’ia essaye de l’attaquer. Le calcul du pourcentage de victoire d’un joueur est effectué de la manière suivante :

- un calcul de pourcentage de checkpoint validé sur le nombre total de checkpoints

- chaque vie perdu retire 10 % de chance de victoire

- un nombre de points de vie inférieur ou égale à 2 retire 30 % de chance de victoire car le robot peut se faire « one-shot »

- la probabilité de victoire en pourcentage est exprimée entre 0 et 100

Si l’ia voit ses points de vie arriver à un niveau critique elle essayera de se protéger voir de se soigner en utilisant un bonus de soin ou un bouclier. Si elle ne possède pas de bonus elle pourra décider de rejoindre une case bonus pour s’en procurer un.

5.1.3 Intelligence artificielle

Une manche de jeu comporte six actions qui doivent être saisies au même moment, et il existe quinze types d’actions différents, soit 15^6=11 millions de combinaisons possibles. Même si toutes les combinaisons d’actions ne sont pas autorisées (ex : utilisation d’un même bonus une seule fois par manche) ce nombre reste trop élevé pour une résolution avec arbre fini.

Nous proposons une intelligence à base d’algorithme génétique. Ce type d’algorithme permet d'obtenir « une solution approchée à un problème d'optimisation, lorsqu'il n'existe pas de méthode exacte, ou que la solution est inconnue, pour le résoudre en un temps raisonnable » Wikipedia.

1) Génération de la population

La première étape d’un algorithme génétique est la génération de la population. Un individu de la population consiste dans notre cas en une suite de six actions. Afin de ne pas traiter le 11 millions de combinaisons possibles, nous utilisons des groupes de deux actions appelés « gènes », que nous disposons à la suite pour composer l’individu.

La sélection des gènes et leur disposition est faite de manière aléatoire.

2) Évaluation

L'évaluation consiste à donner une note ou « fitness score » à chaque individu. Ce score prend en compte le contexte général de la partie et donc l'état des joueurs adverses. Pour simuler la manche, nous donnons aux joueurs adverses le comportement de l'IA heuristique.

Pour chaque évaluation le moteur de jeu simule la manche et calcul le score de l'individu. A la fin de chaque évaluation une fonction rollback est appelée pour revenir à l'état précédent et tester un nouvel individu.

3) Sélection

Cette étape consiste à trier la population. Dans notre cas nous ne gardons que les trois meilleurs individus.

4) Croisement

Cette étape consiste à croiser les gènes des individus pour en obtenir de meilleurs. Nous conservons systématiquement le meilleur et nous croisons le meilleur avec le deuxième et le troisième.

Un croisement consiste en la sélection de gènes de manière aléatoire sur le meilleur individu pour les placer sur un individu moins bon.

5) Mutation

Cette étape consiste en la modification d'un gène de manière aléatoire pour espérer obtenir un individu plus performant.

Cette suite d'étape est bouclée afin d'avoir le meilleur individu possible.

5.2 Conception logiciel

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

Classe IA : Les classes filles de la classe IA implémentent différentes stratégies d'IA. Elles possèdent un camp et une fonction « run ».
Classe RandomIA : Classe qui implémente l'IA aléatoire.
Classe HeuriticIA : Classe qui implémente l'IA heuristique. La fonction pathToNearestCp définit le chemin vers le checkpoint le plus proche, et l'IA choisi entre rallier les checkpoint ou attaquer les joueurs adverses si leur probabilité de gagner est élevée.
Classe Graph : Un Graph est une Position qui possède un prédécesseur. Des instances de Graph sont utilisées dans la recherche d'un chemin.
Classe DeepIA : Classe qui implémente l'IA avancée. Les déplacements sont définis comme pour l'IA Heuristique. L'algorithme génétique permet au bout de quelques itérations de trouver une liste d'actions à effectuer.

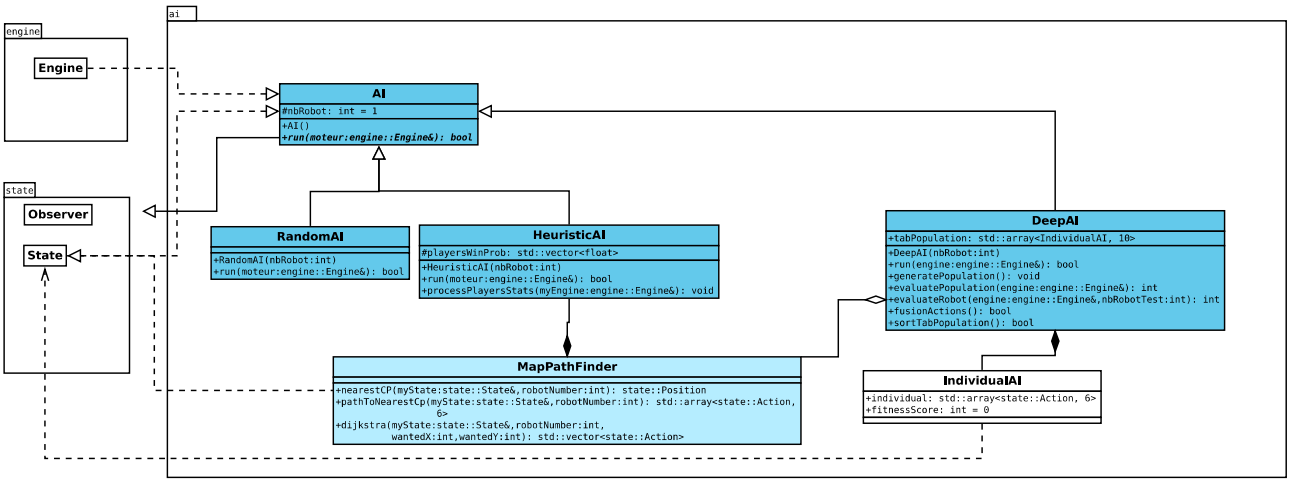

FIGURE 9 – Diagramme de classes du moteur d’intelligence artificielle

6 Modularisation

6.1 Organisation des modules

6.1.1 Répartition sur différents threads

L'objectif est ici est de placer le moteur de jeu sur un thread et le moteur de rendu sur un autre thread. Le moteur de rendu est nécessairement sur le thread principale (contraire matérielle), et le moteur du jeu est sur un thread secondaire. Nous avons deux type d'information qui transite d'un module à l'autre : les commandes et les notifications de rendu.

Les commandes peuvent arriver à n'importe quel moment, y compris lorsque l'état du jeu est mis à jour. Pour résoudre ce problème, nous proposons d'utiliser un double tampon de commandes. L'un contiendra les commandes actuellement traitées par une mise à jour de l'état du jeu, et l'autre accueillera les nouvelles commandes. A chaque nouvelle mise à jour de l'état du jeu, on copie les commandes d'un tampon à l'autre. Le temps de la copie étant négligeable (quelques nanosecondes), le tampon de réception est toujours capable d'être enrichi, et cela sans aucun blocage. Il en résulte une parfaite répartition des traitements, ainsi qu'une latence au plus égale au temps entre deux époques de jeu.

Notifications de rendu. Ce cas est problématique, car il n'est pas raisonnable d'adopter une approche par double tampon. En effet, cela implique un doublement de l'état du jeu (un en cours de rendu, l'autre en cours de mise à jour), ce qui augmente de manière significative l'utilisation mémoire et processeurs, ainsi que la latence du jeu. Nous nous sommes donc tournés donc vers une solution avec recouvrement entre les deux modules, en proposant une approche qui le minimise autant que possible. Pour ce faire, nous ajoutons un cache qui va « absorber » toutes les notifications de changement qu'émet une mise à jour de l'état du jeu. Puis, lorsqu'une mise à jour est terminée, le moteur de jeu envoie un signal au moteur de rendu. Celui-ci, lorsqu'il s'apprête à envoyer ses données à la carte graphique, regarde si ce signal a été émis. Si c'est le cas, il vide le tampon de notification pour modifier ses données graphiques avant d'effectuer ses tâches habituelles. Lors de cette étape, une mise à jour de l'état du jeu ne peut avoir lieu, puisque le moteur de rendu a besoin des données de de l'état pour mettre à jour les siennes. Nous avons donc ici un recouvrement entre les deux processus. Cependant, la quantité de mémoire et de processeur utilisée est très faible devant celles utilisées par la mise à jour de l'état du jeu et par le rendu.

6.1.2 Répartition sur différentes machines

6.2 Conception logiciel

6.3 Conception logiciel : extension réseau