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 seize poules au maximum sur le terrain, donc il peut y avoir jusqu'à seize joueurs sur une partie contrôlant chacun une seule poule, ou bien deux joueurs contrôlant chacun 8 poules. Il est possible de rajouter des I.A., toujours jusqu'à un maximum de seize 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 avec une seule arme, utiliser des objets spéciaux… Chaque tour dure 30s maximum ou s'arrête lorsque le joueur a tiré 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

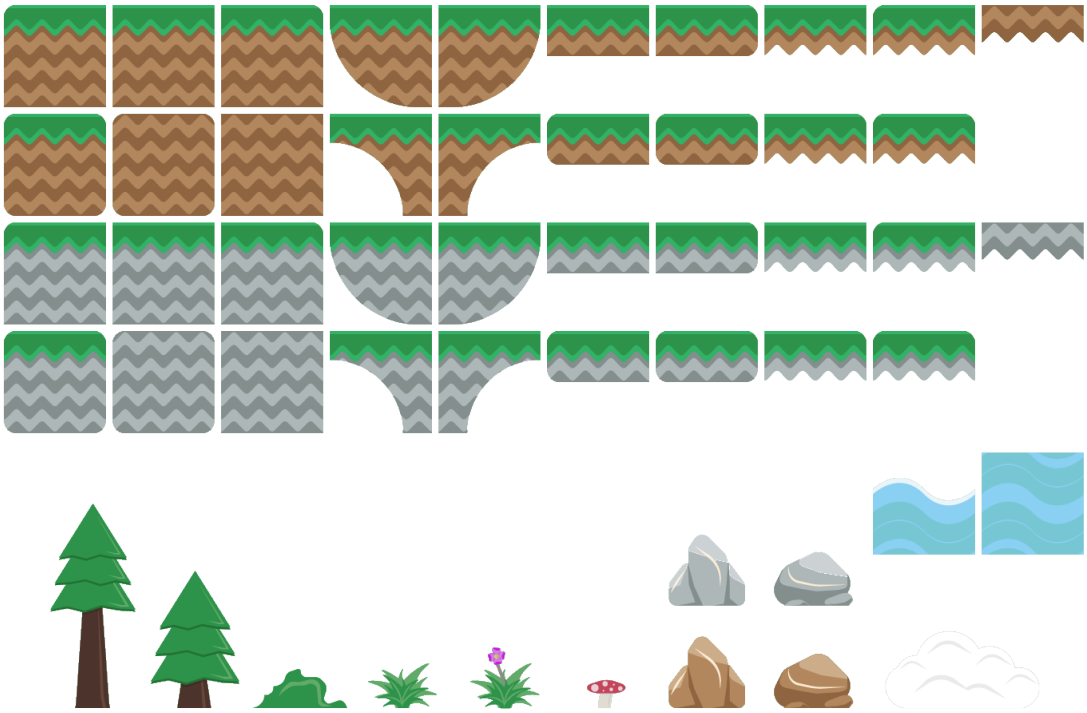

Illustration n°2 : Textures des terrains

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'arme se propose pour équiper les poules, cependant seulement une petite portion d'entre elles seront utilisées.

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). 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/ États d'éléments fixes

L'environnement est formée par une grille d'éléments nommée « cases ». La taille de cette grille 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.
- Les espaces « caisse », qui définissent les positions où peuvent apparaître une caisse d'armes.

2.1.2/ États d'éléments mobiles

Les éléments mobiles possèdent une direction (aucune, droite, gauche, haut, bas), une vitesse et une position. Ils sont dirigés par un joueur ou une IA.

Éléments mobiles « Poule ». Cet élément est dirigé par commande de la propriété de direction (aucune, droite et gauche). 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 « status », et qui peut prendre les valeurs suivantes :

- Status « Alive » : cas le plus courant, la poule est en vie et peut agir.
- Status « Hurt » : cas où la poule est blessée et perd des points de vie.
- Status « Dead » : cas où la poule est morte et ne peut plus agir.

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

- Status « Visible » : cas où l'arme est sortie, la poule est immobile et s'apprête à tirer.
- Status « Invisible » : cas où l'arme est dans la poche, la poule bouge ou c'est à l'équipe adverse d'agir.

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'*Eléments* 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 **Abstract 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 car seuls les éléments mobiles seront amenés 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

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

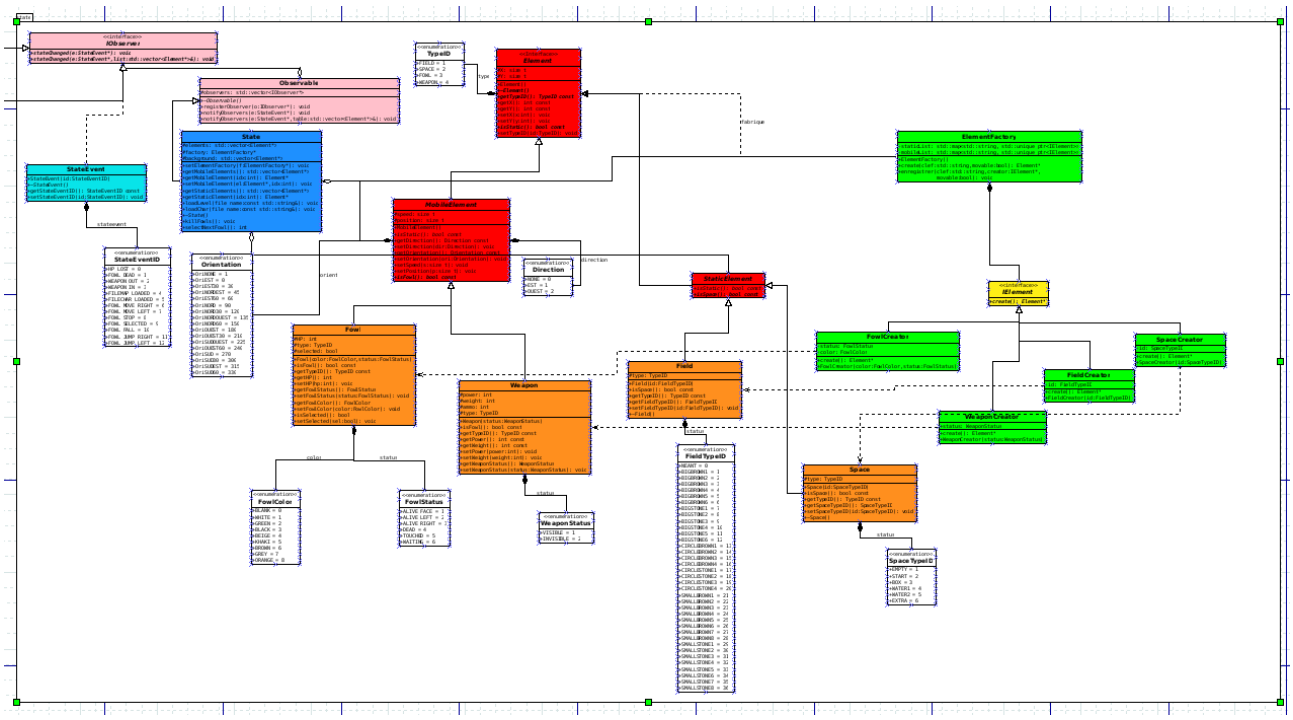

Illustration n°5 : Diagramme des classes pour l'état du jeu

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,
29,29,29,29,29,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,11,11,11,11,11,11,11,36,36,36,36,36,36,36,36,11,11,11,11,
29,29,29,29,29,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,11,11,11,11,11,11,11,36,36,36,36,36,36,36,36,11,11,11,11,
29,29,29,29,29,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,11,11,11,11,11,11,11,01,01,01,01,04,36,36,36,11,11,11,11,
29,29,29,29,29,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,11,11,11,11,11,11,11,36,36,36,36,36,36,36,36,11,11,11,11,
29,29,29,29,29,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,11,11,11,11,11,11,11,36,36,36,36,36,36,03,01,11,11,11,11,
29,29,29,29,29,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,11,11,11,11,11,11,11,36,36,36,36,36,36,36,36,11,11,11,11,
29,29,29,29,29,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,11,11,11,11,11,11,11,01,01,01,04,36,36,36,36,11,11,11,11,
29,29,29,29,29,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,11,11,11,11,11,11,11,36,36,36,36,36,36,36,36,11,11,11,11,
29,29,29,29,29,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,11,11,11,11,11,11,11,36,36,36,36,36,03,01,01,11,11,11,11,
29,29,29,29,29,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,11,11,11,11,11,11,11,36,36,36,36,36,36,36,36,11,11,11,11,
29,29,29,29,29,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,11,11,11,11,11,11,11,01,01,04,36,36,36,36,36,11,11,11,11,
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,
29,29,29,29,29,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,36,03,01,01,01,01,11,11,11,11,
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
```

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 plans (ou « layers ») : un plan pour les éléments « terrain », un plan pour les éléments mobiles (poules, armes), un plan pour les informations (vies, scores, etc.) et un plan pour les éléments « extras » (eau, arbres) . Chaque plan contiendra deux informations bas-niveau qui seront transmises à la carte graphique : une unique texture contenant les tuiles, et une unique matrice avec la position des éléments et les coordonnées dans la texture. En conséquence, chaque plan ne pourra rendre que les éléments dont les tuiles sont présentes dans la texture associée.

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 permanent dans le rendu, on met à jour le morceau de la matrice du plan correspondant.

Pour les changements non permanent, comme les animations et/ou les éléments mobiles, nous modifierons la matrice du plan automatiquement à chaque rendu d'une nouvelle frame.

.

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 donner les informations basiques pour former les éléments bas-niveau à transmettre à la carte graphique. Ces informations sont données à une implantation de Surface. Cette implantation non représentée dans le diagramme, dépendra de la librairie graphique choisie. L'ensemble classe Surface avec ses implantations suivent donc un patron de conception de type Adapter. La première information donnée est la texture du plan, via la méthode loadTexture(). Les informations qui permettrons à l'implantation de Surface de former la matrice des positions seront données via la méthode setSprite(). Notons que, dans un soucis d'efficacité, nous indexons tous les éléments graphiques (« sprites »). Ainsi, la surface sait qu'il faut gérer un nombre fixe de sprites, chacun identifié par un numéro unique. La classe Layer est capable de réagir à des notifications de changement d'état via le mécanisme d'Observer.

Scène. Tous les plans sont regroupés au sein d'une instance de Scene. Les instances de cette classe seront liées (« bind ») à un état particulier. Une implantation pour une librairie graphique particulière fournira des surfaces via les méthodes setXXXSurface(). Notons bien que les instances ont pour rôle de remplir les surfaces, mais pas de les rendre : cela restera le travail de la librairie choisie.

Tuiles. La classe Tile (en jaune) a pour rôle la définition de tuiles au sein d'une texture particulière. Elle stocke les coordonnées d'une unique tuile.Enfin, les implantations de la classe TileSet stocke l'ensemble des tuiles et animations possibles pour un plan donné. Notons que nous ne présentons pas dans le diagramme des implantations, uniquement la classe abstraite TileSet. En outre, on peut voir ces classes comme un moyen de définir un thème graphique : lors de la création d'instance de Layer, on pourra choisir n'importe quel jeu de tuile, pourvu qu'il contiennent des tuiles cohérentes avec le plan considéré.

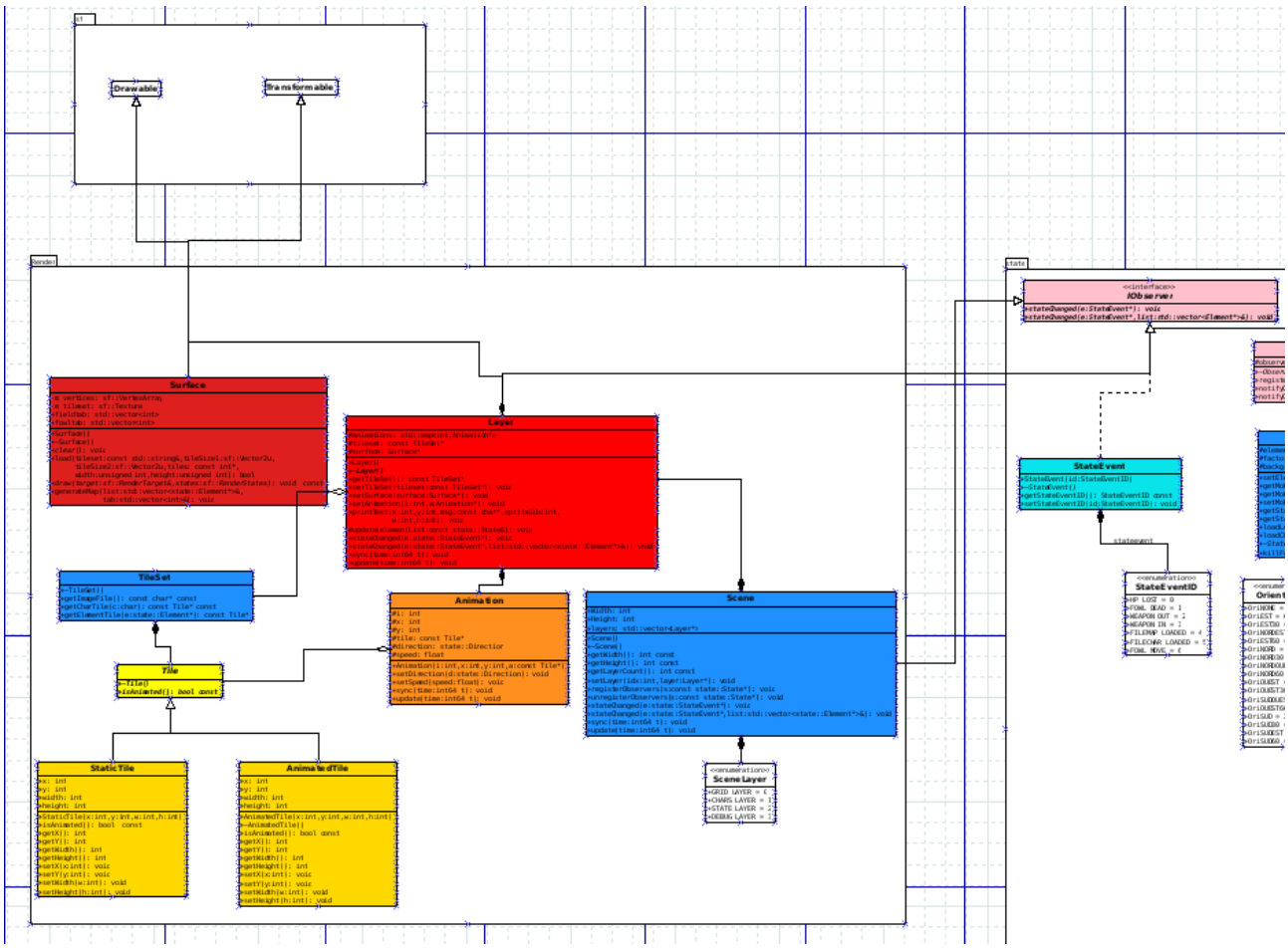

Illustration n°6 : Diagramme des classes du package Render

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 (poules) et rotatif (armes) : Par exemple, lorsque le joueur décide de déplacer sa poule, la poule doit alterner ses pattes. En effet, quand une poule marche, une fois c'est une certaine patte qui est devant et la fois suivante c'est l'autre patte, et ainsi de suite…
- Animations de trajectoire : Par exemple, lorsque le joueur décide d'appuyer dur la touche correspondante à l'action « Tirer » avec une arme à distance, celui-ci doit pouvoir voir la trajectoire parcourue par le projectile de l'arme. De même lorsque le joueur veut faire sauter une poule.

3.4/ Ressources

Voici les textures utilisées pour la construction du terrain :

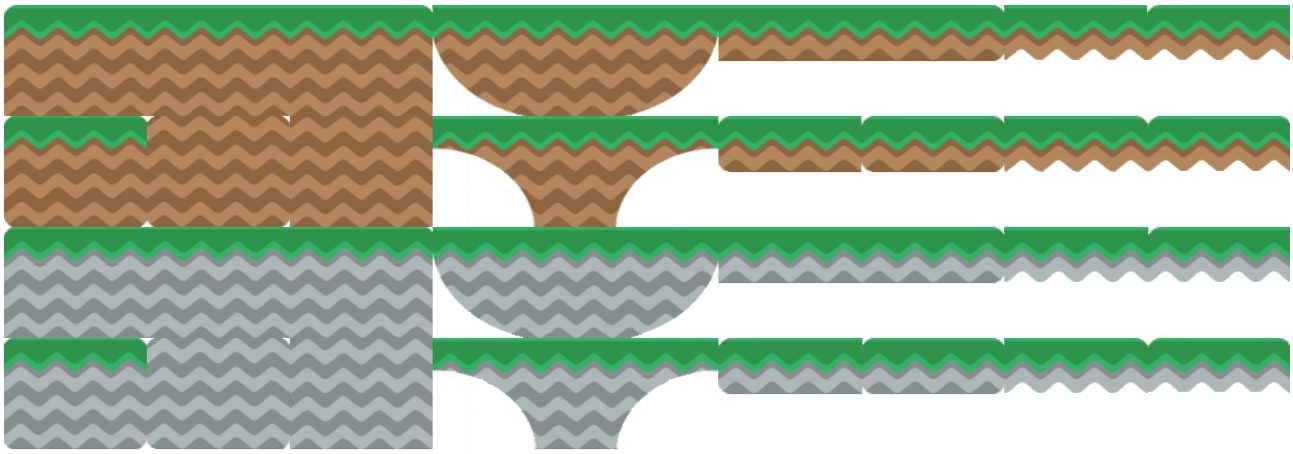

Illustration n°7 : Textures des terrains

3.5/ Exemple de rendu

Voici le rendu correspondant au fichier texte montré à la section 2.4/

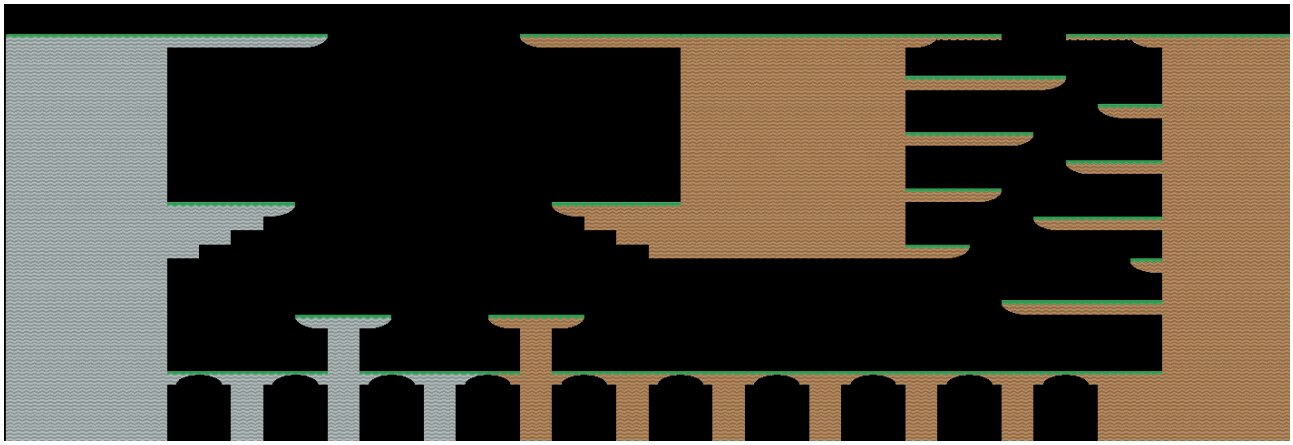

Illustration n°8 : Exemple de map

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 au changement de tour. En effet, lorsqu’un tour se termine, tous les états sont mis à jour.
Il n'y a pas de notion d'état intermédiaire. Ces changements sont calibrés sur le temps qu'il faut pour un élément mobile à vitesse maximale pour passer d'une case à une autre. En conséquence, tous les mouvements auront une vitesse fonction de cet élément temporel unitaire. Notons bien que cela est décorrélé de la vitesse d'affichage : l'utilisateur doit avoir l'impression que tout est continu, bien que dans les faits les choses évoluent de manière discrète.

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 principales : « Charger un niveau » : On fabrique un état initial à partir d'un fichier ; « Nouvelle partie » : Tout est remis à l'état initial
2. Commandes « Mode » : On modifie le mode actuel du jeu, comme « normal », « pause », « rejouer la partie », etc.
3. 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.

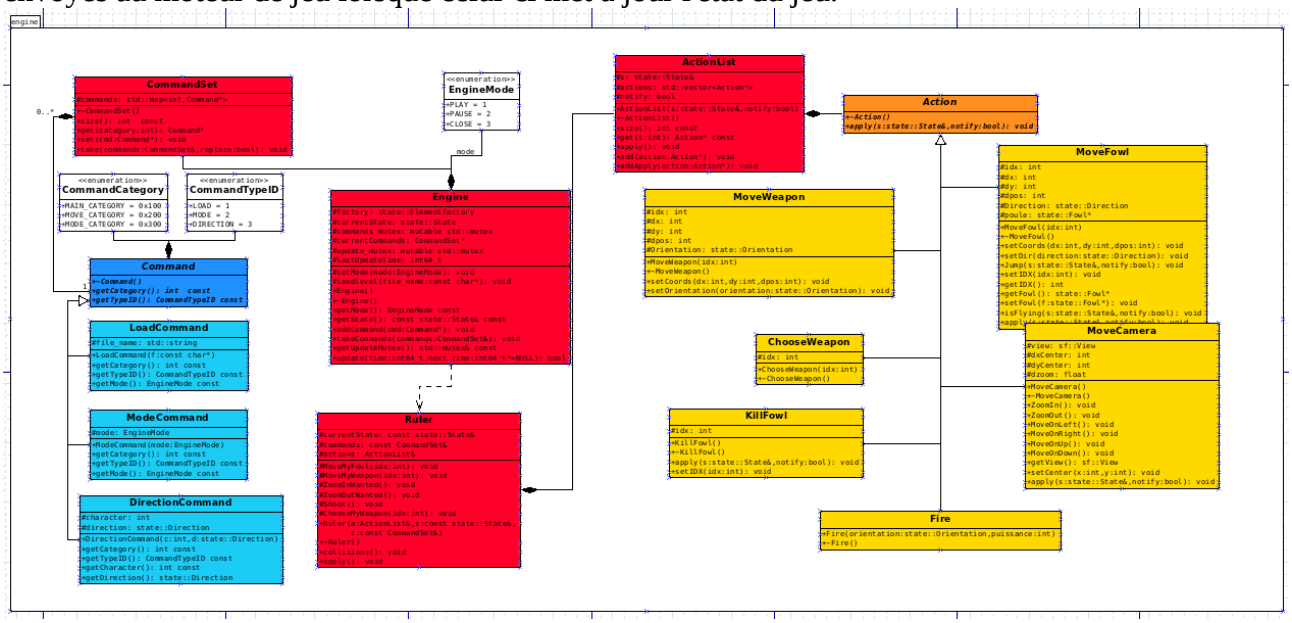

Illustration n°9 : Diagramme des classes pour le moteur du jeu

5 Intelligence Artificielle

5.1 Stratégies

L'ensemble de notre stratégie d'intelligence artificielle reposera sur le principe des poupées russes. L'ensemble sera décomposé en différents niveaux d'intelligence, de la plus sommaire à la plus avancée. Les niveaux supérieurs font appel aux niveaux inférieurs pour réduire les possibilités à étudier, en éliminant les comportements absurdes ou « dangereux ».

5.1.1 Intelligence nulle

Tout d'abord, nous avons crée une « intelligence nulle » basée sur des actions préprogrammées. C'est à dire que, quelque soit l'état actuel du jeu, l'intelligence agira de la même manière.

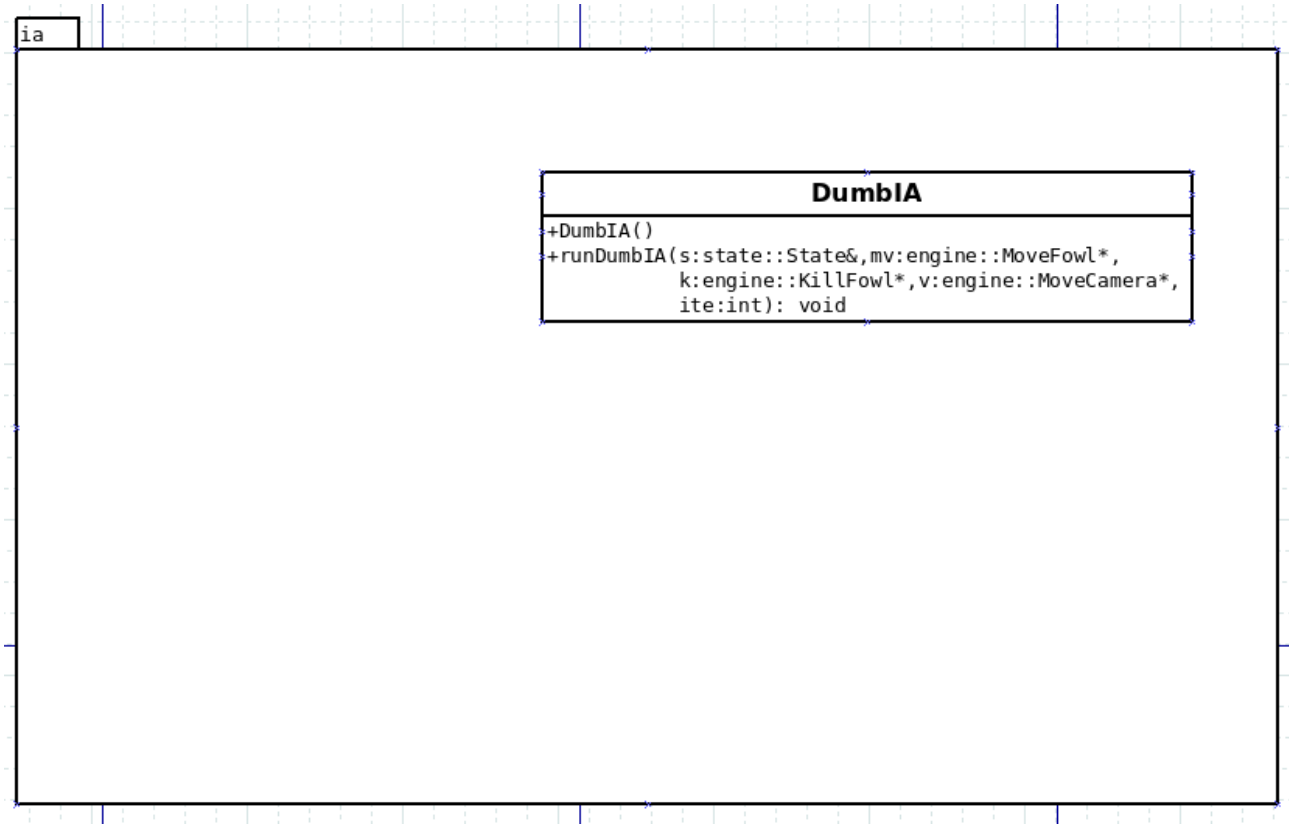

Illustration n°10 : Diagramme des classes pour l'IA