

## 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	7
3. Rendu : Stratégie et Conception	9
3.1 Stratégie de rendu d'un état	9
3.2 Conception logiciel	10
4. Règles de changement d'états et moteur de jeu	12
4.1 Changements extérieurs	12
4.2 Changements autonomes	13
Ces changements s'appliquent à chaque création ou mise à jour d'un état, après les changements extérieurs.	13
4.3 Conception logiciel	14
5. Intelligence artificielle	17
6. Modularisation	21
6.1.1 Répartition sur différents threads	21
6.1.2 Répartition sur différentes machines : rassemblement des joueurs	21
6.1.3 Répartition sur différentes machines : échange des commandes	23
6.1.4 Répartition sur différentes machines : échange des commandes	23

## 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 personnages, un personnage vert et un personnage noir. Chacun son tour, les deux joueurs tirent sur le personnage de son adversaire. 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 dont le personnage meurt a perdu.

### 1.3 Ressources



Figure 2 - Sprites pour les personnages

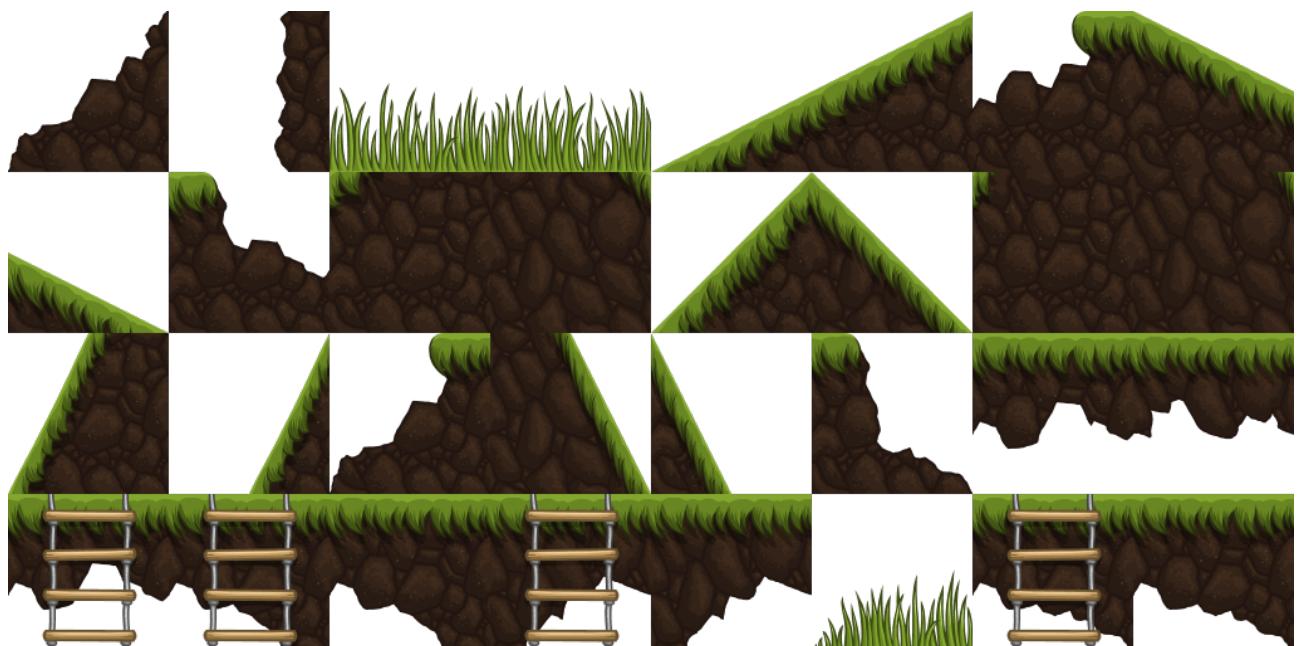


Figure 3 - Sprites pour la carte

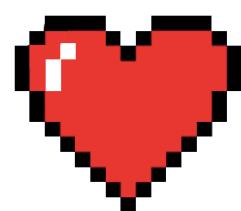


Figure 5 - Sprite pour les  
vies  
4 sur 25

## 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.

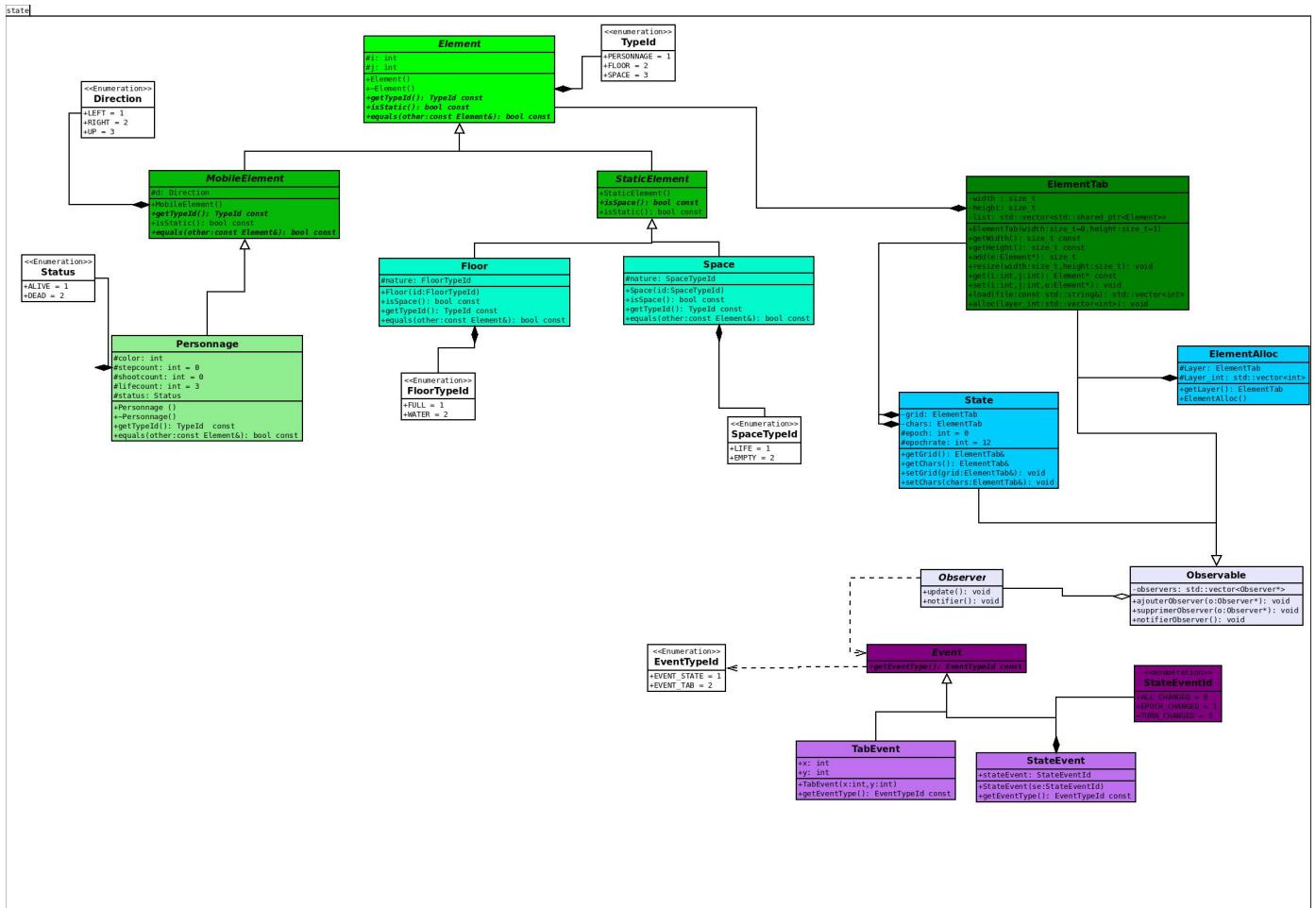
Il en ressort deux groupes de classes principaux :

- 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.



Ecole Nationale  
Supérieure  
de l'Électronique  
et ses Applications

ABOUOBAYD Sanaa  
PHENIX Clara  
Projet 3IS



## 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.

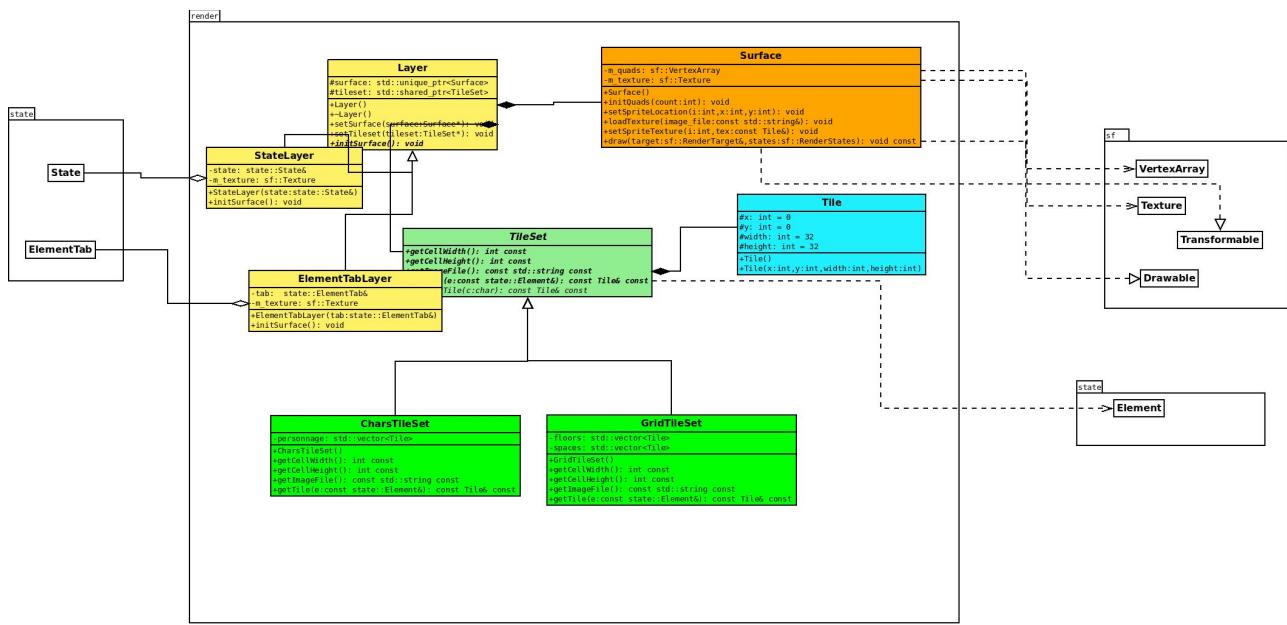


Figure 7 - Diagramme UML render

## 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
- 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 le personnage 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.

- Appliquer les règles de mouvement et de tir pour le personnage
- Quand un personnage se déplace, on incrémente le 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.
- 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 `CommandType` pour identifier précisément la classe d'une instance. Dans chaque commande nous avons créé une méthode `serialize` qui permet de placer les caractéristiques de cette commande (type et attribut) dans un JSON value pour pouvoir ensuite l'enregistrer dans un fichier texte. La méthode `deserialize` permet de récupérer ces caractéristiques depuis le fichier texte, de les placer dans un JSON value et ensuite d'exécuter la commande correspondante.

- *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 nécessaires.
- *ShootCommand* : *Commande de tir vers un ennemi.*
- *JumpCommand* : *Commande de saut pour un personnage.*
- *LoadCommand*: *Charge l'état initial du jeu.*

**Classes Action.** Au sein de nos commandes, différentes actions sont effectuées.

- Au sein de *MoveCharCommand*, on déplace un personnage et suivant où il va, il peut gagner une vie ou bien mourir si il tombe dans l'eau. On a donc trois actions possibles : bouger, récupérer une vie ou mourir.
- Au sein de *ShootCommand* on fait tirer le personnage sur un ennemi. Mais si l'ennemi perd ainsi sa dernière vie, il meurt. On donc ici deux actions tirer et mourir.

Nous avons donc créé pour chaque action une classe qui hérite de la classe mère `Action`. Dans chacune de ces classes, nous avons défini deux méthodes: `apply` et `undo`. `Apply` est appelée dans les classes commandes correspondantes au moment

des modifications de l'état, car ces modifications sont faites dans cette méthode. La méthode undo correspond à la méthode apply mais inversée elle permet de revenir à l'état précédent l'appel à apply.

**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, inclémentement l'époque, puis supprime toutes les commandes. La méthode update() renvoie une pile d'Actions. A chaque fois qu'une méthode apply() est appelée on ajoute l'action correspondante à la pile d'action, prise en argument par la méthode execute() de chaque commande.

Ainsi lorsque l'on veut faire un rollback, on appelle la méthode undo() de la classe Engine qui appelle la méthode undo() de toutes les actions se situant dans la pile en commençant ainsi par la dernière ajoutée.

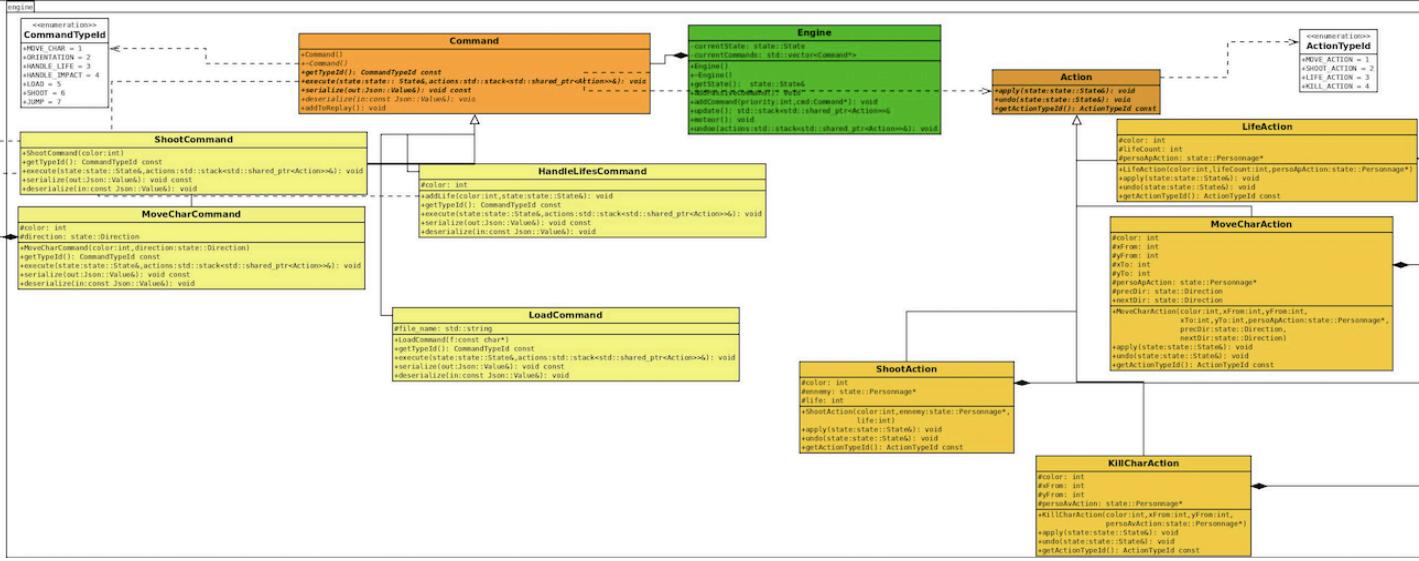


Figure 8 - Diagramme UML engine

## 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.

### 5.1.3 Intelligence avancée

Nous proposons une intelligence plus avancée, basée sur les arbres de recherche. On peut alors représenter le jeu selon un graphe. Un état est un état du jeu à une époque donnée. Les arcs entre les sommets du graphe d'état sont les changements d'états@@. 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,...). L'évaluation(score d'un sommet/état du jeu est déterminé par le nombre pastille restantes si Pacman est vivant. Un sommet/état du jeu avec Pacman mort a un score infini, et n'a qu'un seul arc (sommet/état juste avant la mort de Pacman). Le meilleur choix de mouvement pour Pacman est donc celui du plus court chemin dans le graphe d'état qui mène vers un score nul (toutes les pastilles ont été mangées). 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 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.

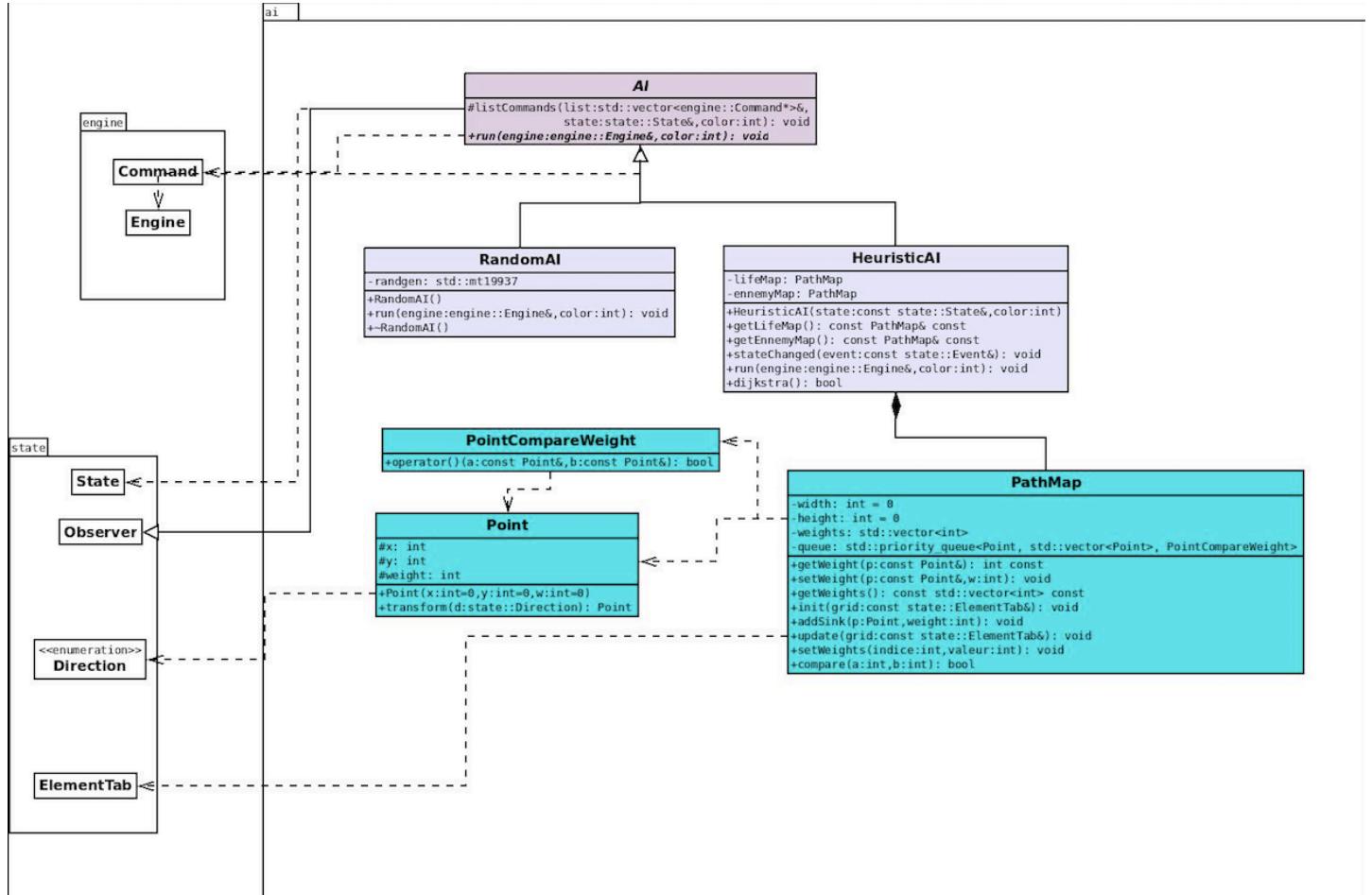


Figure 9 - Diagramme UML ai

## 6. Modularisation

### 6.1 Organisation des modules

#### 6.1.1 Répartition sur différents threads

Dans cette partie, nous avons placé l'exécution de l'heuristique dans un thread. Afin que le rendu ne se lance pas alors que l'état ne soit pas totalement chargé, nous avons utilisé un mutex pour pouvoir bloquer l'exécution du rendu pendant le chargement de l'état, puis de bloquer le changement de l'état, pendant l'exécution du rendu.

#### 6.1.2 Répartition sur différentes machines : rassemblement des joueurs

Le but est de créer une liste de client pour le serveur, afin de jouer en réseau. Pour ce faire, nous formons des services CRUD sur la donnée "player" via une API Web REST :

Requête GET/player/id	
Pas de données en entrée	
Cas <id> négatif (pour consulter la liste des joueurs présents)	Status OK Données sortie : <pre>Type : "array", Items : {     type: "object",     properties: {         "color": { type: string },         "name": { type: string },     }, }, required: [ "color", "name" ]</pre>

Cas player <id> existe (pour consulter les propriétés du joueur <id>)	Status OK Données sortie :  <pre>type: "object", properties: {     "color": { type: string },     "name": { type: string }, }, required: [ "color", "name" ]</pre>
Cas player <id> n'existe pas	Status NOT_FOUND Pas de données de sortie

### Requête POST/player/id

Données en entrée :

```
type: "object",
properties: {
    "color": { type: string },
},
required: [ "color" ]
```

Cas player <id> existe	Status NO_CONTENT Pas de données de sortie
Cas player <id> n'existe pas	Status NOT_FOUND Pas de données de sortie

### Requête PUT/player

Données en entrée :

```
type: "object",
properties: {
    "color": { type: string },
},
required: [ "color" ]
```

Cas il reste une place libre	Status CREATED Données sortie :  <pre>type: "object", properties: {     "id": { type: number, minimum: 0, maximum: 1 }, }, required: [ "id" ]</pre>
Cas plus de place libre	Status OUT_OF_RESOURCES Pas de données de sortie

### 6.1.3 Répartition sur différentes machines : échange des commandes

Nous utilisons un service web CR sur la donnée "commandes par époque" : on peut créer une commande ou un lot de commandes par époques, et consulter les commandes des époques passées.

Les clients envoient leurs commandes moteur au serveur, mais ne les exécutent pas directement. En même temps, il demandent au serveur de manière régulière s'il y a des commandes. Si c'est le cas, les clients exécutent ces commandes, puis attendent à nouveau les prochaines.

Requête GET/command/epoch	
Pas de données en entrée	
Cas <epoch> négatif (pour consulter les commandes des époques précédentes)	Status OK Données sortie : liste des commandes sérialisées en JSON
Cas <epoch> existe	Status OK Données sortie : liste des commandes sérialisées en JSON correspondante à l'époque <epoch>
Cas <epoch> n'existe pas	Status NOT_FOUND Pas de données de sortie

Requête PUT/command	
Données en entrée :	
<pre>type: "object", properties: {     "commande": commande sérialisée }, required: [ "command" ]</pre>	<p>Pour tous les cas</p> <p>Status CREATED Données sortie :</p> <pre>type: "object", properties: {     "epoch": { type: number}, }, required: [ "epoch" ]</pre>

### 6.1.4 Répartition sur différentes machines : échange des commandes

Nous avons créé un service web qui permet d'obtenir le statut de la partie : Lorsque l'état est "CREATING", la partie est entrain de se créer et attend les joueurs. Dès que deux joueurs rejoignent la partie, l'état passe à "RUNNING" et la partie peut commencer.

## 6.2 Conception logiciel

Le diagramme des classes pour la modularisation est représenté en figure 10.

**Game.** La classe Game et les classes associées représentent les joueurs qui seront présents dans la partie. Elle contient également un moteur de jeu, exécuté dans un thread séparé (le thread principal est réservé au HTTP). Une fois la partie démarrée, le moteur de jeu tourne en boucle et le client demande des notifications de commandes. Un système d'enregistrement mémorise les commandes en JSON.

**EngineServer.** Cette classe est une classe fille d'Engine. Elle permet d'ajouter des commandes au serveur.

**Services :** Les classes filles de la classe AbstractServices servent à implanter les services REST, et la classe ServiceManager va gérer ces services :

- VersionService : renvoie la version actuelle de l'API. Il est fortement recommandé d'avoir ce traditionnel service pour pouvoir prévenir les conflits de version.
- PlayerService : contient les services CRUD qui vont permettre de consulter, ajouter, modifier, ou supprimer des joueurs.
- CommandService : fournit les services CR pour la ressource "commandes par époque". Permet d'ajouter, et consulter ces lots de commandes.
- GameService : fournit un unique service qui permet de consulter l'état du jeu. Lorsque l'état est "CREATING", la partie est entrain de se créer et attend les joueurs. Dès que deux joueurs rejoignent la partie, l'état passe à "RUNNING" et la partie peut commencer.

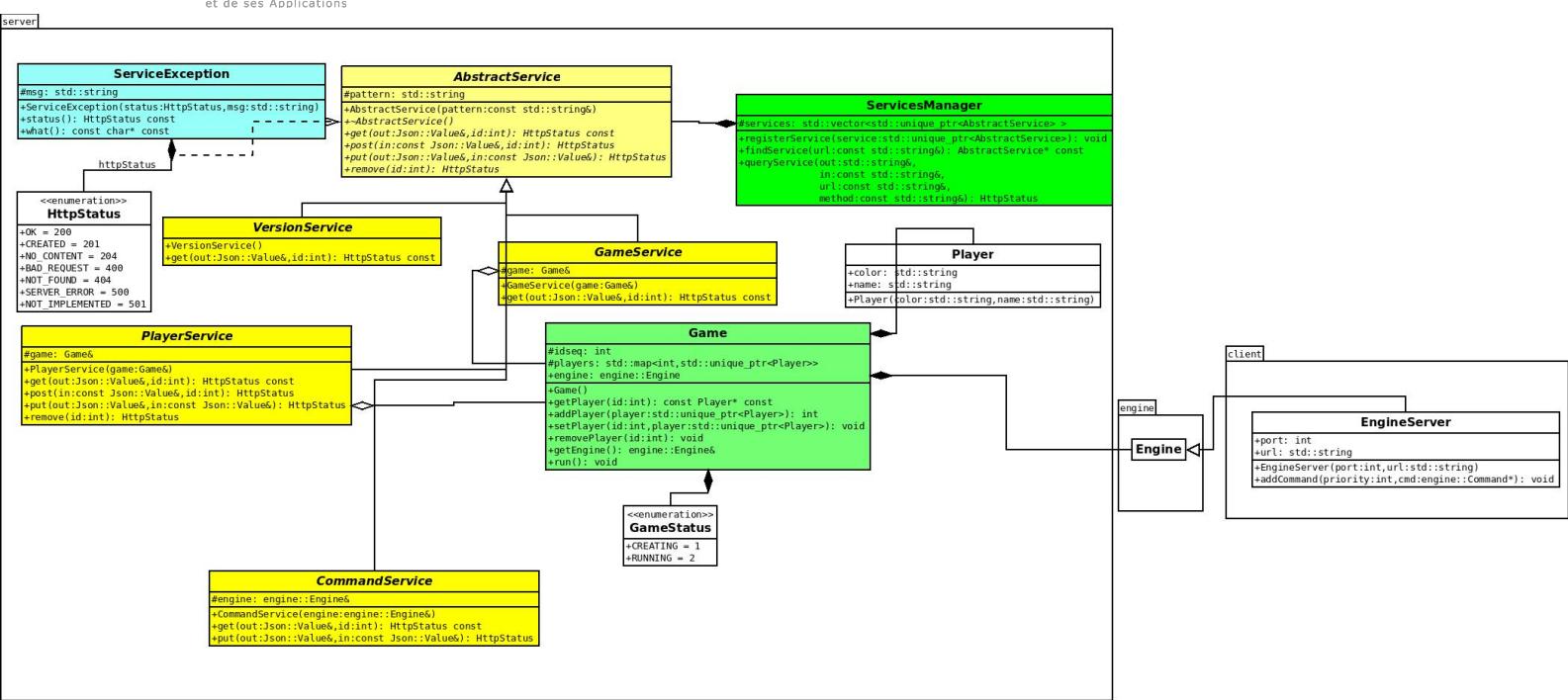


Figure 10 - Diagramme UML modularisation