

Projet Worms 3IS 2017 - 2018

ABOUOBAYD Sanaa
PHENIX Clara

Professeur encadrant :
Philippe-Henri GOSELIN



Figure 1 - Exemple du jeu Worms

Sommaire

1. Présentation générale	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 Etat éléments fixes :	5
2.1.2 Etat éléments mobiles :	6
2.1.3 Etat général :	6
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	11
Ces changements s'appliquent à chaque création ou mise à jour d'un état, après les changements extérieurs.	11
4.3 Conception logiciel	12
5. Intelligence artificielle	14

1. Présentation générale

1.1 Archétype

L'objectif de ce projet est la réalisation du jeu Worms en 2D, avec les règles les plus simples. Un exemple est présenté en Figure 1.

1.2 Règles du jeu

Le jeu se déroule dans un unique monde en 2D et oppose deux équipes, une équipe verte et une équipe noire, composées chacune de trois personnages. Chacun son tour, les deux joueurs choisissent un de leurs personnages pour tirer sur ses adversaires. Un seul type d'armes est mis à disposition pour chaque personnage. Le joueur n'a droit qu'à un seul tir et à trois pas par tour. Un personnage meurt s'il est touché trois fois. Le premier joueur qui perd tous ses personnages a perdu.

1.3 Ressources



By
Thérapie

Figure 2 - Sprites pour les personnages

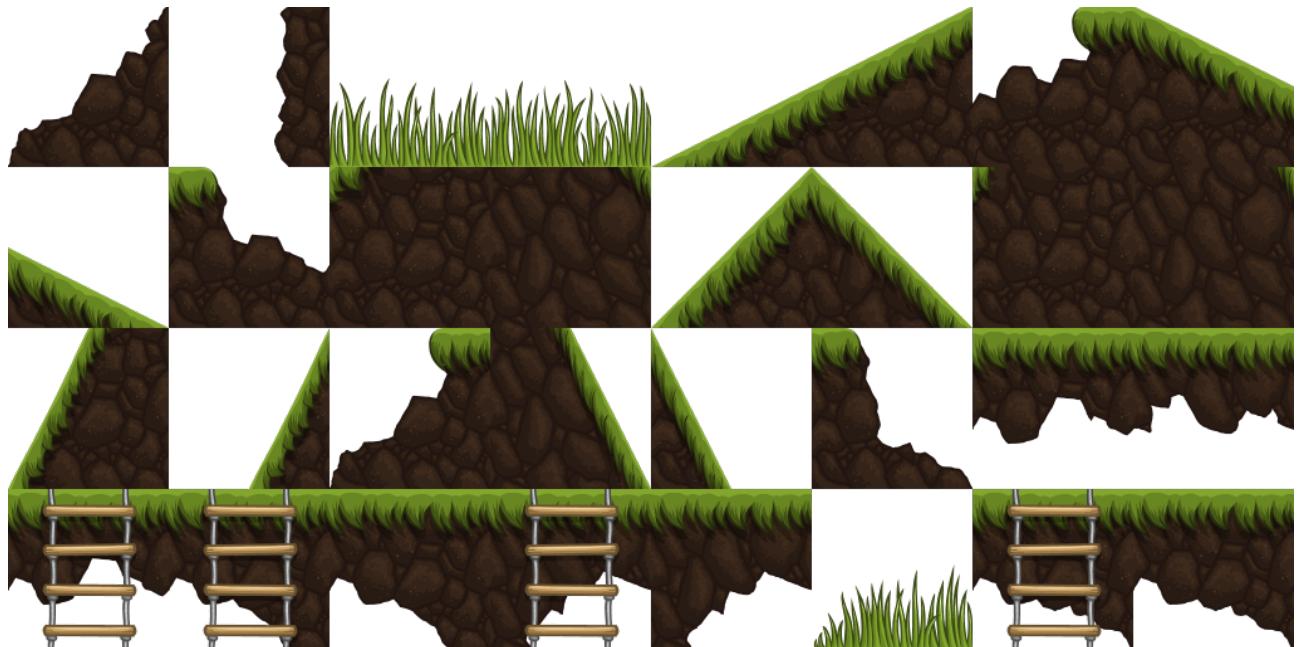


Figure 3 - Sprites pour la carte



Figure 4 - Sprites pour le décor

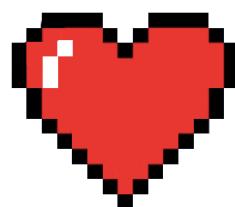


Figure 5 - Sprite pour les vies

2. Description et conception des états

2.1 Description des états

Un état du jeu est composé d'un ensemble d'éléments fixes (les cases), et des éléments mobiles (les personnages).

Les propriétés suivantes sont communes à tous les éléments:

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

2.1.1 Etat éléments fixes :

La carte du jeu est composée de "cases". Ces cases sont elles aussi divisées en deux types :

- **Cases Sol** : ce sont les éléments infranchissables par les éléments mobiles et sur lesquelles les personnages peuvent se déplacer. Il est possible de les empiler pour rajouter du relief à la carte. Cet élément peut changer d'état au cours de la partie et devenir une case vide (définie plus loin). Le choix des textures qui constituent ces cases est purement esthétique et n'influence pas sur le déroulement du jeu.
- **Cases Espace** : ce sont les éléments franchissables par les éléments mobiles. On différencie deux types d'espace :
 - Les espaces *vides*
 - Les espaces *vies*, contiennent les vies accessibles pour les personnages et permettant le gain d'une vie pour le personnage qui le récupère.

2.1.2 Etat éléments mobiles :

Les éléments mobiles possède une direction (aucune, gauche, droite ou haut).

- **Personnage** : Cet élément est dirigé par le joueur, qui commande les propriétés de déplacement. Chaque personnage dispose d'un compteur de pas, qui limitera ses déplacements pendant un tour. Un personnage peut avoir deux états :
 - **Etat vivant** : C'est l'état normal, il peut tirer et se déplacer dès qu'il est sélectionné par le joueur.
 - **Etat mort** : Le personnage n'est plus visible à l'écran, lorsqu'il a perdu ses trois vies.

2.1.3 Etat général :

En plus des éléments décrits précédemment, le jeu possède les propriétés générales suivantes :

- **Epoque** : représente l'heure correspondant à l'état, ie le nombre de tic de l'horloge depuis le début de la partie.
- **Vitesse** : c'est la vitesse à laquelle l'état du jeu sera mis à jour.
- **Compteur de vies** : permet de compter le nombre de vies restantes que possède un personnage.
- **Compteur de parties gagnées** : le nombre de parties remportées par l'un des deux joueurs.

2.2 Conception logiciel

Nous avons représenté en figure 6 le diagramme UML, correspondant à la description des états.

Il en ressort deux groupes de classes principaux :

- Le groupe de classe Element. Ce groupe est construit à partir du polymorphisme. La classe abstraite Element est à la tête de cette hiérarchie. Toutes les classes décrivant un élément du jeu hérite de cette classe. Nous avons classé ces éléments en deux catégories : les éléments statiques et les éléments mobiles. Afin de les reconnaître nous avons donc créé une méthode `isStatic()`. En se basant sur le même principe, nous avons créé la méthode `isSpace()` pour la classe `StaticElement`.

- Le groupe de classe dédié au placement des éléments. Pour pouvoir situer ces éléments dans l'espace, nous avons réalisé la classe ElementTab qui représente une grille en deux dimensions. Chaque « Element » appartient à la grille ElementTab. La classe State permet d'obtenir les informations concernant l'état.

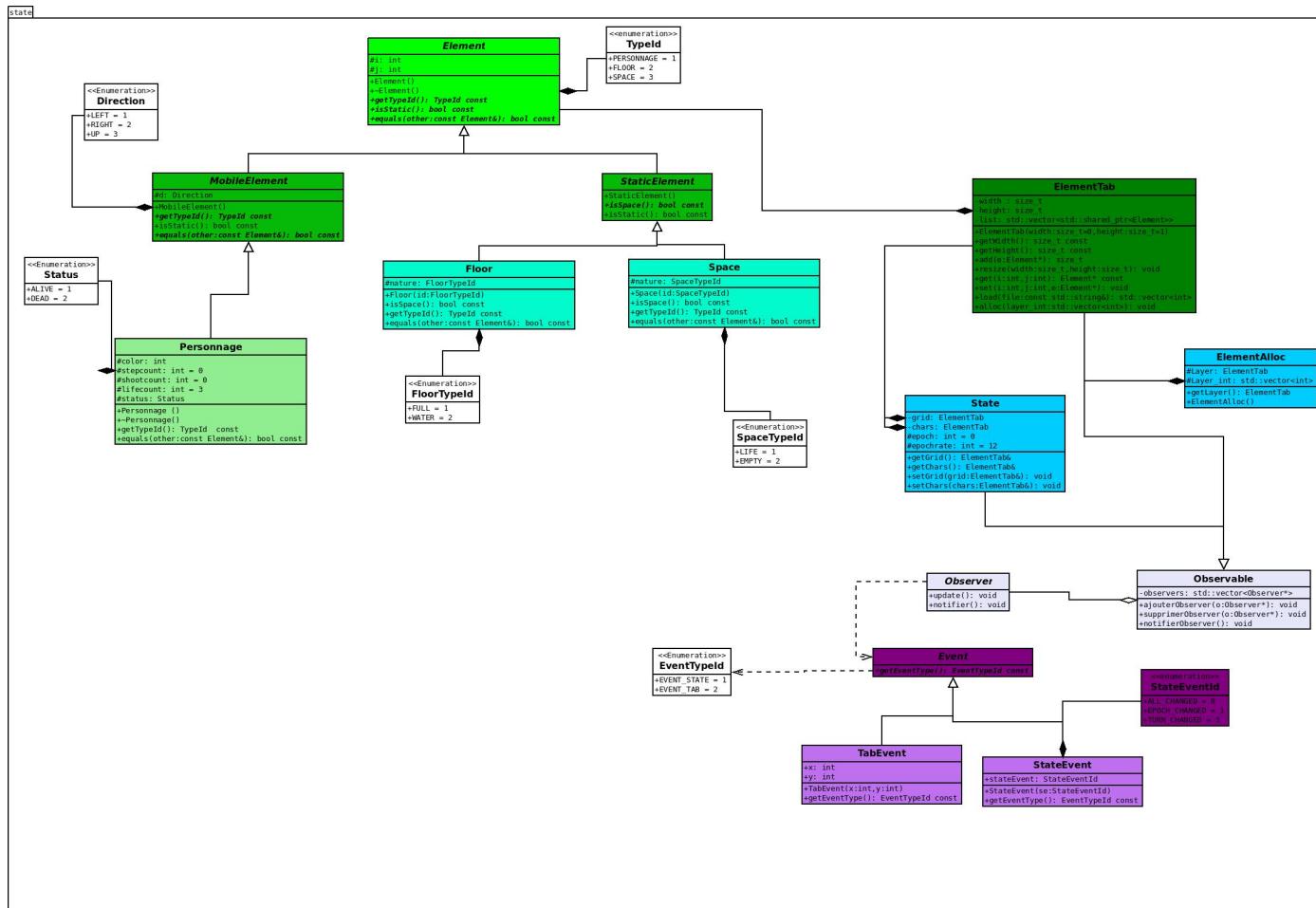


Figure 6 - Diagramme UML state

3. Rendu : Stratégie et Conception

3.1 Stratégie de rendu d'un état

Nous avons choisi séparer notre carte en plusieurs couches (layers) : une couche background, une couche mer, une couche terre et par la suite une couche personnage et une couche information/bonus. Chaque couche correspond à un fichier .txt qui sera lu et qui permettra d'afficher la carte.

Chaque élément du fichier correspond à la position de la tiled équivalente, dans le sprite. Grâce à cette information on va pouvoir associer chaque élément à un sprite et ainsi pouvoir l'afficher graphiquement.

3.2 Conception logiciel

Plans : La classe Layer sert à donner des informations 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.

Surface : Une surface contient une texture du plan et une liste de paire de paires de quadruplets de vecteurs 2D. Chaque quadruplet possède des éléments texCoords, qui correspondent aux coordonnées des quatre coins de la tuile à sélectionner dans la texture, et des éléments position, qui correspondent aux coordonnées des quatre coins du carré où doit être dessiné la tuile à l'écran.

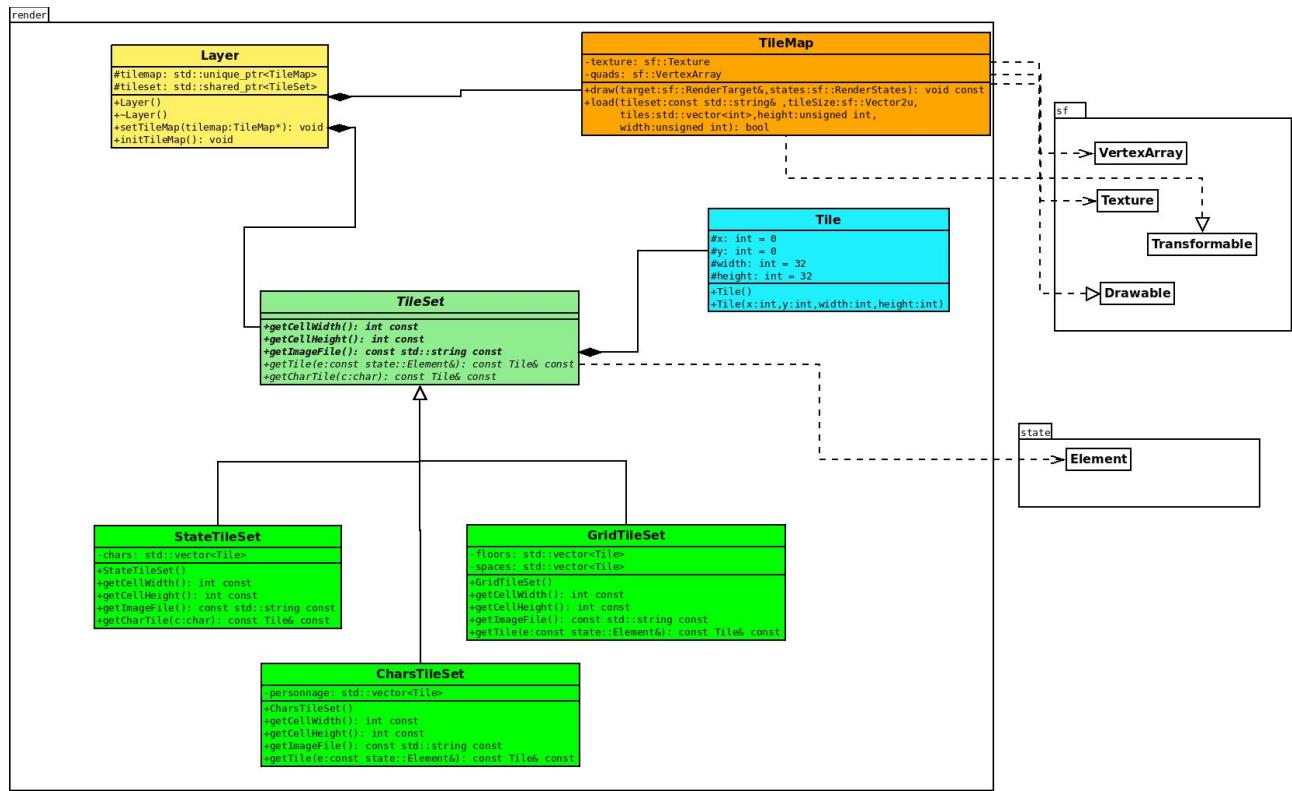


Figure 7 - Diagramme UML render

4. Règles de changement d'états et moteur de jeu

4.1 Changements extérieurs

Les changements extérieurs sont provoqués par des commandes extérieures, comme la pression sur une touche (sélectionnée par le joueur) ou un ordre provenant du réseau :

- Démarrer une nouvelle partie : On fabrique un état initial à partir d'un fichier
- Sélectionner le personnage qu'on souhaite déplacer ou avec lequel on va tirer sur l'équipe adverse
- Déplacer personnage, paramètres « personnage », « direction » : Un personnage se déplace toujours selon orientation, mais ne dépasse pas ses nombres de pas limités.
- Tir : Un personnage peut tirer sur l'équipe adverse, mais ne peut pas excéder un tir par tour.

4.2 Changements autonomes

Ces changements s'appliquent à chaque création ou mise à jour d'un état, après les changements extérieurs.

- Au début de chaque tour, l'un des deux joueurs choisit l'un de ses trois personnages avec lequel il souhaite jouer.
- Appliquer les règles de mouvement et de tir pour le personnage
- Quand un personnage se déplace, on incrémente de compteur de pas.
- Quand le compteur de pas du personnage arrive à 3, le personnage ne peut plus se déplacer.
- Quand un personnage tire, le compteur de tir passe à 1. Il ne peut plus tirer par la suite et le tour passe à l'autre joueur.
- Si une case "Floor" est touchée par un tir, elle devient une case "Space"
- Quand un personnage est touché par un tir, son compteur de vie est décrémenté.
- Si le compteur de vies d'un personnage est à 0, alors le personnage obtient le statut "DEAD"
- Quand un personnage se trouve sur une case "LIFE", il gagne une vie. On incrémente son compteur de vie de 1.
- Si un personnage se trouve sur une case "SPACE", il meurt. Il obtient alors le statut « DEAD ».

4.3 Conception logiciel

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, clavier, réseau, ...). On définit un type de commande avec `CommandTypeid` pour identifier précisément la classe d'une instance.

- *OrientationCommand* : Modifie l'orientation d'un personnage ;
- *MoveCharCommand*: Déplace un personnage en fonction de son orientation;
- *HandlelifesCommand* : Regarde si un personnage est sur une vie, et en fonction des cas applique les changements nécessaire.
- *HandleImpactCommand* : Regarde si le tir du personnage touche une case "Space", "Floor" ou "Personnage", et en fonction des cas, applique les changements nécessaires.
- *ShootCommand* : *Commande de tir vers un ennemi.*
- *JumpCommand* : *Commande de saut pour un personnage.*
- *LoadCommand*: *Charge l'état initial du jeu.*

Engine: Cette classe stocke les commandes dans une `std::map` avec clé entière.

Ce mode de stockage permet d'introduire une notion de priorité : on traite les commandes dans l'ordre de leur clés, de la plus petite à la plus grande. Lorsqu'une nouvelle époque démarre, ie lorsqu'on a appelé la méthode `update()` après un temps suffisant, le moteur appelle la méthode `moteur()` de chaque commande, incrémenté l'époque, puis supprime toutes les commandes.

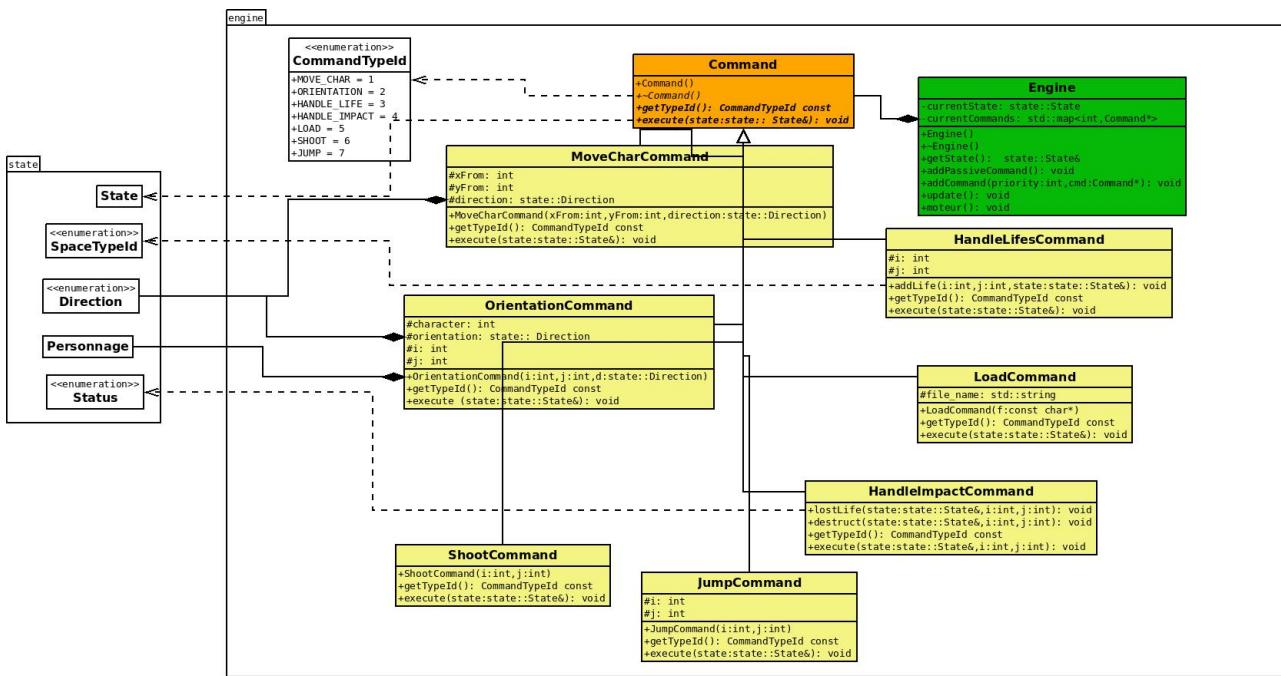


Figure 8 - Diagramme UML engine

5. Intelligence artificielle

5.1 Stratégies

5.1.1 Intelligence aléatoire

On crée toutes les commandes possibles en fonction de la position du personnage. On stocke ces commandes dans une liste. Ensuite l'IA choisit aléatoirement une commande et l'exécute.

5.1.2 Intelligence basée sur des heuristiques

Nous proposons ensuite un ensemble d'heuristiques pour offrir un comportement meilleur que le hasard, et avec une chance notable de résoudre le problème complet (ie, ne pas être le premier à perdre ses trois vies) :

- Si le personnage en action a moins de trois vies on s'approche d'une case vie pour en récupérer une.
- Sinon, si on est loin d'un ennemi on se dirige vers lui pour pouvoir être assez près pour lui tirer dessus. Si on est proche on s'éloigne de cet ennemi, d'une distance qui permet quand même de lui tirer dessus.

La plupart des heuristiques proposées sont mis en œuvre en utilisant des cartes de distance vers un ou plusieurs objectifs.

5.2 Conception logiciel

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

Classes AI: La méthode listCommands permet de remplir une liste de commandes.

- RandomAI : choisit aléatoirement une commande et l'exécute.
- HeuristicAI: choisit la commande qui lui paraît la plus judicieuse en fonction de l'état du personnage.

PathMap: La classe PathMap permet de calculer une carte des distances à un ou plusieurs objectifs. Plus précisément, pour chaque case « espace » du niveau, on peut demander un poids qui représente la distance à ces objectifs. Pour s'approcher d'un objectif lorsqu'on est sur une case, il suffit de choisir la case adjacente qui a un plus petit poids, si l'on veut s'approcher de l'objectif. Si l'on veut s'éloigner d'un objectif (ennemi) il faudra choisir la case adjacente qui a un plus grand poids.

- lifeMap : Objectif cases vie, pour pouvoir récolter une vie
- ennemyMap : Objectif ennemis (normales ou super), pour s'en approcher pour pouvoir tirer dessus, ou bien s'en éloigner si on a très peu de vies.

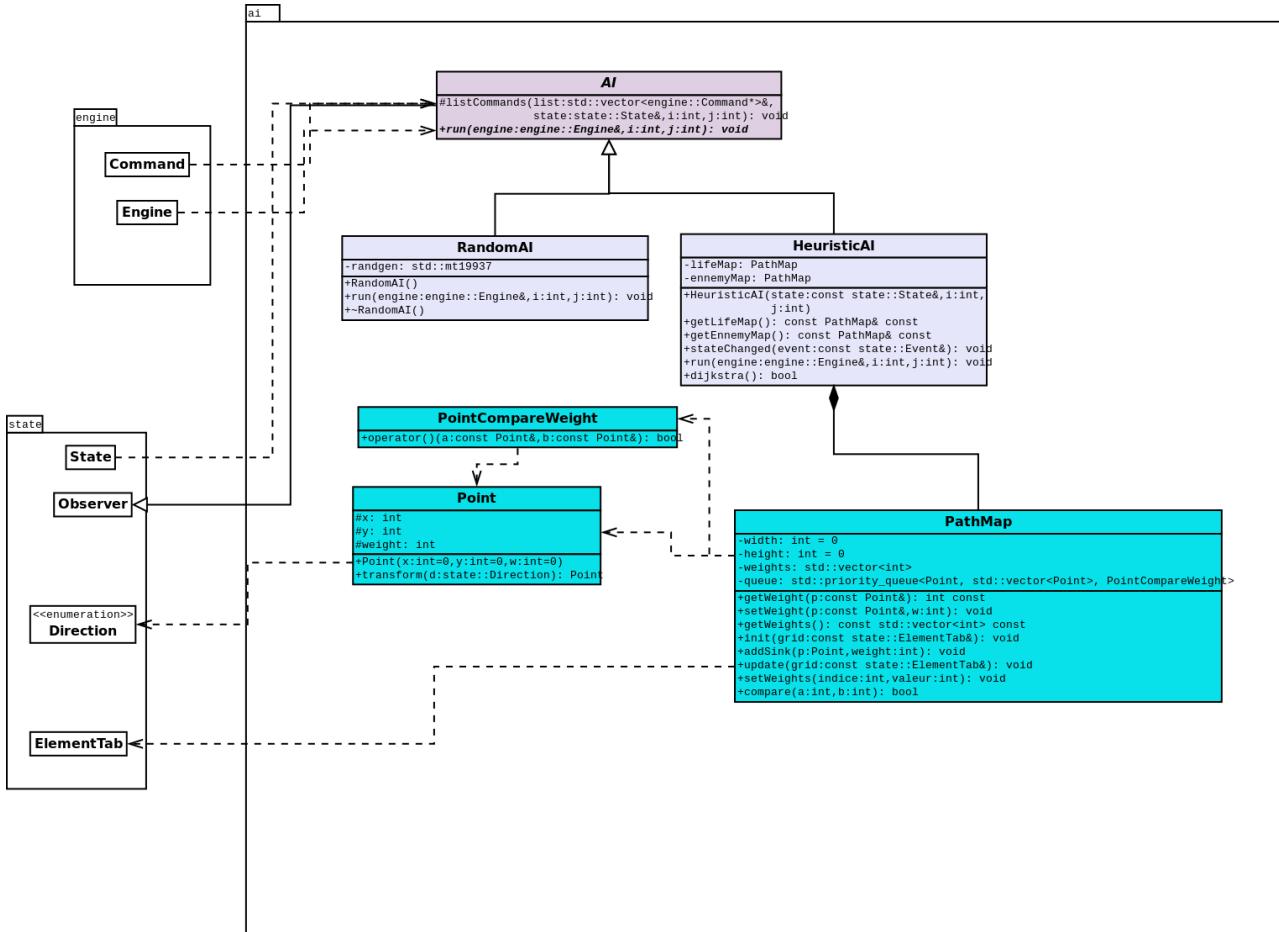


Figure 9 - Diagramme UML ai

