

PROJET LOGICIEL TRANSVERSAL:
« CotCot WarFarm »

Jimmy HENNION - Antoine NOURRISSON

Table des matières

| | |
|--|----|
| 1/ Description du jeu..... | 3 |
| 1.1/ Présentation Générale..... | 3 |
| 1.2/ Règles du Jeu..... | 3 |
| 1.3/ Textures Employées..... | 4 |
| 2/ Description et conception des états..... | 5 |
| 2.1/ Description des états..... | 5 |
| 2.1.1/ États d'éléments fixes..... | 5 |
| 2.1.2/ États d'éléments mobiles..... | 6 |
| 2.1.3/ État général..... | 6 |
| 2.2/ Conception Logiciel..... | 7 |
| 2.3/ Conception Logiciel : extension pour le rendu..... | 8 |
| 2.4/ Ressources..... | 9 |
| 3/ Rendu : Stratégie et Conception..... | 10 |
| 3.1/ Stratégie de rendu d'un état..... | 10 |
| 3.2/ Conception Logiciel..... | 10 |
| 3.3 Conception logiciel : extension pour les animations..... | 10 |
| 3.4/ Ressources..... | 11 |
| 3.5/ Exemple de rendu..... | 12 |
| 4/ Règles de changement d'états et moteur de jeu..... | 12 |
| 4.1/ Horloge Globale..... | 12 |
| 4.2/ Changements extérieurs..... | 12 |
| 4.3/ Changements autonomes..... | 13 |
| 4.4/ Conception logiciel..... | 13 |
| 4.5 Conception logiciel : extension pour la parallélisation..... | 13 |
| 5 Intelligence Artificielle..... | 14 |
| 5.1 Stratégies..... | 14 |
| 5.1.1 Intelligence minimale..... | 14 |
| 5.1.2 Intelligence moyenne..... | 14 |

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 dure 30s maximum ou s'arrête lorsque le joueur a tiré/frappé avec son arme. Le décor est indestructible.

Les poules meurent lorsque:

- elles perdent tous leurs points de vie
- elles sont projetées hors de l'écran et se noient dans l'eau (effectivement, une poule ne peut ni voler ni nager !)

1.3/ Textures Employées

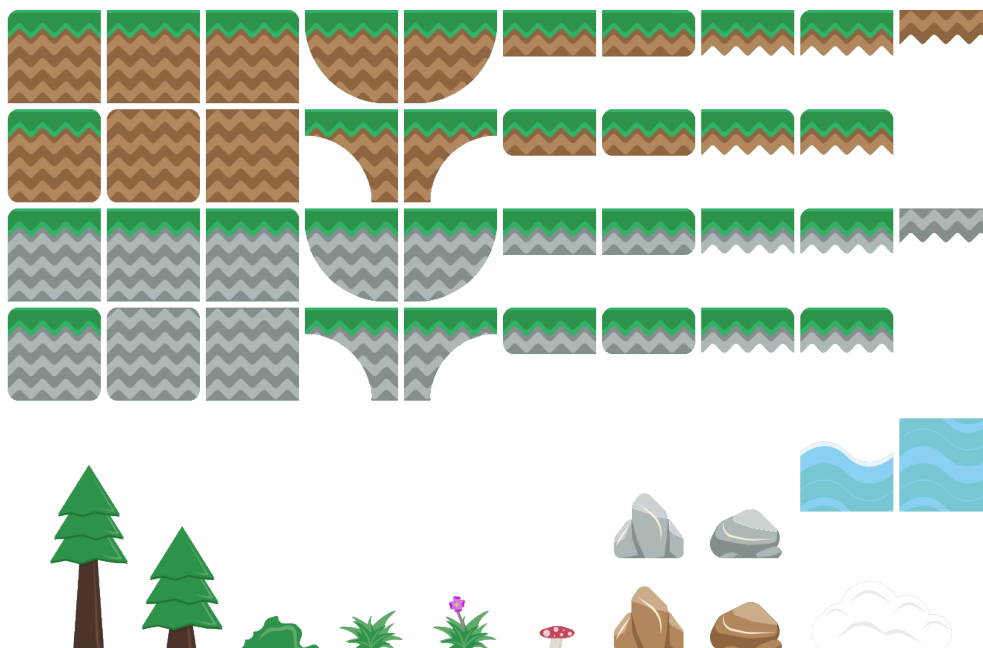


Illustration n°2 : Textures des terrains



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 :

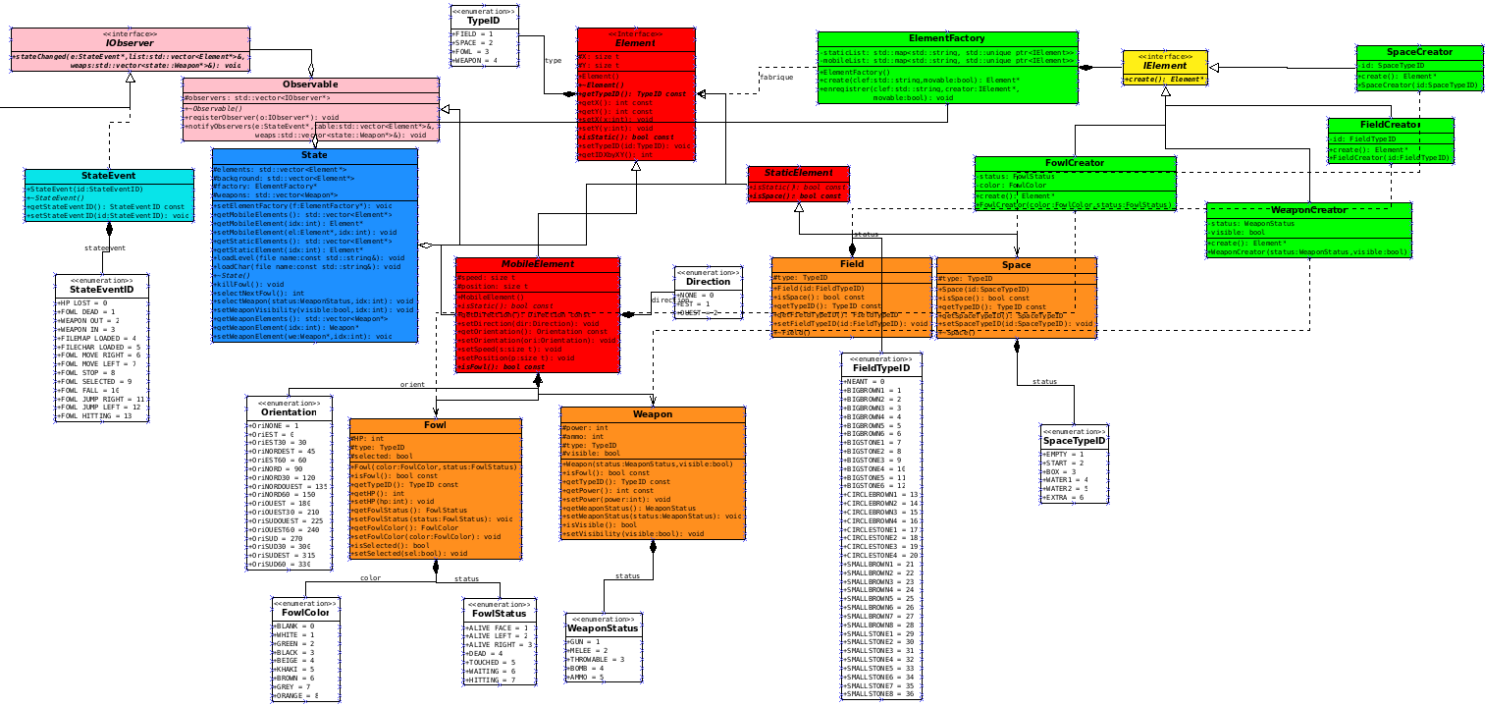


Illustration n°5 : Diagramme des classes pour l'état du jeu

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) :

[illegible]

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 permettront 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 sur 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 :

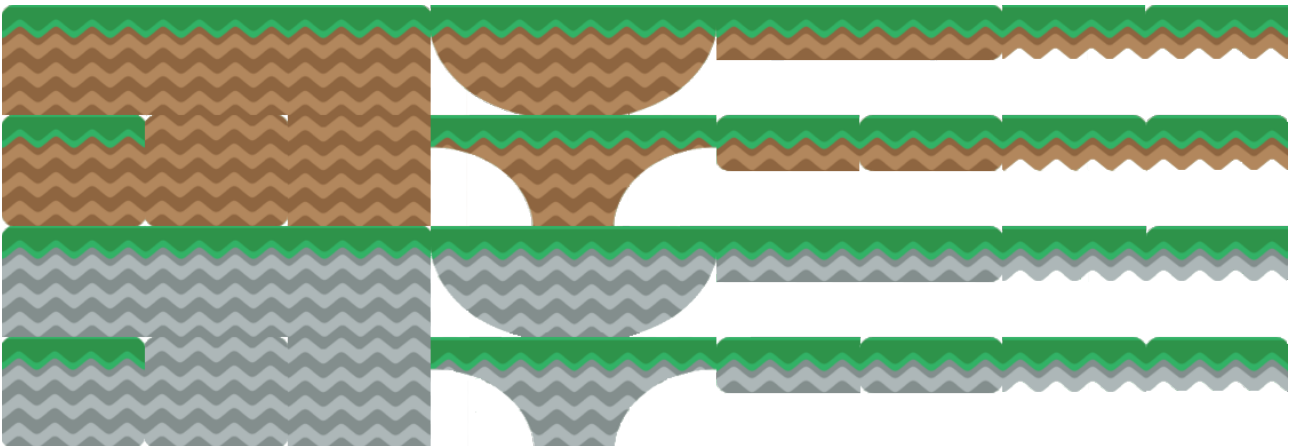


Illustration n°7 : Textures des terrains

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 :

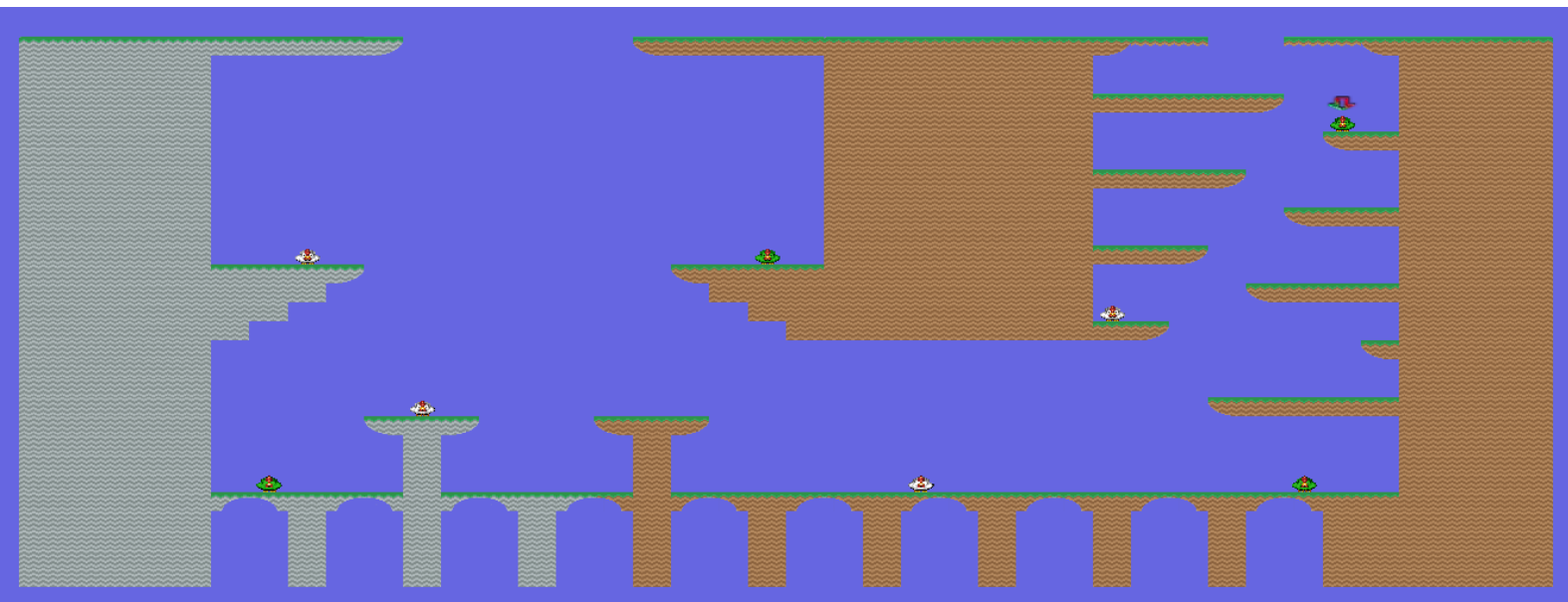


Illustration n°8 : Exemple de rendu du jeu

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.

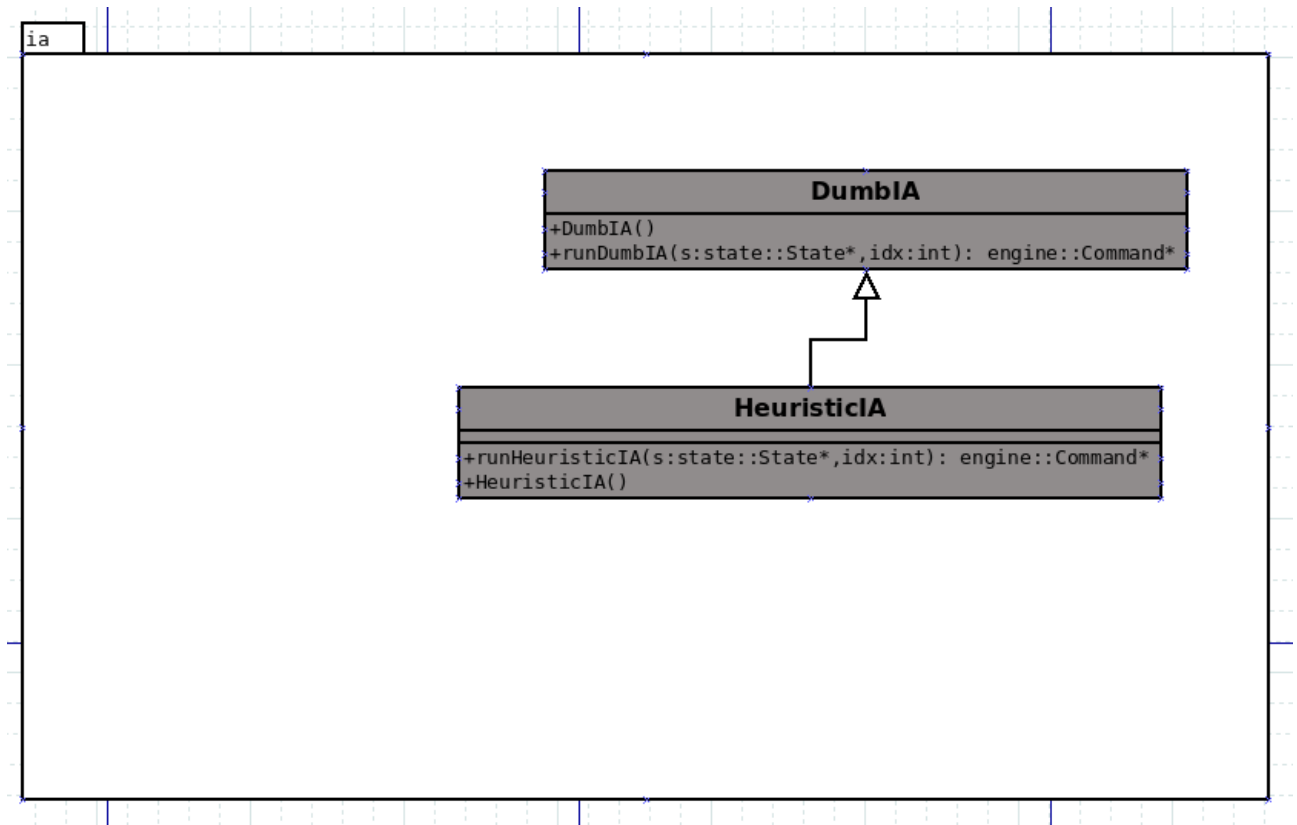


Illustration n°10 : Diagramme des classes pour l'IA