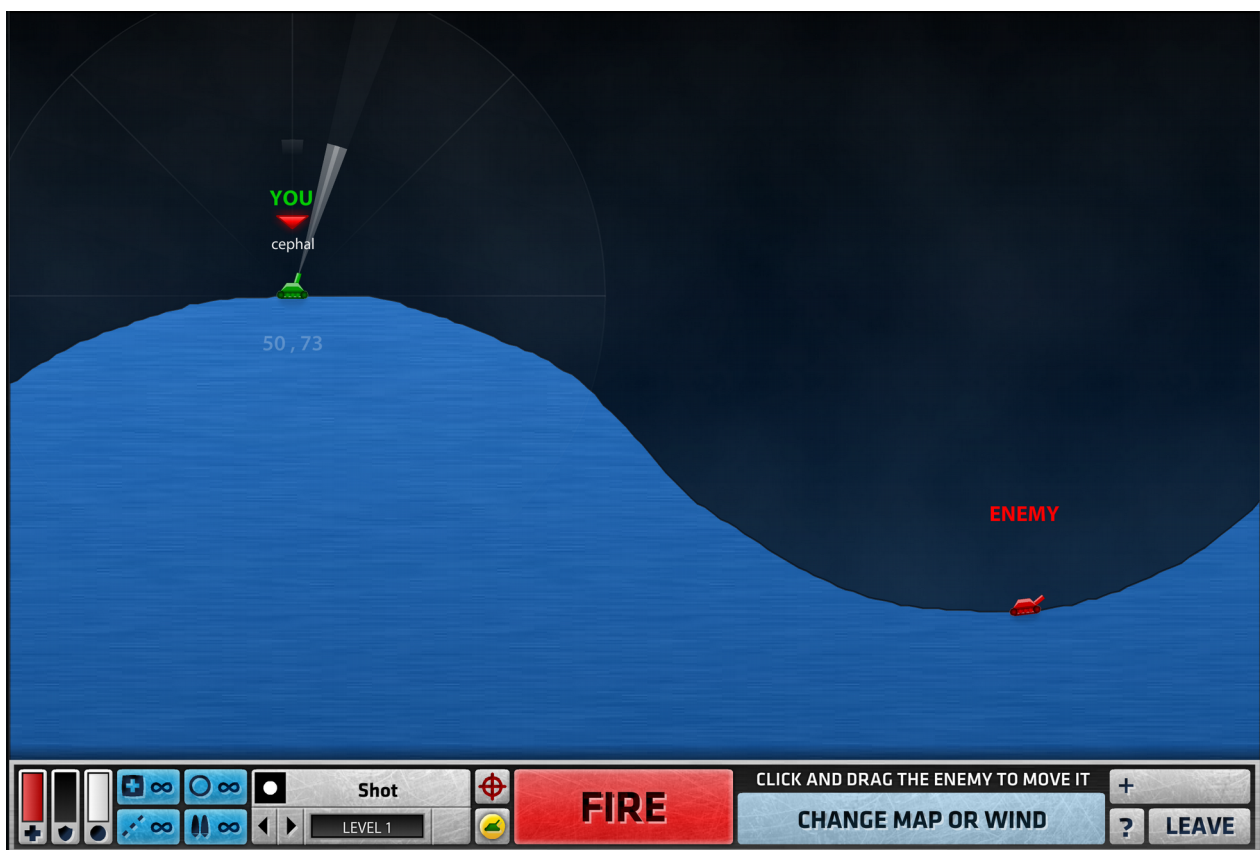


# Projet Logiciel Transversal

Pierre-Bernard LE ROUX – Corentin MORISSE



*Illustration 1: ShellShock Live 2, archétype de notre jeu*

# Table des matières

1	Objectif.....	3
1.1	Présentation générale.....	3
1.2	Règles du jeu.....	3
1.3	Conception Logiciel.....	3
2	Description et conception des états.....	4
2.1	Description des états.....	4
2.2	Conception logiciel.....	4
2.3	Conception logiciel : extension pour le rendu.....	4
2.4	Conception logiciel : extension pour le moteur de jeu.....	4
2.5	Ressources.....	4
3	Rendu : Stratégie et Conception.....	6
3.1	Stratégie de rendu d'un état.....	6
3.2	Conception logiciel.....	6
3.3	Conception logiciel : extension pour les animations.....	6
3.4	Ressources.....	6
3.5	Exemple de rendu.....	6
4	Règles de changement d'états et moteur de jeu.....	8
4.1	Horloge globale.....	8
4.2	Changements extérieurs.....	8
4.3	Changements autonomes.....	8
4.4	Conception logiciel.....	8
4.5	Conception logiciel : extension pour l'IA.....	8
4.6	Conception logiciel : extension pour la parallélisation.....	8
5	Intelligence Artificielle.....	10
5.1	Stratégies.....	10
5.1.1	Intelligence minimale.....	10
5.1.2	Intelligence basée sur des heuristiques.....	10
5.1.3	Intelligence basée sur les arbres de recherche.....	10
5.2	Conception logiciel.....	10
5.3	Conception logiciel : extension pour l'IA composée.....	10
5.4	Conception logiciel : extension pour IA avancée.....	10
5.5	Conception logiciel : extension pour la parallélisation.....	10
6	Modularisation.....	11
6.1	Organisation des modules.....	11
6.1.1	Répartition sur différents threads.....	11
6.1.2	Répartition sur différentes machines.....	11
6.2	Conception logiciel.....	11
6.3	Conception logiciel : extension réseau.....	11
6.4	Conception logiciel : client Android.....	11

# 1 Objectif

## 1.1 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.*

Notre projet consiste à créer un jeu tour par tour dont le gameplay s'inspire du jeu *ShellShock Live* 2 représenté sur l'Illustration 1 et disponible à l'adresse : <http://www.shellshocklive2.com/> (inscription requise). Il y a deux modes de jeu. Dans le premier mode, l'utilisateur jouera contre une IA. Le second mode sera multijoueur (2 joueurs) : un joueur A affrontera un joueur B.

## 1.2 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.*

Les deux joueurs (humain et IA ou humain A et humain B) peuvent déplacer un petit tank sur un décor 2D. Chaque tank dispose du même niveau de vie initial. Le tank possède deux types de tirs pour toucher le tank adverse : le tir horizontal où la visée est simple mais les dégâts sont moindres et le tir vertical où la visée est plus difficile mais les dégâts plus importants. Le premier joueur qui n'a plus de vie a perdu et cela met fin à la partie.

## 1.3 Conception Logiciel

*Présenter ici les packages de votre solution, ainsi que leurs dépendances.*

## 2 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 logiciel. 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.*

### 2.1 Description des états

Un état du jeu est formée par un ensemble d'éléments fixes (le paysage) et un ensemble d'éléments mobiles (les tanks et les projectiles). Tous les éléments possèdent les propriétés suivantes :

- coordonnées (x,y) dans la grille ;
- identifiant de type d'élément : ce nombre indique la nature de élément (ie classe).

#### 2.1.1 État Éléments fixes

Le paysage est formé par une grille d'éléments nommés « cases ». La taille de cette grille est fixée au démarrage du niveau. Les types de cases sont :

**Cases « Obstacle ».** Les cases « obstacle » 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 les éléments franchissables par les éléments mobiles. On considère les types de cases « espace » suivants :

- les espaces « vides » ;
- les espaces « départ joueur », qui définissent la position initiale du joueur (le cas échéant) ;
- les espaces « départ AI», qui définissent les position initiale des AI.

#### 2.1.2 État éléments mobiles

Les éléments mobiles possède une orientation (horizontal gauche, vertical gauche, horizontal droite, vertical droite).

**Élément mobile « Tank » .** Cet élément est dirigé par le joueur, qui commande la propriété d'orientation et de tir (le tir étant gérer dans le package « engine » ). Les tanks disposent également d'un « nombre de pv », qui sert à déterminer les points de vie restant avant de mourir, et d'un « TankTypeID » qui donne leurs couleurs et types (leurs texture). Enfin, on utilise une propriété que l'on nommera « PlayerType », et qui peut prendre les valeurs suivantes :

- PlayerType « realPlayer » : il s'agit du tank sur lequel agit les commandes du joueur humain.
- PlayerType « AIPlayer » : les tanks dirigés par l'AI.

**Éléments mobiles « Projectiles ».** Ces éléments sont également commandés par la propriété d'orientation (« right up » étant remplacé par « down », lorsque le missile redescend). Ils sont décomposé en **Élément mobile projectile Obus** et **Élément mobile projectile Missile**. La première catégorie d'éléments représente les obus tirés horizontalement qui explosent en touchant un obstacle, et la deuxième sont ceux qui font une trajectoire en cloche.

Les **Élément mobile projectile Missile** possèdent une propriété « hauteur » donnant l'apogée du missile, une propriété « distance » donnant la distance déjà parcour et une propriété « explose » indiquant si le missile a explosé.

## 2.2 Conception logiciel

Le diagramme des classes pour les états est présenté en Illustration 2: Diagramme des classes d'état, dont nous pouvons mettre en évidence les groupes de classes suivants :

**Classes Element.** Toute la hiérarchie des classes filles d'*Element* (en jaune) permettent de représenter les différentes catégories et types d'élément. Etant donné qu'il n'y a pas de solution standard efficace en C++ pour faire de l'introspection, nous avons opté pour des méthodes comme *isStatic()* qui indiquent la classe d'un objet. Ainsi, une fois la classe identifié, il suffit de faire un simple cast.

**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. L'idée est de l'utiliser, en autres, pour créer un niveau à partir d'un fichier texte. Par exemple, au caractère « - », on fait correspondre une instance de *Obstacle* avec la propriété « *ObstacleTypeId* » égale à *SAND*.

Dans le but de faciliter l'enregistrement des différentes instanciations possibles, nous avons créé le couple de classes *AElementAlloc* et *ElementAlloc<E,ID>*. La première est une classe abstraite dont le seul but est de renvoyer une nouvelle instance. La seconde sert à créer des implantations de la première pour une classe et une propriété d'identification particulière. Par exemple, pour créer le créateur de mur horizontal et l'enregistrer dans une factory, il suffit de faire :

```
factory.registerType('-', new ElementAlloc<Obstacle,ObstacleTypeId>(SAND));
```

**Conteneurs d'élément.** Viennent ensuite les classes *State*, *ElementList* et *ElementGrid* qui permettent de contenir des ensembles d'éléments. *ElementList* contient une liste d'éléments, et *ElementGrid* étends ce conteneur pour lui ajouter des fonctions permettant de gérer une grille. Enfin, la classe *State* est le conteneur principal, avec une grille pour le niveau, et une liste pour les éléments mobiles (les tanks et les projectiles).

## 2.3 Conception logiciel : extension pour le rendu

**Observateurs de changements.** Dans le diagramme de classes, nous présentons en mauve les classes permettant à des tiers de réagir lorsqu'un événement se produit dans l'un des éléments d'état. Les premiers intéressés par cette fonctionnalité sont les clients de rendu, qui pourront mettre à jour les textures/sprites en fonction des changements au sein de l'état actuel.

Les observateurs implantent l'interface *StateObserver* 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**.

## 2.4 Conception logiciel : extension pour le moteur de jeu

Au sein des différentes classes d'*Element*, ainsi que leurs conteneurs comme *ElementList* et *State*, nous avons ajouté des méthodes de clonage et de comparaison :

- Méthodes *clone()* : elles renvoient une copie complète (les sous-éléments sont également copiés) de l'instance.
- Méthodes *equals()* : elles permettent de comparer l'instance actuelle avec une autre, et renvoient *true* si les deux instances ont un contenu identique.

## 2.5 Ressources

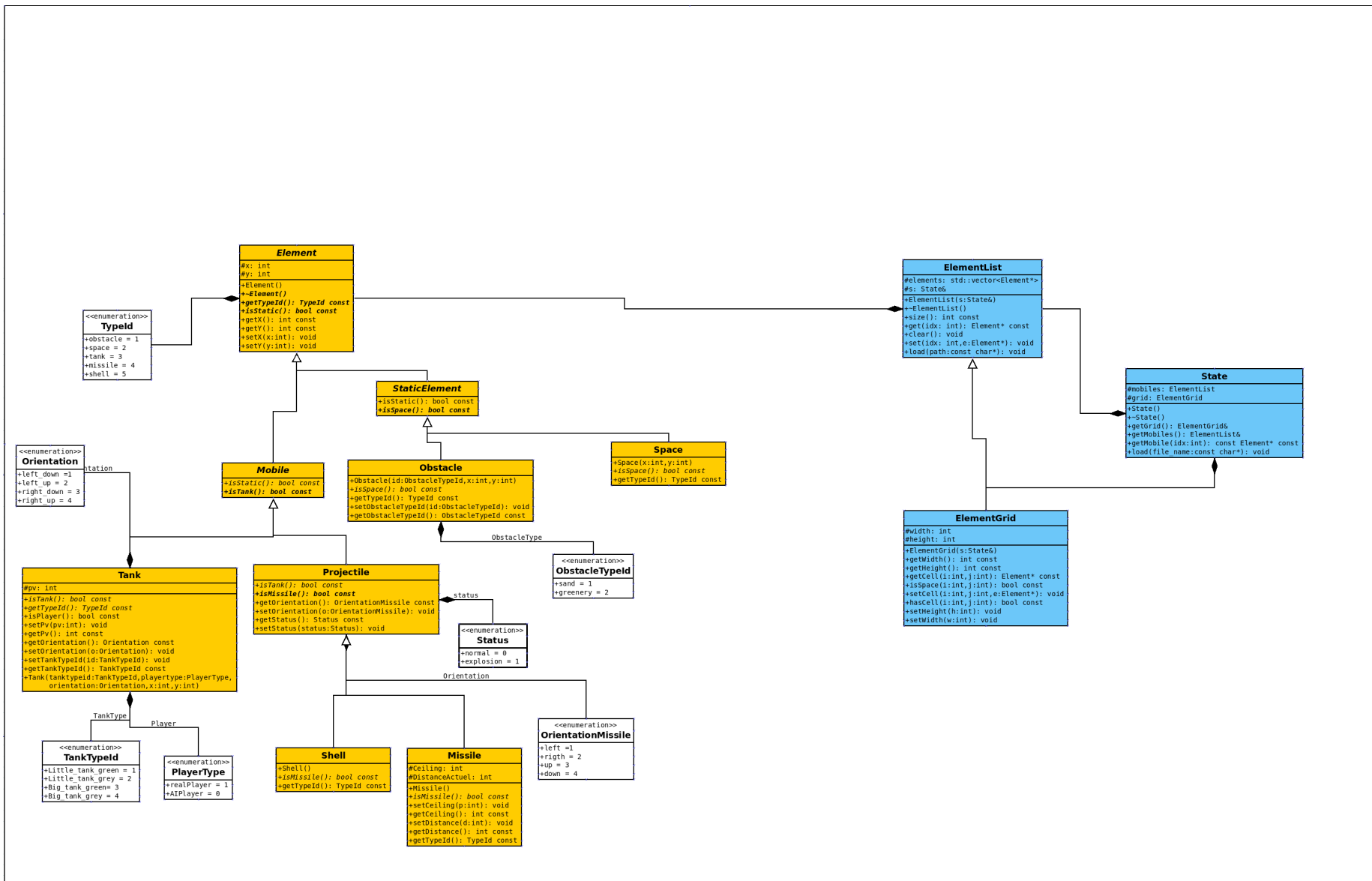


Illustration 2: Diagramme des classes d'état

## 3 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'implémentent 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.*

### 3.1 Stratégie de rendu d'un état

Pour le rendu d'un état, nous découpons la scène à rendre en plans (ou «layers»): un plan pour le niveau (sable, ciel, etc.) et un plan pour les éléments mobiles (tank et projectiles). Chaque plan contiendra deux informations bas-niveau qui seront transmises à la carte graphique: une unique texture contenant les tuiles (ou «tiles»), 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.

### 3.2 Conception logiciel

Le diagramme des classes pour le rendu général, indépendante de toute librairie graphique, est présenté en Illustration 4.

#### Plans et Surfaces.

Le cœur du rendu réside dans le groupe (en jaune) autour de la classe Layer. 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 (SFML ici).

La première information donnée est la texture du plan, via la méthode loadTexture(). Les informations qui permettront à l'implantation de Surface de former la matrice des positions seront données via la méthode setSprite().

#### 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. L'implantation pour SFML fournit 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 a pour rôle la définition de tuiles au sein d'une texture particulière avec leur position dans cette texture.

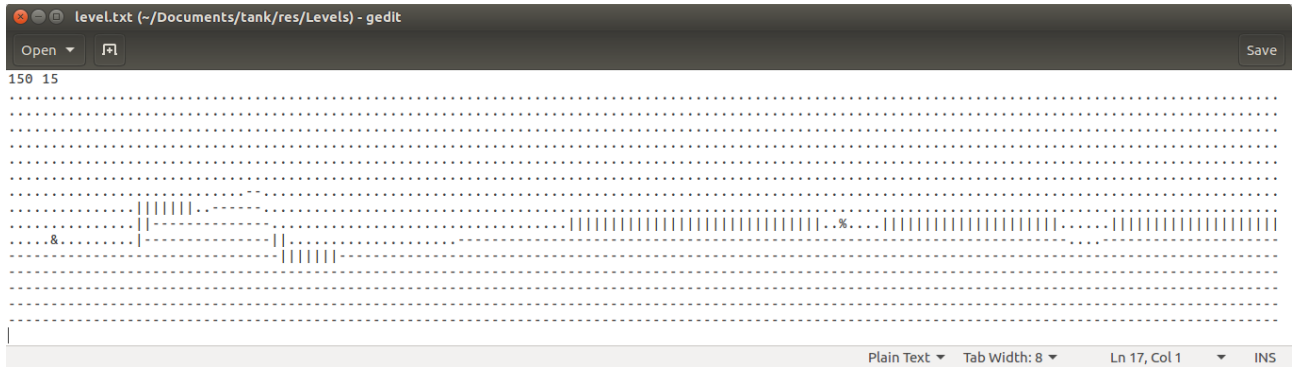


### 3.3 Ressources

Voici un exemple de textures pour le plan grille : 

### 3.4 Codage d'un niveau

Le rendu de chaque niveau du jeu est défini dans un fichier .txt. Voici un exemple :

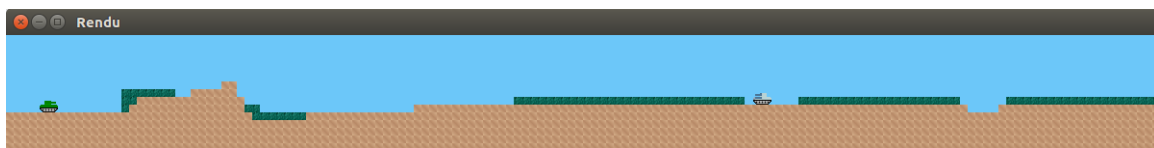


Les différents symboles qui représentent les éléments du rendu (Illustration 3) sont :

- « & » : représente le tank 1 commandé par un joueur humain ;
- « % » : représentent le tank 2 commandé par un joueur humain ou par l'IA ;
- « . » : représente le ciel bleu ;
- « - » : représente le sol ;
- « | » : représente les éléments de verdure (en vert sur Illustration 3).

### 3.5 Exemple de rendu

Voici typiquement un exemple de rendu pour un niveau :



*Illustration 3: Aperçu du rendu*

render

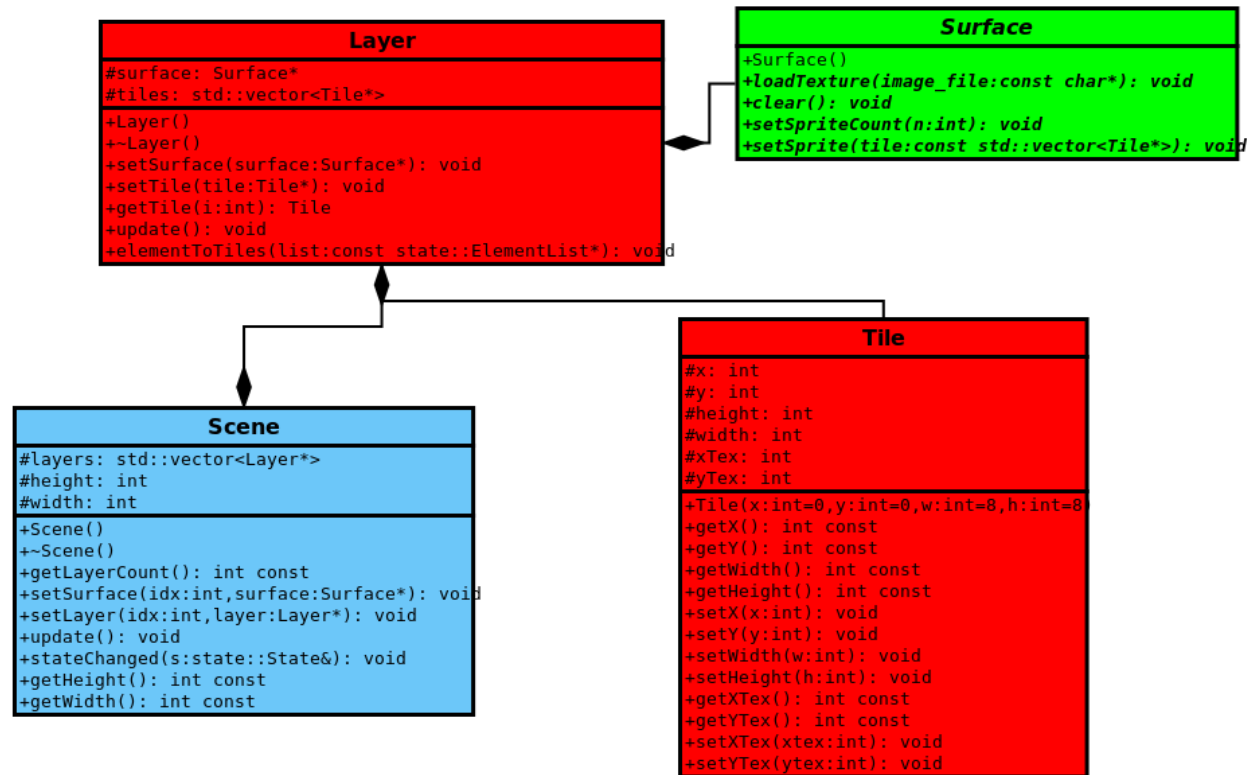


Illustration 4: Diagramme de classes pour le rendu

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

*Dans cette section, il faut présenter les événements qui peuvent faire passer d'un état à un autre. Il faut également décrire les aspects liés au temps, comme la chronologie des événements et les aspects de synchronisation. Une fois ceci présenté, on propose une conception logiciel pour pouvoir mettre en œuvre ces règles, autrement dit le moteur de jeu.*

### 4.1 Horloge globale

Les changements d'état suivent deux horloges : de manière régulière, le thread chargé de l'engine vérifiera les commandes pour charger un niveau et celle pour changer de mode ; et lorsque l'on appuie sur « fin du tour » les autres actions stockées sont effectuées (si licite).

### 4.2 Changements extérieurs

Les changements extérieurs sont provoqués par des commandes extérieures, 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.
2. Commande « Mode » : On modifie le mode actuel du jeu, comme «normal » ou « pause ».
3. Commande « déplacement vers » : Si cela est possible (pas de mur), la position du personnage est modifiée.
4. Commande « Orientation » : On change l'orientation du personnage.
5. Commande « Shot » : On tire un projectile, calcule et anime son déplacement.
6. Commande « fin du tour » : On finit le tour et applique les commandes stockées

### 4.3 Changements autonomes

Les changements autonomes sont le déplacement des projectiles après leur création via la commande « shot ».

### 4.4 Conception logiciel

Le diagramme des classes pour le moteur du jeu est présenté en Illustration 5.

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 toute autre source). Notons bien que ces classes ne gèrent

absolument pas l'origine des commandes, ce sont d'autres éléments en dehors du moteur de jeu qui fabriquerons les instances de ces classes. A ces classes, on a défini un type de commande avec `CommandCategory` pour identifier précisément la classe d'une instance et s'assurer que chaque commande est exclusive. Par exemple, toutes les commandes d'orientation 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 coeur du moteur. Elle stocke les commandes dans une instance de `CommandSet`.

Lorsqu'une nouvelle époque démarre, ie lorsqu'on a appelé la commande « fin de tour », 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.

Notons bien que ce ne sont pas les règles du jeu : chaque instance de ces classes applique la modification qu'elle contient, sans se demander si cela a un sens.

## **4.5 Conception logiciel : extension pour l'IA**

## **4.6 Conception logiciel : extension pour la parallélisation**

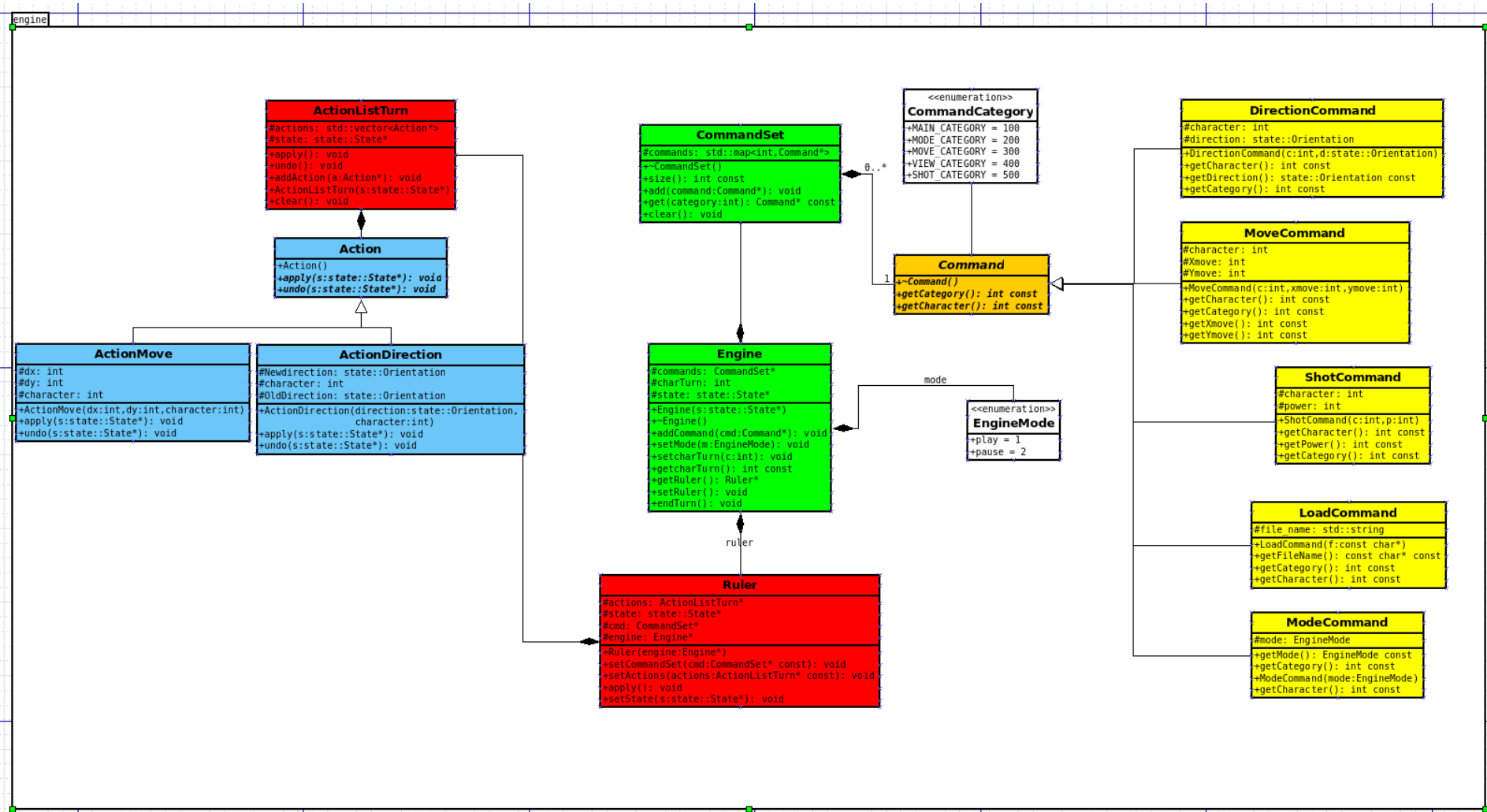


Illustration 5: Diagrammes des classes pour le moteur de jeu

## **5 Intelligence Artificielle**

*Cette section est dédiée aux stratégies et outils développés pour créer un joueur artificiel. Ce robot doit utiliser les mêmes commandes qu'un joueur humain, ie utiliser les mêmes actions/ordres que ceux produit par le clavier ou la souris. Le robot ne doit pas avoir accès à plus information qu'un joueur humain. Comme pour les autres sections, commencez par présenter la stratégie, puis la conception logicielle.*

### **5.1 Stratégies**

#### **5.1.1 Intelligence minimale**

#### **5.1.2 Intelligence basée sur des heuristiques**

#### **5.1.3 Intelligence basée sur les arbres de recherche**

### **5.2 Conception logiciel**

### **5.3 Conception logiciel : extension pour l'IA composée**

### **5.4 Conception logiciel : extension pour IA avancée**

### **5.5 Conception logiciel : extension pour la parallélisation**

## 6 Modularisation

*Cette section se concentre sur la répartition des différents modules du jeu dans différents processus. Deux niveaux doivent être considérés. Le premier est la répartition des modules sur différents threads. Notons bien que ce qui est attendu est une parallélisation maximale des traitements: il faut bien démontrer que l'intersection des processus communs ou bloquant est minimale. Le deuxième niveau est la répartition des modules sur différentes machines, via une interface réseau. Dans tous les cas, motivez vos choix, et indiquez également les latences qui en résulte.*

### 6.1 Organisation des modules

#### 6.1.1 Répartition sur différents threads

#### 6.1.2 Répartition sur différentes machines

### 6.2 Conception logiciel

### 6.3 Conception logiciel : extension réseau

### 6.4 Conception logiciel : client Android

*Illustration 6: Diagramme de classes pour la modularisation*



