Projet Logiciel « Worms »

Guillaume Noailhac & Thomas Delhaye

Table des matières

I. Objectif

1. Présentation générale

Présenter ici une description générale du projet. On peut s'appuyer sur des schémas ou croquis pour illustrer cette présentation. Éventuellement, proposer des projets existants et/ou captures d'écrans permettant de rapidement comprendre la nature du projet.

L'objectif de ce projet est la réalisation d'un jeu de type Worms avec des règles simples ainsi qu'une représentation graphique épurée et en « tuile ». Dans ce jeu plusieurs équipes s'affrontent sur une carte, le vainqueur étant celui qui restera vivant jusqu'à la fin.

2. Règles du jeu

Présenter ici une description des principales règles du jeu. Il doit y avoir suffisamment d'éléments pour pouvoir former entièrement le jeu, sans pour autant entrer dans les détails. Notez que c'est une description en « français » qui est demandé, il n'est pas question d'informatique et de programmation dans cette section.

Environnement

Le jeu Worms sera en vue de coté et la gravité sera prise en compte.

L'environnement sera composé de blocs de terrain destructibles sur lesquels les personnages pourront évoluer, d'eau dans laquelle les personnages se noieront et finalement, de l'air, où ils pourront se déplacer.

Réglage de la partie

Avant le déroulement de la partie, l'utilisateur pourra choisir le nombre d'équipes présentent sur le terrain (entre 2 et 4 équipes) ainsi que le chef d'équipe (joueur ou intelligence artificielle). Chaque équipe sera composée de deux à quatre membres qui s'affronteront sur la même carte.

Déroulement d'une partie

Au début de la partie la carte est généré.

Au tour par tour, les joueurs contrôleront pendant deux minutes, dans l'ordre, un membre de leur équipe. Ce membre pourra alors lancer une capacité du joueur (déplacement, tir …) ce qui mettra fin à son tour.

Les personnages

Les dégâts subis peuvent provenir de joueur adverses mais aussi de membres de sa propre équipe (ou de sois même). Les membres des équipes démarrent tous avec un capital de 100 points de vie et meurent lorsqu'ils arrivent à 0. Les membres peuvent aussi mourir de noyade quelque soit l'état de leur vie.

Les capacités

Les capacités sont propres au joueur (partagées par tous les membres de l'équipe) et ne se rechargent pas au long de la partie. Pour l'ensemble des armes, le joueur devra définir la puissance du tir ainsi que la direction de celui ci.

II. Description et conception des états

L'objectif de cette section est une description très fine des états dans le projet. Plusieurs niveaux de descriptions sont attendus. Le premier doit être général, afin que le lecteur puisse comprendre les éléments et principes en jeux. Le niveau suivant est celui de la conception logicielle. Pour ce faire, on présente à la fois un diagramme des classes, ainsi qu'un commentaire détaillé de ce diagramme. Indiquer l'utilisation de patron de conception sera très apprécié. Notez bien que les règles de changement d'état ne sont pas attendues dans cette section, même s'il n'est pas interdit d'illustrer de temps à autre des états par leur possibles changements.

1. Description des états

Un état du jeu est formé par un ensemble de bloc (la carte) et de bloc Worms, chacun d'eux aura pour attribut les coordonnées (x, y) dans leur grille respective.

a. Etat des blocs de la carte

La carte est formé par une grille d'éléments nommé « cases ». La taille de cette grille est fixée au démarrage du niveau. Une case ne comprend qu'un seul type d'objet et ne peut en avoir plusieurs en même temps, elle peut soit contenir :

Bloc « espace » : Cette case représentera l'air, elle sera franchissable par le Worms.

Bloc « terrain » : Cette case sera infranchissable car elle est de type solide. Cette case pourra varier de texture, elle contiendra donc d'une variable allant de 0 à 3, suivant la solidité du bloc. Lorsque la solidité du bloc passera à zéro, celui ci deviendra une case de type espace ou eau suivant l'altitude.

Bloc « eau » : La case eau est une case franchissable, mais a pour effet la mort du Worms.

b. Etat des blocs Worms

La carte des éléments mobile sera une seconde grille de même taille que celle pour les éléments fixes, mais celle-ci aura seulement des objet vides ou :

Bloc « Worms » : Le bloc Worms est de type personnage, plusieurs types existeront, cela dépendra des caractéristiques du Worms qui pourront être par exemple empoisonné ou pas, à quelle équipe celui-ci appartient...

Les caractéristiques du bloc Worms seront :

-vie totale

-état du Worms (mort, vivant, empoisonné)

-place dans la grille

c. Etat des joueurs

Un joueur sera identifié par une couleur, il possèdera comme attribut :

-la vie total de son équipe

-un compteur de capacité restante

d. Etat générale

Dans l'état générale nous ajoutons une horloge, ainsi les tours des joueurs sera d'un temps limité et l'on pourra de même mesurer le temps de la partie global.

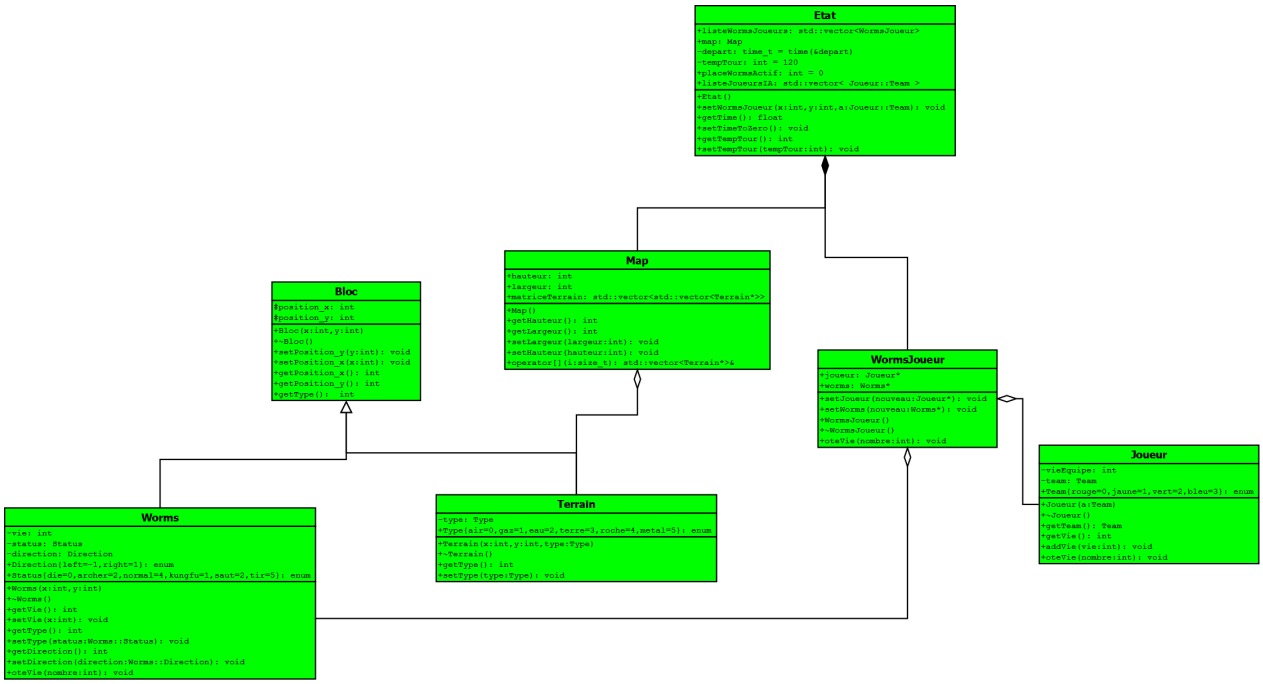


Diagramme des classes d'état

III. Rendu : Stratégie et Conception

Présentez ici la stratégie générale que vous comptez suivre pour rendre un état. Cela doit tenir compte des problématiques de synchronisation entre les changements d'états et la vitesse d'affichage à l'écran. Puis, lorsque vous serez rendu à la partie client/serveur, expliquez comment vous aller gérer les problèmes liés à la latence. Après cette description, présentez la conception logicielle. Pour celle-ci, il est fortement recommandé de former une première partie indépendante de toute librairie graphique, puis de présenter d'autres parties qui l'implémente pour une librairie particulière. Enfin, toutes les classes de la première partie doivent avoir pour unique dépendance les classes d'état de la section précédente.

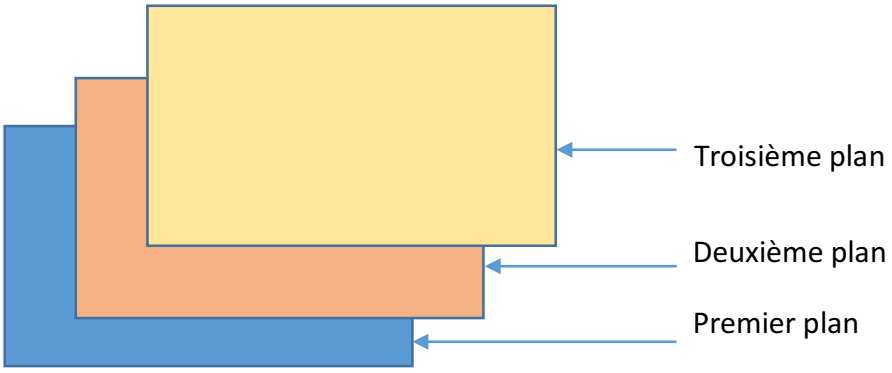
1. Stratégie de rendu d'un état

Nous avons décidé dans cette partie de travaillé avec trois plan bas niveau et une image de fond. Ces trois plans se recoupent les uns sur les autres, ce qui nous permettra de faire des calques et d'afficher les informations les plus importantes au premiers plan.

Pour éviter une surcharge du CPU, nous allons travailler avec des tuiles que la carte graphique devra aller chercher sur une seule image par plan.

Le premier plan représentera le terrain de la map (eau, terre, roche…).

Le second, qui lui sera ajouté sur le premier, représentera les Worms ainsi que leur point de vie. Pour finir le troisième qui lui représentera la vie globale restante de tous les joueurs ainsi que le menu (capacités...).



Le premier et le troisième plan seront remis à jour au démarrage de chaque tour (toutes les deux minutes). Quand au deuxième il sera mis à jour en fonction des actions.

Voici un exemple du rendu que l'on pourra avoir.

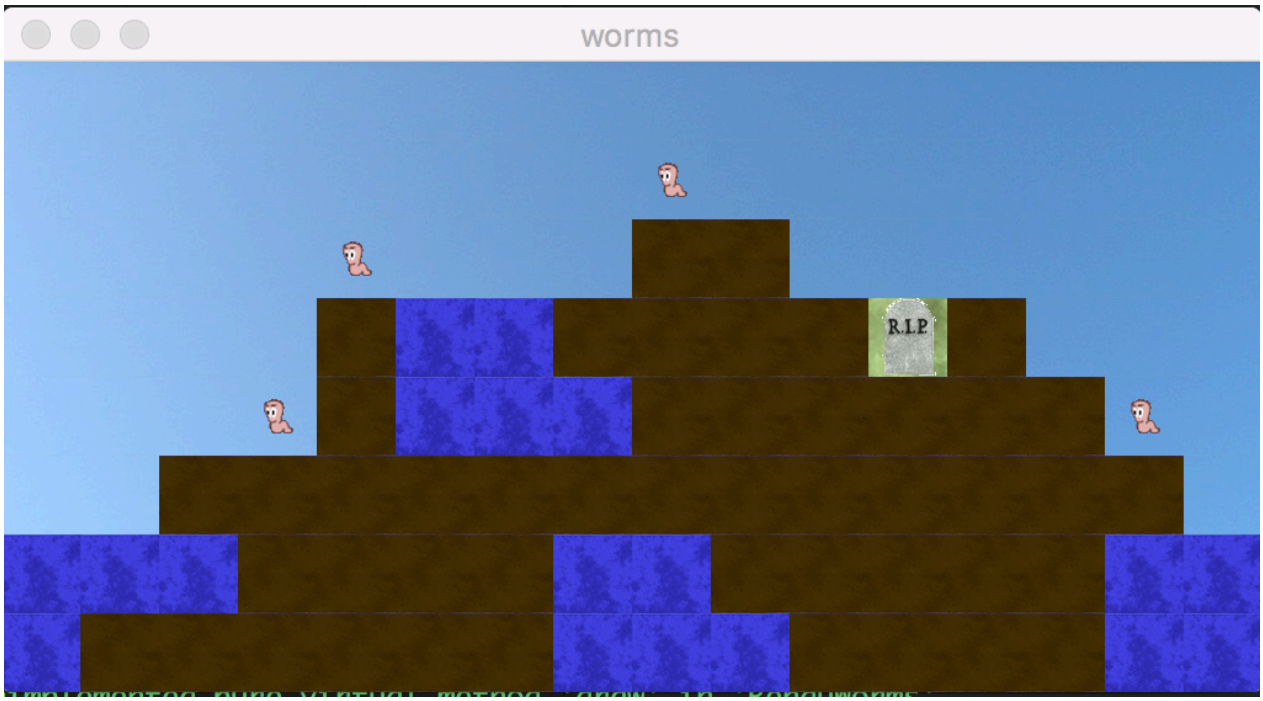
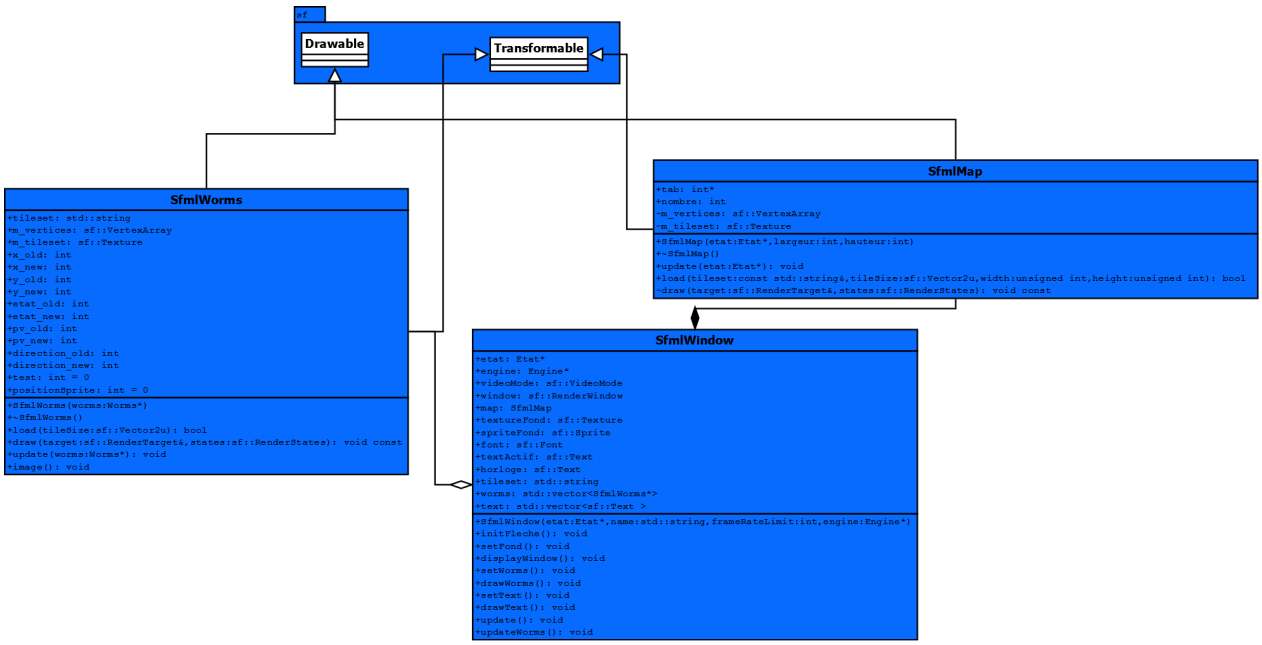


Diagramme de classes pour le rendu



Afin de communiquer entre le moteur de jeu et le moteur de rendu, nous avons décidé de mettre les donnés d'un état dans des fichiers .txt (situés dans res/txt). Ces fichiers sont donc générés par le moteur de jeu puis ensuite traités par le moteur de rendu afin d'obtenir un affichage.

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

1. Horloge globale

Les changements d'état sont calibrés sur les actions de l'utilisateur. Afin de réaliser des animations, le moteur de rendu se rafraichira 30 fois par seconde mais cela n'implique pas 30 changements d'état par seconde. En effet, l'état sera mis à jours lorsque l'utilisateur aura utilisé une capacité ou à la fin du décompte imposé de deux minutes. Arrivé à ce terme, le nouvel état sera calculé par le moteur de jeu et régénéré pour être ensuite envoyé au moteur de rendu par l'intermédiaire de fichiers .txt. Un fichier contiendra la map, un contiendra la liste Worms-joueur avec leur position et leur vie, un autre contiendra les capacités restantes des joueurs et un dernier les propriétés de la fenêtre de rendu.

2. Changements extérieurs

Les changements extérieurs proviennent des interactions de l'utilisateur. Elles peuvent provenir de pressions sur le clavier ou sur l'écran (souris ou tactile).

1) Les commandes principales permettent de charger le niveau, créer des équipes et lancer la partie.

2) Les commandes de la partie permettent de mettre à jour l'état de la partie en effectuant des déplacements, des tires et en utilisant diverses capacités.

3. Changements autonomes

Les changements autonomes s'effectueront après le la mise en action d'un changement extérieurs. Ils permettront de respecter les règles que l'on se sera fixé et la mise à jour continuel de notre jeu au cours des actions faites par l'utilisateur.

De mêmes elles seront aussi appliquées lors du changement de tour, c'est à dire lorsque le joueur aura fini son tour.

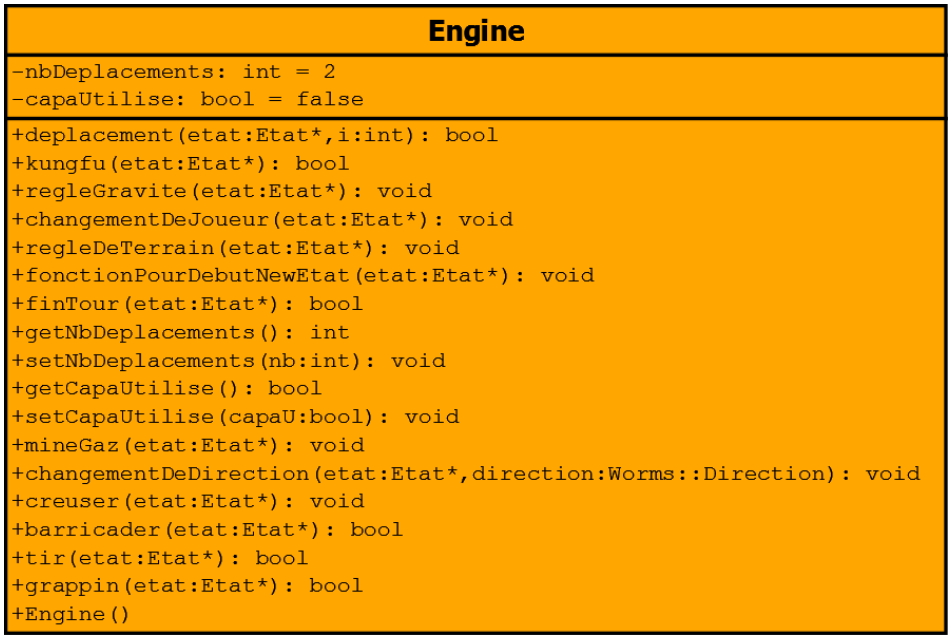
4. Conception logiciel

Classes Commandes : Cette classe représentera l'ensembles des actions qu'il sera permis de faire. Lors de l'appuis d'une touche, cette touche sera reliée à un ensemble d'action qui seront instancié dans la classe commandes.

Class Engine : Cette classe sera notre classe globale qui possèdera en attribut un état du jeu et une une classe commande qui nous permettra de faire des actions sur notre état du jeu, elle nous permettra donc de faire le lien entre notre état du jeu et la liste de nos actions possibles.

Elle devra aussi contenir une autre classe qui représentera l'historique de nos actions

Elle contiendra aussi l'ensemble de nos règles.



V. Intelligence Artificielle

1. Stratégies

L'ensemble de notre stratégie d'intelligence artificielle repose sur le principe des poupées russes. L'ensemble est 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 ».

a. Intelligence minimale

L’IA simple aura pour simple but de se rapprocher en direction d’un Worms adverse.

b. Intelligence basée sur des heuristiques

Dans le but d'avoir une sorte d'étalon, mais également un comportement par défaut lorsqu'il n'y a pas de critère pour choisir un comportement, nous proposons une intelligence normale, basée sur les principes suivants :

Tant que c'est possible on avance en direction du Worms adverse le plus proche tout en prenant compte du terrain pour éviter de mourir suite à des causes de noyades ou autres qui serait du au terrain traversé.

Lorsqu’un Worms adverse est à proximité une action hostile sera tenté dans sa direction.

Le Worms sera également capable d’anticiper s’il y a deux cases à monter alors il placera un bloc en avant des deux autres pour se construire un escalier se qui lui permettra de se déplacer normalement sans utiliser d’autre compétences.

Il aura aussi la faculté de combler certain espace vide ou remplis d’eau afin de pouvoir passer.

Au début de son tour et après chaque déplacement le Worms va vérifier s’il y a un autre Worms a son niveau et lui tirer dessus si tel est le cas. Il vérifiera aussi s’il y a un Worms à coté de lui afin de lui faire un coup de poing qui produira un maximum de dégât

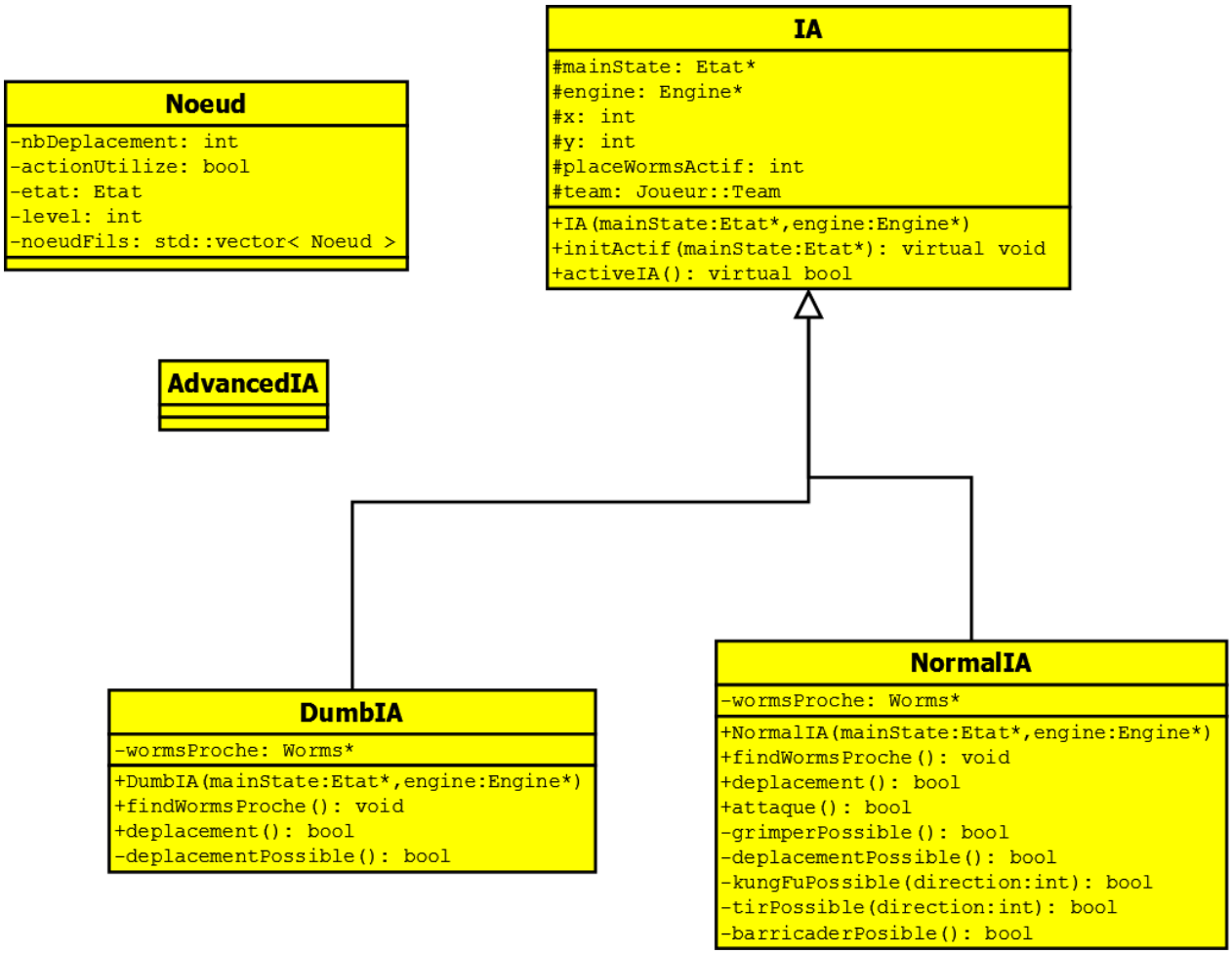


Diagramme de classes pour l'IA normal

Nous avons réalisé une interface IA qui sera implémentée par les différentes intelligence artificielle, elle aura pour attribues les caractéristiques du Worms contrôlé.

Chaque IA aura ces méthodes de calcul permettant la réalisation des actions qui seront directement envoyées depuis l'IA au moteur de jeu.

A ce stade l'IA s'exécute dans un thread différent du moteur de jeu, elle réalise ses calculs lorsque le Worms actif fait parti d'une des équipes qu'elle contrôle.

Notre objectif est de paralléliser les traitements en utilisant des threads pour les différentes parties de notre logiciel.

Nous cherchons donc à ce que le moteur de jeu, le moteur de rendu et l'IA travaillent en parallèle.

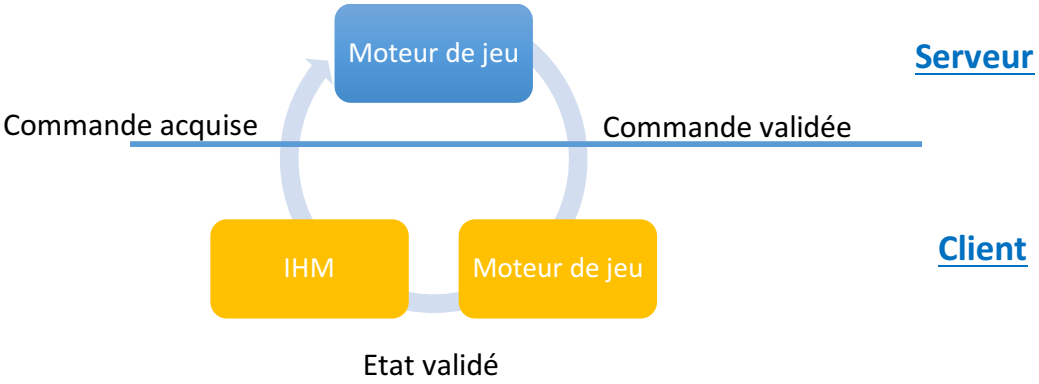
Néanmoins, ces différents modules sont susceptibles d'accéder au même moment à des ressources, ce qui peut entrainer des problèmes dans le fonctionnement global.

Le problème essentiel étant la synchronisation entre l'acquisition des commandes et le rafraîchissement de l'état de jeu.

La solution retenu fut de créer deux buffer, contenant des ensembles d'action à exécuter, dans le moteur de jeu dont l'un contiendra les commandes actuellement traitées par une mise à jour de l'état du jeu, et l'autre accueillera les nouvelles commandes. A chaque fois que le tampon contenant les commandes à traiter sera vide, on permute les deux tampons : celui qui accueillait les commandes devient celui traité par la mise à jour, et l'autre devient disponible pour accueillir les futures commandes. Ainsi, il existe toujours un tampon capable de recevoir des commandes, et cela sans aucun blocage. Il en résulte une parfaite répartition des traitements, ainsi qu'une latence au plus égale au temps entre deux époques de jeu.

Un autre problème concerne la synchronisation de l'affichage avec les modifications de l'état de jeu. Nous devons éviter d'afficher un état au même moment où celui ci est modifié.

2. Modularisation pour API Web



Les commandes sont générées depuis l'ihm par interaction de l'utilisateur, elles se trouvent don du coté client. Nous allons les transmettre par Json au moteur de jeux coté serveur, qui est en charge de valider ces commandes.

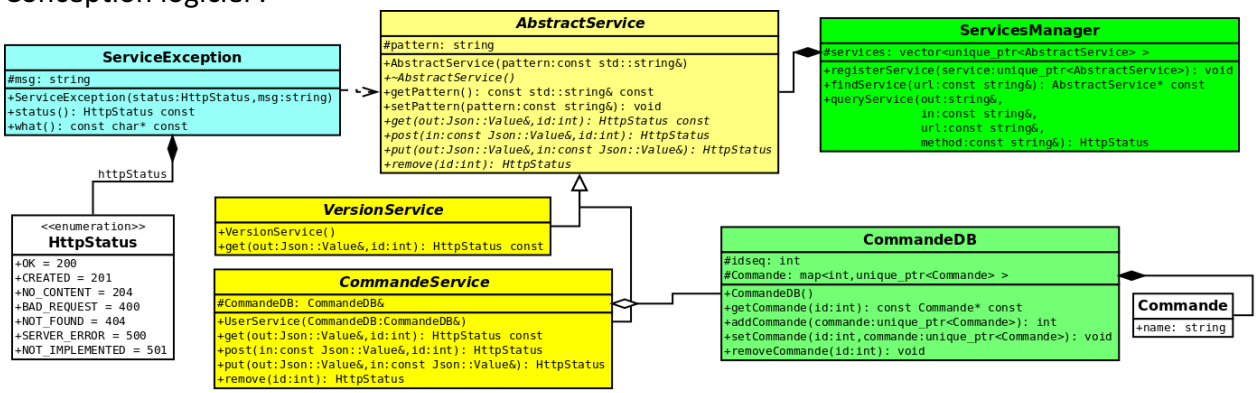
Exemple de commande au format Json :

```
{
    "name" : barricader
}
```

Il nous faudra donc sérialiser nos commandes par Json puis les désérialiser pour permettre au moteur de jeu sur le serveur de les interpréter.

Les commandes une fois validées par le serveur de jeu coté serveur seront renvoyées de la même manière au moteur de jeu du client qui les mettra en œuvre et les affichera sur l'ihm. Pour ce fait, ce n'est pas le serveur qui notifiera des changements mais le client qui interrogera régulièrement celui ci afin de se mettre à jour.

Conception logiciel :



Pour la mise en œuvre nous utiliserons l'implémentation d'un service REST vue en TD.

Nous allons ajouter une classe permettant le transfert des commandes au format Json sur le réseau mais aussi une classe envoyant directement les commandes au moteur de jeu local (côté client) afin de permettre à l'utilisateur de jouer sans passer par le réseau.