PROJET LOGICIEL TRANSVERSAL:
« CotCot WarFarm »

Jimmy HENNION - Antoine NOURRISSON

Table des matières

1/ Description du jeu

1.1/ Présentation Générale

CotCot WarFarm est un jeu s'inspirant de Worms Armaggedon (version 2D). Au lieu d'assister à une guerre entre des vers de terre, ce sont des poules qui s'affrontent par équipe. Elles peuvent, tout comme dans Worms, utiliser diverses armes telles que les grenades, les roquettes, les pistolets,…

Illustration n°1: Le jeu Worms

Il y a toujours la notion de gravité terrestre sans inclure la force du vent.

Il y aura en tout une capacité de huit poules au maximum sur le terrain, donc il peut y avoir jusqu'à huit joueurs sur une partie contrôlant chacun une seule poule, ou bien deux joueurs contrôlant chacun 4 poules. Il est possible de rajouter des I.A., toujours jusqu'à un maximum de huit poules sur le terrain.

1.2/ Règles du Jeu

Chaque joueur contrôle une équipe de poules. Le but de chaque joueur est de tuer toutes les poules des joueurs adverses. Le gagnant est celui qui a au moins une poule encore en vie tandis que tout les poules adverses sont mortes.

Le déroulement se fait au tour par tour. Les actions possibles à chaque tour sont: se déplacer, sauter, tirer/frapper avec une seule arme… Chaque tour permet à une poule de se déplacer sur 300 unités de mouvements au maximum (1 unité de mouvement=2pixels), ou alors s'arrête lorsque le joueur a tiré/frappé avec son arme. Le décor est indestructible.

Les poules meurent lorsqu'elles sont touchées par une poule ennemie.

1.3/ Textures Employées

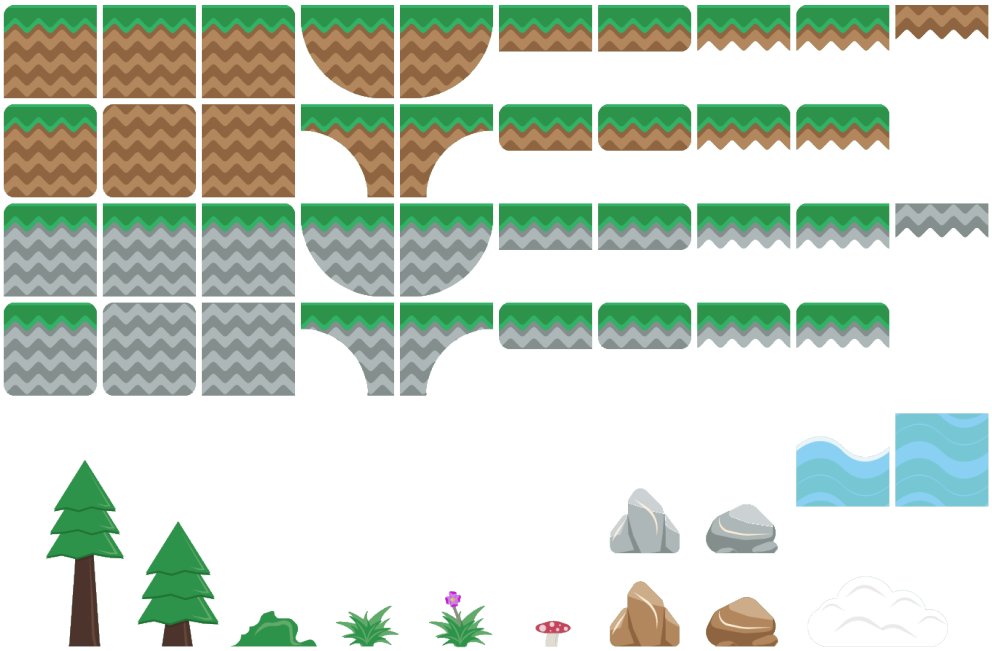

Illustration n°2 : Textures des terrains

Illustration n°3 : Texture des personnages

Comme il est possible de le remarquer, il y a différentes couleurs de poules, et ces couleurs permettent de distinguer l'équipe. Il y a donc une couleur par équipe. De plus, certaines textures sur cette image ne sont pas utilisées, telles que les derrières des poules…

Illustration n°4 : Textures des armes

Ici toute une panoplie d'armes se propose pour équiper les poules, cependant seulement une petite portion d'entre elles seront utilisées. Pour le moment seule l'épée est utilisable.

2/ Description et conception des états

2.1/ Description des états

Un état du jeu est formé par un ensemble d'éléments fixes (l'environnement) et un ensemble d'éléments mobiles (les poules, les armes). Ces deux catégories sont regroupées sous une classe « Element ». Tous les éléments possèdent les propriétés suivantes :

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

2.1.1/ États d'éléments fixes

L'environnement est formée par une liste de cases d'éléments fixes nommée « background ». La taille de cette liste est fixée au démarrage du niveau. Les types de cases sont:

Cases « Terrain ». Les cases « Terrain » sont des éléments infranchissables pour les éléments mobiles. Le choix de la texture est purement esthétique, et n'a pas d'influence sur l'évolution du jeu.

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

- Les espaces « vides »
- Les espaces « eau », qui contiennent une texture eau et qui permettent de définir des lieux où les poules peuvent mourir.
- Les espaces « extra », qui contiennent une texture en arrière plan (pour faire joli !).
- Les espaces « départ », qui définissent les positions initiales possibles pour les poules.

2.1.2/ États d’éléments mobiles

Les éléments mobiles possèdent une direction (aucune, est, ouest) et une position (déterminée grâce aux coordonnées X et Y). Ils sont dirigés par un joueur ou une IA.

Éléments mobiles « Poule ». Cet élément est dirigé par commande de la propriété de direction (aucune, est et ouest). Ces éléments possèdent deux propriétés. D’une part, on utilise « la couleur » pour différencier les équipes de poules . D’autre part, on utilise une propriété que l’on nommera « FowlStatus », et qui peut prendre les valeurs suivantes :

- Status « Fowl_Alive » : cas le plus courant, la poule est en vie et peut agir.
- Status « Fowl_Hurt » : cas où la poule est blessée et perd des points de vie.
- Status « Fowl_Dead » : cas où la poule est morte et ne peut plus agir.
- Status « Fowl_Waiting » : cas où la poule ne joue pas et attend son tour.
- Status « Fowl_Hitting » : cas où la poule utilise une arme.

Éléments **mobiles « Arme ».** Cet élément est dirigé par commande de la propriété de direction (aucune, est et ouest). On utilise la propriété « WeaponStatus » qui prendra les valeurs suivantes :

- Status « Gun » : cas où l’arme utilise des munitions, comme un lance-roquette.
- Status « Melee » : cas où l’arme s’utilise au corps à corps, comme une épée.
- Status « Throwable » : cas où l’arme est une arme de jet, comme un shuriken.
- Status « Bomb » : cas où l’arme est une bombe qui se dépose.
- Status « Ammo » : cas où l’arme désignée est en réalité la munition d’une autre arme (pour gérer l’animation il faut les distinguer).

2.1.3/ État général

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

- Compteur MortSubite : compte le nombre de « tic » de l’horloge globale depuis le début de la partie. A un nombre choisi de « tic », nous passons dans l’état mort subite pour accélérer la fin de la partie. L’état mort subite réduit la vie de toutes les poules encore vivantes à 1 point et provoque la montée de l’eau.
- Compteur Tour : compte 30 secondes. Un tour dure au maximum 30 secondes.
- Compteur Poule : compte le nombre de poules vivantes par équipe.
- Compteur équipe : compte le nombre de points de vie par équipe.

2.2/ Conception Logiciel

Nous pouvons mettre en évidence les groupes de classes suivants (diagramme des classes présenté en Illustration n°5) :

Classe Éléments. Toute la hiérarchie des classes filles d'*Elements* permettent de représenter les différentes catégories et types d'éléments (en rouge et en orange sur le diagramme des classes). Pour faire de l'introspection, nous avons opter pour des méthodes comme *isStatic()* qui indiquent la classe d'un objet.

Fabrique d'éléments. Dans le but de pouvoir fabriquer facilement des instances d'Element, nous utilisons la classe *ElementFactory*. Cette classe, qui suit le patron de conception **Factory**, permet de créer n'importe quelle instance non abstraite à partir d'un caractère (classes vertes sur le diagrammes des classes). L'idée est de l'utiliser, en autres, pour créer un niveau à partir d'un fichier texte. Par exemple, chaque texture correspondant à un type de terrain est identifiée par une chaîne de caractères composée de chiffres (« 02 » identifie la texture représentant un carré de pierre aux angles droits et ayant de l'herbe, tandis que « 36 » identifie la tuile vide).

La classe *ElementFactory* est composée d'une interface *Ielement (en jaune sur le diagramme des classes)* qui va nous permettre d'accéder aux classes *FowlCreator*, *WeaponCreator*, *FieldCreator* et *SpaceCreator* qui serviront à fabriquer directement le type d'élément voulu.

Conteneur d'éléments. La classe State (en bleu foncé sur le diagramme des classes) est le conteneur d'élément mobiles et statiques, mais seuls les éléments mobiles seront amenés à changer d'état. Les éléments statiques étant indestructibles, ils n'auront pas à changer d'état.

2.3/ Conception Logiciel : extension pour le rendu

Observateurs de changements. Dans le diagramme de classes, nous présentons en roses les classes permettant à des tiers de réagir lorsqu'un événement se produit dans l'un des éléments d'état.
Les observateurs implantent l'interface IObserver pour être avertis des changements de propriétés d'état. Pour connaître la nature du changement, ils analysent l'instance de StateEvent. La conception de ces outils suit le patron Observer. C'est pourquoi on prend aussi les éléments statiques dans la classe State pour pouvoir les communiquer au rendu qui les affichera en permanence.

Voici le diagramme des classes pour représenter les états du jeu :

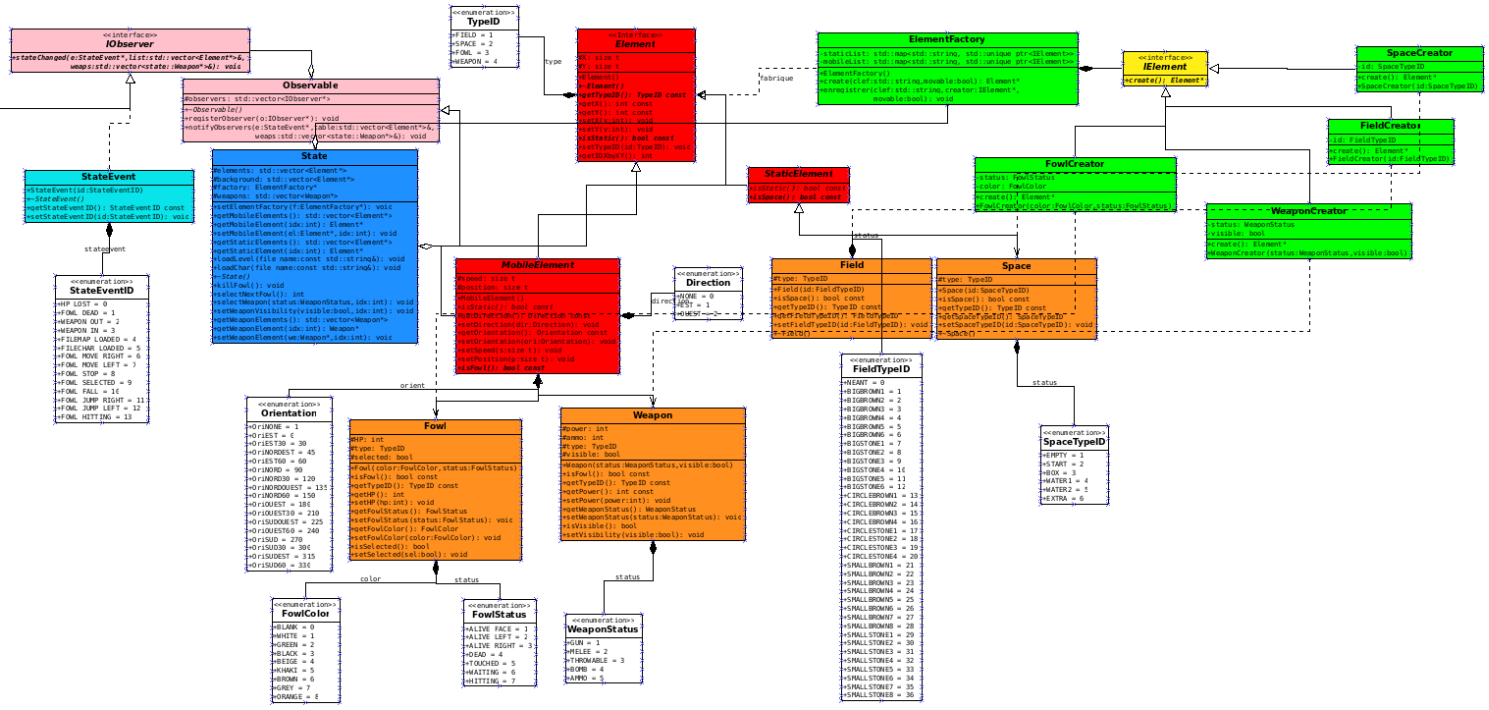

Illustration n°5 : Diagramme des classes pour l'état du jeu

2.4/ Ressources

Les niveaux sont codés sous la forme de matrice (ici 30x40), puis traduits en instance d'éléments grâce à la factory. Voici un exemple basique (uniquement cases «terrains») du contenu d’un fichier texte. Le rendu en texture correspondant est visible à la section 3.4/ (Illustration n°8) :

```
36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,
18,19,19,19,19,19,19,19,19,22,36,36,36,36,36,36,03,01,01,01,01,00,01,01,01,01,01,01,04,07,07,36,36,07,07,03,01,01,01,02,
29,29,29,29,29,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,11,11,11,11,11,11,11,36,36,36,36,36,36,36,36,11,11,11,11,
29,29,29,29,29,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,11,11,11,11,11,11,11,36,36,36,36,36,36,36,36,11,11,11,11,
29,29,29,29,29,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,11,11,11,11,11,11,11,01,01,01,01,04,36,36,36,11,11,11,11,
29,29,29,29,29,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,11,11,11,11,11,11,11,36,36,36,36,36,36,36,36,11,11,11,11,
29,29,29,29,29,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,11,11,11,11,11,11,11,36,36,36,36,36,36,03,01,11,11,11,11,
29,29,29,29,29,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,11,11,11,11,11,11,11,36,36,36,36,36,36,36,36,11,11,11,11,
29,29,29,29,29,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,11,11,11,11,11,11,11,01,01,01,04,36,36,36,36,11,11,11,11,
29,29,29,29,29,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,11,11,11,11,11,11,11,36,36,36,36,36,36,36,36,11,11,11,11,
29,29,29,29,29,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,11,11,11,11,11,11,11,36,36,36,36,36,03,01,01,11,11,11,11,
29,29,29,29,29,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,11,11,11,11,11,11,11,36,36,36,36,36,36,36,36,11,11,11,11,
29,29,29,29,29,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,11,11,11,11,11,11,11,01,01,04,36,36,36,36,36,11,11,11,11,
29,29,29,29,29,19,19,19,22,36,36,36,36,36,36,36,36,03,01,01,01,11,11,11,11,11,11,11,36,36,36,36,36,36,36,36,11,11,11,11,
29,29,29,29,29,29,29,29,36,36,36,36,36,36,36,36,36,36,11,11,11,11,11,11,11,11,11,11,36,36,36,36,03,01,01,01,11,11,11,11,
29,29,29,29,29,29,29,36,36,36,36,36,36,36,36,36,36,36,36,11,11,11,11,11,11,11,11,11,36,36,36,36,36,36,36,36,11,11,11,11,
29,29,29,29,29,29,36,36,36,36,36,36,36,36,36,36,36,36,36,36,11,11,11,11,11,11,11,11,01,04,36,36,36,36,36,36,11,11,11,11,
29,29,29,29,29,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,03,11,11,11,11,
29,29,29,29,29,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,11,11,11,11,
29,29,29,29,29,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,11,11,11,11,
29,29,29,29,29,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,03,01,01,01,01,11,11,11,11,
29,29,29,29,29,36,36,36,36,21,19,22,36,36,36,03,01,04,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,11,11,11,11,
29,29,29,29,29,36,36,36,36,36,29,36,36,36,36,36,11,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,11,11,11,11,
29,29,29,29,29,36,36,36,36,36,29,36,36,36,36,36,11,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,11,11,11,11,
29,29,29,29,29,36,36,36,36,36,29,36,36,36,36,36,11,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,11,11,11,11,
29,29,29,29,29,31,30,19,31,30,29,31,30,19,31,12,11,13,12,01,13,12,01,13,12,01,13,12,01,13,12,01,13,12,01,01,11,11,11,11,
29,29,29,29,29,36,36,29,36,36,29,36,36,29,36,36,11,36,36,11,36,36,11,36,36,11,36,36,11,36,36,11,36,36,11,11,11,11,11,11,
29,29,29,29,29,36,36,29,36,36,29,36,36,29,36,36,11,36,36,11,36,36,11,36,36,11,36,36,11,36,36,11,36,36,11,11,11,11,11,11,
29,29,29,29,29,36,36,29,36,36,29,36,36,29,36,36,11,36,36,11,36,36,11,36,36,11,36,36,11,36,36,11,36,36,11,11,11,11,11,11,
29,29,29,29,29,36,36,29,36,36,29,36,36,29,36,36,11,36,36,11,36,36,11,36,36,11,36,36,11,36,36,11,36,36,11,11,11,11,11,11
```

De même pour la génération des personnages (poules), un fichier texte permet de les générer sur la map qui sera affichée depuis l’exemple précédent. Les identifiants ne sont pas les mêmes mais sont indépendants de ceux permettant de générer le terrain.

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 surface : une surface pour les éléments « terrain » et « extra » (eau, nuage, arbre), une surface pour les éléments mobiles (poules, armes) et une surface pour les diverses informations (poule sélectionnée, vies, etc.) . La surface contiendra à la fois des informations sur les textures des terrains et personnages, mais aussi leur position sur la carte. Ces informations sont sous la forme de tableaux qui seront envoyés à la carte graphique via les méthodes des classes SFML.

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 dans le rendu, on met à jour automatiquement l’endroit du tableau de texture et/ou position correspondant, et on affiche de nouveau le rendu.

3.2/ Conception Logiciel

Le diagramme des classes pour le rendu général, indépendante de toute librairie graphique, est présenté en Illustration 8.

Plans et Surfaces. Le cœur du rendu réside dans la classe Layer (en rouge). Le principal objectif des instances de Layer est de recevoir les informations de changement d’état, et de transmettre les ordres à Surface pour obtenir et faire transmettre les éléments bas-niveau à la carte graphique. Cette implantation non représentée dans le diagramme, dépendra de la librairie graphique SFML. Les informations qui permettrons à l'implantation de Surface de former la matrice des positions des personnages ainsi que la matrice des textures seront données via la méthode loadChar(), de même pour le terrain avec loadMap(). La classe Layer est capable de réagir à des notifications de changement d'état via le mécanisme d'Observer.

3.3 Conception logiciel : extension pour les animations

Animations. Les animations sont gérées par les instances de la classe Animation. Chaque plan tient une liste de ces animations, et le client graphique fait appel aux méthodes de mise à jour pour faire évoluer ses surfaces. Nous faisons ce choix car certaines évolutions dans l'affichage ne dépendent pas d'un changement d'état, ou sont d'une fréquence différente. Pour les animations, on peut distinguer deux cas :

- Animations de déplacement horizontale (sauts de poules) : Par exemple, lorsque le joueur décide de déplacer sa poule en sautant, certes sa position changera, mais l’état n’en sera pas affecté. On génère ainsi une animation de saut. Les déplacements basiques de la poule sont quant à eux directement gérés dans la classe Surface grâce aux tableaux de vertex.
- Animations d’arme : Par exemple, lorsque le joueur décide d’appuyer dur la touche correspondante à l’action « Frapper » avec une arme au corps à corps, celui-ci doit pouvoir voir l’animation de la poule frappant devant elle.

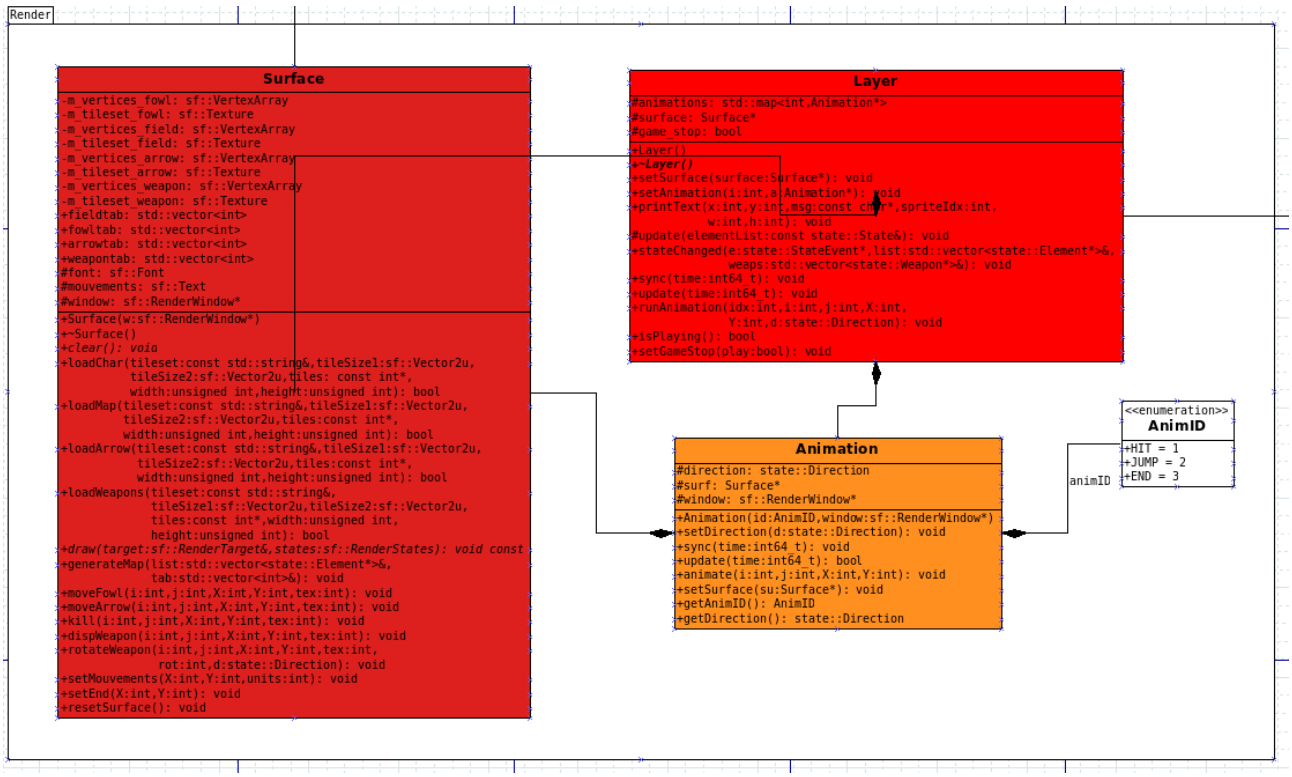

Illustration n°6 : Diagramme des classes du package Render

3.4/ Ressources

Voici les textures utilisées pour la construction du terrain :

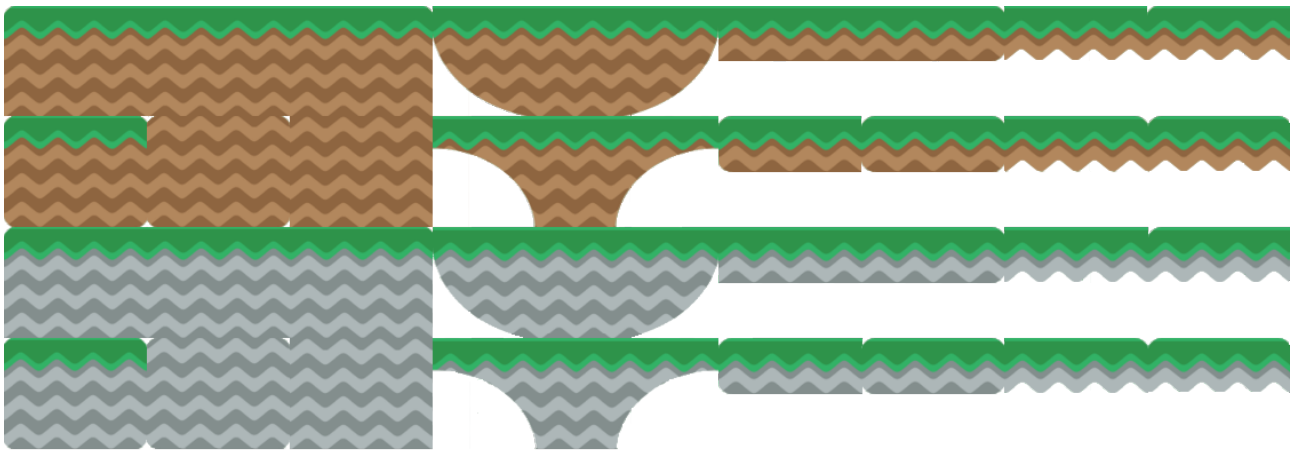

Illustration n°7 : Textures des terrains

3.5/ Exemple de rendu

Voici le rendu correspondant au fichier texte montré à la section 2.4/, ainsi qu'avec le fichier texte permettant d'afficher les poules et divers extras :

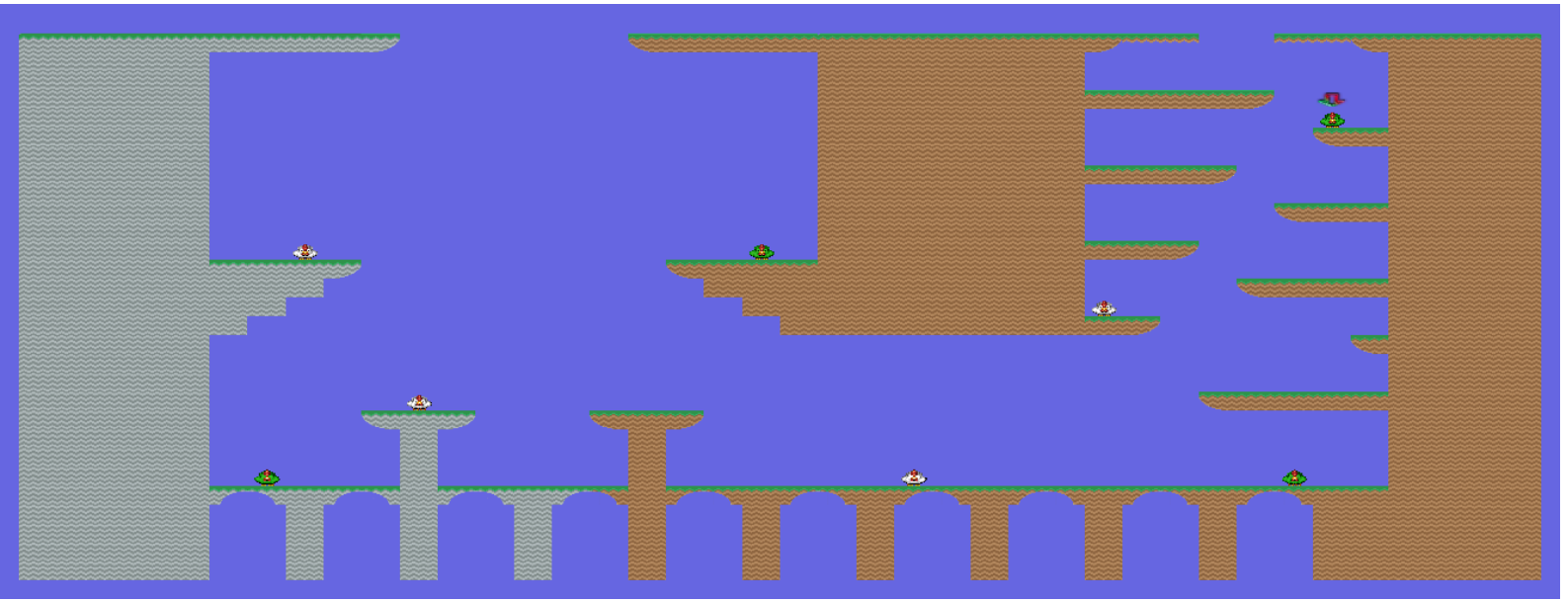

Illustration n°8 : Exemple de rendu du jeu

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

4.1/ Horloge Globale

Les changements d'état suivent une horloge globale : de manière régulière, on passe directement d'un état à un autre. Dans notre cas, notre horloge correspond à chaque action effectuée durant un tour. En effet, lorsque l'on déplace une poule, on communique un changement d'état qui traduit qu'une poule a été sélectionnée et qu'elle se met en mouvement. On attend ensuite de savoir ses prochaines actions.

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 :

1. Commandes d'IA : « Lancer IA d'un certain niveau » : Le jeu se déroule automatiquement et l'on assiste à une bataille entre IAs.
2. Commandes « Direction personnage », paramètres « personnage », « direction » : Si cela est possible (pas de mur), la direction du personnage est modifiée.

4.3/ 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. Appliquer les règles de mouvement pour la poule dont c’est le tour
2. Appliquer les règles de mouvement pour l’arme utilisée par la poule
3. Chaque arme a un nombre d’utilisation initial par équipe de poule. Chaque fois qu’une arme est utilisée, son nombre d’utilisation est décrémenté.
4. Si une poule est touchée par une arme, alors la poule obtient le statut « touchée ».
5. Si une poule a le statut « touchée » alors son attribut HP est diminuée relativement à l’arme qui l’a touchée. Ensuite la poule obtient le statut « en vie ».
6. Si l’attribut HP d’une poule tombe à 0, alors la poule obtient le statut « morte ».

4.4/ Conception logiciel

Le diagramme des classes pour le moteur du jeu est présenté en Illustration 9. 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 extérieure, provenant par exemple d'une touche au clavier (ou tout autre source). A ces classes, on a défini un type de commande avec CommandTypeId pour identifier précisément la classe d'une instance. En outre, on a défini une catégorie de commande, dont le but est d'assurer que certaines commandes sont exclusives. Par exemple, toutes les commandes de direction pour un personnage sont exclusives : on ne peut pas demander d'aller à la fois à gauche et à droite. Pour l'assurer, toutes ces commandes ont la même catégorie, et par la suite, on ne prendra toujours qu'une seule commande par catégorie (la plus récente).

Engine. C'est le cœur du moteur. Elle stocke les commandes dans une instance de CommandSet. Lorsqu'une nouvelle époque démarre, ie lorsqu'on a appelé la méthode update() après un temps suffisant, le principal travail du moteur est de transmettre les commandes à une instance de Ruler. C'est cette classe qui applique les règles du jeu. Plus précisément, et en fonction des commandes ou de règles de mises à jour automatiques, elle construit une liste d'actions. Ces actions transforment l'état courant pour le faire évoluer vers l'état suivant.

Action. Le rôle de ces classes est de représenter une modification particulière d'un état du jeu.

4.5 Conception logiciel : extension pour la parallélisation

Engine. Dans le but de pouvoir utiliser plus tard le moteur de jeu dans un thread séparé, nous ajouterons un deuxième jeu de commandes appelé waitingCommands qui récupère les commandes envoyés au moteur de jeu lorsque celui-ci met à jour l'état du jeu.

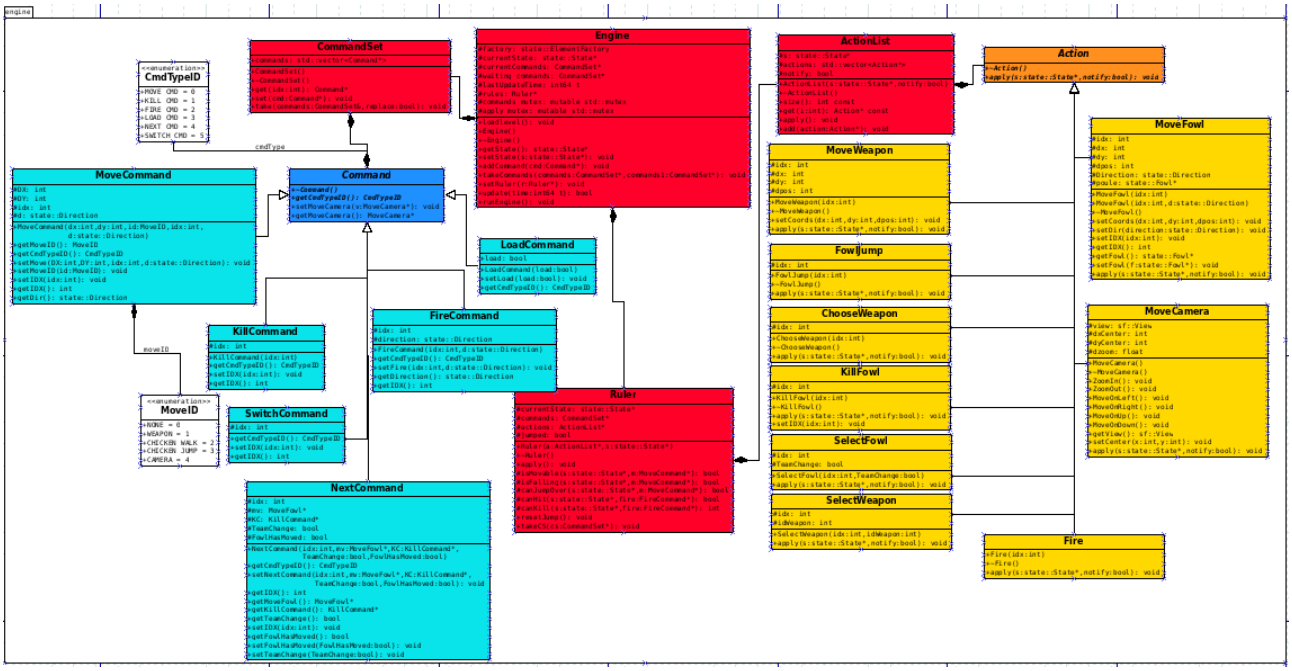

Illustration n°9 : Diagramme des classes pour le moteur du jeu

5 Intelligence Artificielle

5.1 Stratégies

L'ensemble de notre stratégie d'intelligence artificielle reposera sur différents niveaux d'intelligence, de l'IA la plus sommaire à la plus avancée. Les niveaux supérieurs feront appel aux niveaux inférieurs pour réduire les possibilités à étudier, en éliminant les comportements absurdes ou « dangereux ».

5.1.1 Intelligence minimale

Tout d'abord, nous avons créé une intelligence minimale basée sur des actions totalement aléatoires. C'est à dire que, quelque soit l'état actuel du jeu, l'intelligence agira de manière aléatoire, sans se soucier des possibles ennemis à côté d'elle.

5.1.2 Intelligence moyenne

Cette IA agira de manière plus raisonnée sur certaines situations, comparée à l'IA minimale. Ces situations sont :

- Lorsqu'une poule ennemie est située non loin d'elle sur la même ligne, elle se rapprochera alors d'elle.
- Si elle se retrouve à une distance faible de cette même poule ennemie, alors elle sortira son arme de corps à corps (l'épée), et frappera.
- Si aucune de ces conditions ne peut se remplir durant son tour, alors l'IA se déplacera aléatoirement tout comme le fait l'IA minimale.

5.1.3 Intelligence Forte

Cette dernière IA agit de manière plus raisonnée que l'IA moyenne. Elle peut décider de fuir

une poule ennemie si son nombre d'unités de déplacement ne lui permet pas de s'approcher assez d'une poule ennemie pour la tuer. Dans le cas contraire, elle s'en approche et essaie de la tuer. Où qu'elle soit, elle ne se déplace pas par hasard et décide chacun de ses mouvements. Bien que l'on n'utilise pas d'arbre Minimax pour l'IA forte, nous obtenons de meilleurs résultats qu'avec l'IA heuristique.

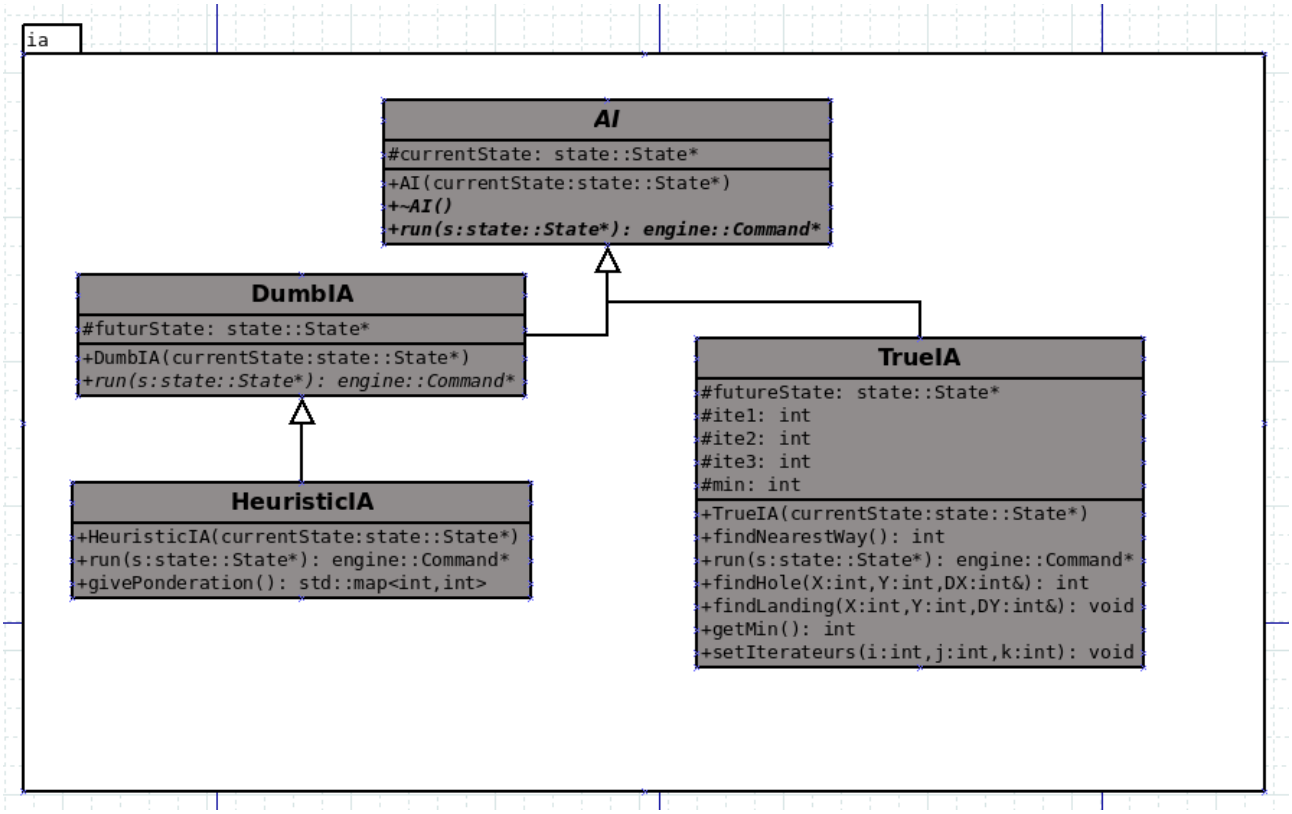

Illustration n°10 : Diagramme des classes pour l'IA

6 Modularisation

6.1 Liste des commandes

Lors d'une partie, le joueur humain peut utiliser plusieurs touches du clavier pour effectuer les actions qu'il souhaite. La liste ci-dessous répertorie les différentes touches du clavier qui sont associées aux différentes actions possibles en jeu :

- **Tabulation :** La touche Tabulation du clavier permet à l'utilisateur de choisir la poule avec laquelle il veut agir. Attention, une fois que la poule sélectionnée s'est déplacée, le joueur ne peut plus changer de poule et doit donc faire son tour avec la poule qu'il a déplacée.
- **Q :** La touche Q du clavier permet à l'utilisateur de déplacer sa poule vers la gauche.
- **D :** La touche D du clavier permet à l'utilisateur de déplacer sa poule vers la droite.
- **Entrée :** La touche Entrée du clavier permet à l'utilisateur de faire sauter sa poule.
- **Espace :** La touche Espace du clavier permet à l'utilisateur de faire en sorte que la poule sélectionnée frappe avec son épée.
- **Les touches directionnelles :** Les touches directionnelles (flèches) du clavier permettent à l'utilisateur de bouger la caméra comme il le souhaite. Cependant, il ne peut pas sortir la caméra de la map.
- **+ et – du pavé numérique :** Les touches + et – du pavé numérique permettent à l'utilisateur de zoomer (+) ou de dézoomer (-) la caméra (dans une certaine limite).
- **I :** La touche I du clavier permet à l'utilisateur de faire fonctionner (en boucle) l'intelligence artificielle minimale (DumbIA).
- **H :** La touche H du clavier permet à l'utilisateur de faire fonctionner (en boucle) l'intelligence artificielle moyenne (HeuristicIA).
- **G :** La touche G du clavier permet à l'utilisateur de faire fonctionner (en boucle) l'intelligence artificielle élevée (TrueIA)
- **L :** La touche L du clavier permet à l'utilisateur de recharger la map dans son état initial. Elle n'est pas encore utilisable pour le moment depuis que nous avons commencé le thread.

6.2 Organisation des modules

6.2.1 Répartition sur différents threads

Bien que cela ne soit pas une nécessité dans notre programme, l'objectif de cette étape est de pouvoir exécuté le moteur de jeu de manière totalement parallèle par rapport au reste (Rendu, Client) du programme. Ainsi nous avons créé un thread du Moteur communicant avec le thread du Rendu (la partie Client n'étant pas encore réalisée). Ces deux threads communiquent via deux informations :

Commandes : Depuis le thread principal (Rendu), l'utilisateur appuie sur des touches. Ces touches génèrent des commandes qui sont chargées directement dans la classe Engine (la classe phare du moteur de jeu). Ces commandes peuvent arrivées à n'importe quel moment, c'est pourquoi il est important de sécuriser les commandes en cours d'exécution par le moteur et celles qui arrivent. Ainsi, la classe Engine possèdent deux listes de commandes (provenant de la classe CommandSet), une s'appelant waiting_commands et l'autre currentCommands. Comme vous pouvez le deviner, toutes les commandes générées iront se stocker dans la variable waiting_commands, alors qu'en même temps le moteur fera exécuter les commandes contenues dans currentCommands. Une horloge au préalable lancée dans la fonction runEngine de la classe Engine permettra de mettre à jour périodiquement la liste currentCommands en la swappant avec waiting_commands. Cependant, des mutex ont été soigneusement placés pour éviter toute confusion lors d'une demande d'ajout de commande en même temps que l'on souhaite swapper les deux listes de commandes.

Notification de l'état vers le Rendu : Les actions générées par les commandes traitées par le Moteur de jeu ont un impact direct sur la classe State, qui représente l'état du jeu. A chaque exécution de la méthode apply d'une action, l'état se trouve modifié et notifie ce changement au Rendu. On a ainsi une communication du Moteur vers le Rendu en passant par le pattern Observer mis en place depuis le début du projet.

6.3 API Web

// Définition du format pour la liste de commandes. Cette dernière portera le nom de CommandSet et sera la liste de commandes demandée à chaque période de l'horloge.

```
type:object,
properties:{
    «CommandSet»:{
            type:array
            minItems:1,
            maxItems: 1000,
            items:{
                type: «object»,
                properties:{
                  «cmdTypeID»:{type: number, minimum: 0, maximum: 5},
                  «idx»:{type: number, minimum: 0, maximum: 1199},

                  … Les attributs suivant dépendent des valeurs de cmdTypeID:
                  // Cas où cmdTypeID=0 → MOVE_CMD
                  «DX»:{type: number, minimum: 0},
                  «DY»:{type: number, minimum: 0},
                  «direction»:{type: string, pattern: «([a-zA-Z])»},

                  // Cas où cmdTypeID=1 → KILL_CMD, rien à rajouter

                  // Cas où cmdTypeID=2 → FIRE_CMD
                  «direction»:{type: string, pattern: «([a-zA-Z])»},

                  // Cas où cmdTypeID=3 → LOAD_CMD, cas non traitées

                  // Cas où cmdTypeID=4 → NEXT_CMD
                  «TeamChange»:{type: bool},
                  «FowlHasMoved»:{type: bool},

                  // Cas où cmdTypeID=5 → SWITCH_CMD
                  «idWeapon»:{type: number, minimum: 0},
                },
                required:[«id», «cmdTypeID»],
            }
      }
},
required[«CommandSet»]
```

// Obtenir la prochaine liste de commandes:

```
1. requête: GET /CommandSet/<epoch>
2. réponses:
- Si il y a bien une liste de commande non vide à la bonne époque:
      (a) Status 200 = OK
      (b) Données:
            type: «object»,
            properties:{
                  «CommandSet»:{cf. Format JSON CommandSet},
            },
            required:[«CommandSet»]
- Cas où la liste de commande est vide: Status 404 = Not Found
- Cas où l'époque n'est pas la bonne: Status 401 = Unautorized
```

// Ajouter une commande à la liste sur le serveur:

1. requête: PUT /CommandSet
   Accept: application/json
   {
      item: Command (On ajoute une commande à la liste de commandes)
   }
2. réponses:
- Si la liste de commandes a encore de la place: Status 200 = OK.
- Si la liste est pleine: Status 401 = Unautorized

// Obtenir le numéro de période:

1. requête: GET /<epoch>
2. réponses:
- Si il existe bel et bien au moins une époque initiale:
   (a) Status 200 = OK
   (b) Données:
      type: «object»,
      properties:{
         «epoch»:{type: number, minimum: 0},
      },
      required:[«epoch»]
- Sinon: Status 404 = Not Found

// Format Json pour le Lobby, ce dernier servira à accueillir les différents utilisateurs dans des salles (rooms) pouvant contenir jusqu'à deux joueurs chacune.

type:object,
properties:{
   «Lobby»:{
   type: array,
   minItems:1,
   maxItems:10,
   items:{
      type:«object»,
      properties{
         «id»: {type:number, minimum:1, maximum:10},
         «status»: {type:bool},
         «players»: {**cf format JSON tableau Players**}
         },
         required[«id»,«status»,«players»],
      }
   }
},
required[«Lobby»],

// id:numéro de la room (Un Lobby peut contenir jusqu'à 10 rooms).
// status: état de la room (true=pleine/false=non remplie).
// players: liste des joueurs présents dans la room.

// Format Json du tableau «players»

```
type:object,
properties:{
      «players»:{
      type: array,
      minItems:0,
      maxItems:2,
      items:{
            type:«object»,
            properties{
                  «name»: {type:string, minLength:3},
                  «status»: {type:bool},
                  «host»: {type:bool}
                  },
                  required[«name»,«status»],
            }
      }
},
required[«players»],
```

// name: nom d'un joueur.
// status: état du joueur (true=présent/false=absent/afk).
// host: signale si le joueur est administrateur d'une room.

// Cas où un joueur arrive dans le lobby, il peut ainsi voir les rooms qui sont disponibles/indisponibles.

1.requête*:* GET /Lobby/
2.réponses:
– Cas où la liste des rooms existe:
      (a)Status 200 = OK
      (b)Données:
            type: «object»,
            properties:{
                  «Lobby»:{**cf format JSON Lobby**},
            },

– Cas où la liste des rooms n'existe pas: Status 404 = Not Found

// Cas où un joueur choisit une room

1. requête: PUT /room
      Accept: application/json
      {
            «name»:{type:string}
      }
2. réponses:
– Cas où room est disponible: Status 200 = OK
– Cas où la room est indisponible: Status 301 = Redirection vers le Lobby
– Cas où la room n'existe pas: Status 404 = Not Found

// Cas où un joueur arrive dans la room, on souhaite obtenir les informations de cette room

1. requête: GET /Lobby/room
2. réponses:
- Cas où le joueur est dans la room
    (a)Status 200 = OK
    (b)Données:
        type: «object»,
        properties:{
            «Room»:{**cf Item dans le format JSON Lobby**},
        },
– Cas où le joueur n’est pas dans la room: Status 401 = Unautorized

// Cas où un joueur veut créer une room

1. requête: PUT /Lobby/createroom
    Accept: application/json
2. réponses:
– Cas où la création d’une room est possible: Status 200 = OK
– Cas où le serveur est saturé: Status 401 = Unauthorized

//Le serveur s’occupera lui-même de supprimer la liste de commandes qu’il stocke et de supprimer les rooms qui se retrouvent vides lorsque tous les joueurs en partent.