Compte-Rendu de Projet Logiciel Transversal

Zhang Wenyuan - Denamganaï Kevin

29 octobre 2015

TABLE DES MATIÈRES

TABLE DES FIGURES

Première partie

OBJECTIF

1

PRÉSENTATION GÉNÉRALE

1.1 INSPIRATION

L'objectif de ce projet est la réalisation d'un jeu proche du jeu fictif ***KillBallad*** présenté dans l'animé **Robotics ;Notes** (cf. *Figure* 1). Ce jeu a pour particularité de faire intervenir un gameplay très proche de la commande d'un robot humanoïde. Ce sera un aspect central dans notre projet sans pour autant oublier l'aspect tour par tour.

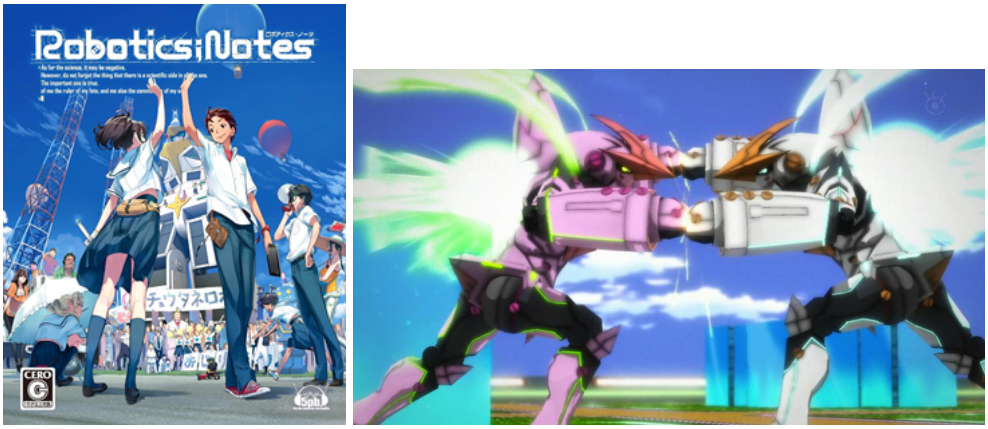

FIGURE 1 – Illustration de l'animé **Robotics ;Notes** et du jeu *KillBallad*.

1.2 RÈGLES DU JEU

Initialement, deux modes de jeu sont prévus :

— Mode **Course** : le joueur doit amener son personnage/robot jusqu'à la ligne d'arrivée le plus rapidement possible (en terme de nombre de tour joué), en évitant les obstacles sur sa route. Il pourra y avoir présence ou non d'un ou plusieurs adversaires.

— Mode **Combat** : Cela consiste en un combat à la façon des Sumotoris, le joueur faisant face à un adversaire, son but et de contrôler son personnage/robot de sorte à appliquer des forces qui vont déséquilibrer son opposant et le faire tomber. La chute donnant le perdant.

Toutefois, dans le but de ne pas trop compliqué le projet, nous commencerons par nous concentrer sur le Mode **Course** qui présente le moins de difficultés.

1.2.1 *Toribash*

Nous ne connaissions pas ce jeu à l'origine et l'avons découvert vers le début de l'implantation du projet. Notre projet, en ce qui concerne le Mode **Combat**, est similaire au jeu **Toribash**.

FIGURE 2 – Illustration "*in game*" du jeu **Toribash**.

Nous nous en inspirerons principalement concernant le *gameplay* relatif aux commandes du personnage/robot.

Deuxième partie

DESCRIPTION ET CONCEPTION DES ÉTATS

2

DESCRIPTION DES ÉTATS

Un état du jeu peut se décomposer autour d'éléments fixes et d'éléments mobiles qui, au départ, sont tous des éléments comportant les attributs suivants :

— la Pose : $x \in \mathbb{SE}(3)$, l'espace spécial euclidien de dimension 3, qui décrit entièrement la position et l'attitude de l'objet.
— un identifiant qui permet de déterminer le type (de classe) de l'élément.

2.1 ELÉMENTS FIXES

L'environnement est composé d'un sol et d'obstacle à franchir. Le milieu de taille limité, prédéfini selon la course choisie. Parmi les éléments fixes de l'environnement, on compte :

— les **Obstacles** : Il s'agit d'éléments de tailles variable franchissable ou non, selon la taille. Par exemple, le sol est un obstacle infranchissable car de taille maximale.
— les **Orbes Bonus** : qui peuvent donner des capacités telles que le fait de pouvoir briser/détruire des obstacles franchissables pendant un temps déterminé ou un nombre de coups possibles.

2.2 ELÉMENTS MOBILES

Au sein de l'environnement, en ne considérant toujours que le mode **Course**, les éléments mobiles sont les personnages/robots. Plus exactement, on va définir les personnages/robots comme une composition d'éléments mobiles de bases chaînés entres eux qui contiendront donc des attributs supplémentaires, nécessaire au bon fonctionnement du moteur physique qui devra gérer les évolutions de ces élèments mobiles, telles que :

— un torseur cinématique : une résultante $\vec{\Omega} \in \mathbb{R}^3$, vecteur vitesse de rotation et un vecteur vitesse d'un point $P \in S$ le solide/élément mobile, $\vec{v}_P \in \mathbb{R}^3$
— un torseur dynamique : une résultante $\vec{A} \in \mathbb{R}^3$, vecteur quantité d'accélération et un moment dynamique en un point $P \in S$ le solide/élément mobile, $\vec{\delta}_P \in \mathbb{R}^3$
— la liste (map) des liaisons cinématiques de l'élément avec d'autres éléments.
— ...

Pour le moment, comme on le verra dans le diagramme de classe, les choix à l'égard du moteur physique n'étant pas encore arrêtés, cette partie du moteur d'états n'est pas encore décidée.

En ce qui concerne le bonus, notre personnage/robot va pouvoir acquérir des capacités pendant une durée déterminée en nombre de tour parmi les suivantes :

— devenir plus grand : et ainsi se mouvoir plus facilement, par exemple car des obstacles infranchissable par le passé à cause de leur taille deviendraient maintenant franchissable.
— obtenir la capacité à briser/détruire les obstacles.
— ralentir l'adversaire pendant un certain nombre de tour.

2.3 ETAT GÉNÉRAL

A l'ensemble des éléments fixes et mobiles, on rajoute des attributs tels que :

— Le compteur d'orbes bonus encore présentes dans l'environnement.
— Une horloge qui compte le temps qui s'est écoulé afin que les intégrations des mouvements de chaque solide les uns par rapport aux autres soient toutes synchrones.
— Une vitesse à laquelle l'horloge du jeu se met à jour par rapport au nombre de tour et/ou au temps réel.

3

CONCEPTION LOGICIELLE

Le diagramme des classes pour les états est présenté dans la *Figure 3*, dont nous pouvons mettre en évidence les particularités suivants :

3.1 COMPOSITION D'ÉLÉMENT

Les éléments mobiles, étant uniquement les personnages/robots, il serait logique d'avoir une gestion des parties de chaque personnage/robot sous la forme d'une composition. C'est la raison pour laquelle le *pattern composit* est utilisé.

3.2 FABRIQUE D'ÉLÉMENT

Les éléments fixes, dont nous avons besoin de créer un grand nombre afin de construire les pistes sur lesquels les courses auront lieu(en se référant au mode de course), seront générés à l'aide du *pattern factory* afin de répondre efficacement à ces besoins de génération en grand nombre..

3.3 ENVIRONNEMENT

La classe **Environnement** est la classe par laquelle on peut gérer facilement les éléments. Elle nous permet de supprimer et ajouter certains éléments pendant le déroulement du jeux et en même temps d'accéder facilement à ces élèments dans le but de mettre a jour leurs attributs le cas échéant.

3.4 ATTRIBUTS DE LA CLASSE IELEMENTMOBILE EN FONCTION DU MOTEUR PHYSIQUE

Ne sachant pas encore tous les détails concernant le moteur physique, il n'est pas encore possible de savoir quelles seront les attributs nécessaires pour la gestion des mouvements des éléments mobiles. De plus ce moteur physique est susceptible de changer, dès lors le *pattern strategy* rendra cette évolution plus facile au niveau des états.

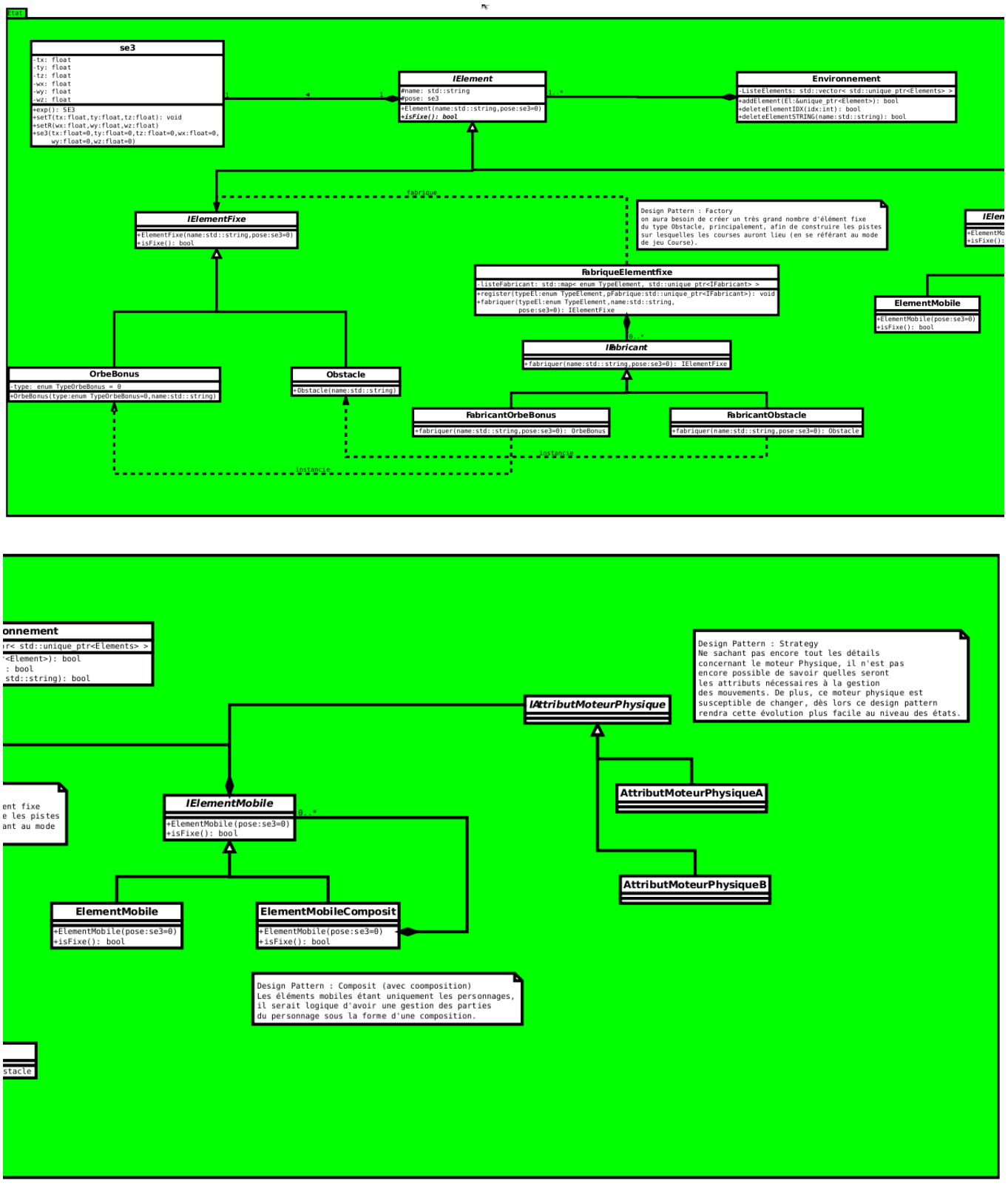

FIGURE 3 – Diagramme de classe des états

Troisième partie

RENDU : STRATÉGIE ET CONCEPTION

4

STRATÉGIE DE RENDU D'UN ÉTAT

4.1 PATTERN : MODÈLE-VUE-CONTRÔLEUR

En se plaçant dans le cadre d'un *pattern Model-View-Controler*, la partie ici présente concerne principalement la *Vue*. On peut distinguer plusieurs objectifs de rendu :

— **Monde3D** : représenter le monde 3D avec lequel le joueur intéragis et qui se base sur les informations du *Modèle* pour être généré.
— **ActionneursMonde3D** : représenter les actionneurs avec lesquels le joueur influence le *Modèle*.
— **Menus** : représenter des actionneurs au format de menus permettant la navigation entre les différents modes de jeu ainsi que de régler des paramètres de jeu, potentiellement.

Nous reviendrons sur ces objectifs de rendus dans la partie 5.

4.2 GESTION DU TEMPS

L'**État** du jeu ne changeant qu'entre les tours de jeu, puisque le **Moteur Physique** n'est mis à jour que entre les tours de jeu, il sera nécessaire que la partie *Vue* soit rafraîchie à la même vitesse que l'**État** uniquement dans ces moments là. Une horloge à 30-60Hz sera donc nécessaire ici. Ensuite, durant les tours de jeu, il n'y a aucune contrainte mis à part que la *Vue* doit être réactive aux modifications engendrées par la partie *Contrôleur*. Donc, en n'espérant que ça ne surchargera pas le processeur, on pourra rester sur une horloge à 30-60Hz au niveau de toutes les couches de rendu.

5

CONCEPTION LOGICIEL

5.1 EXTENSION DE L'ÉTAT

Se plaçant dans le cadre du *pattern Model-View-Controler*, la partie *Modèle* contient donc l'**État** et le **Moteur Physique** qui permet à ce premier d'évoluer. Dès lors, il sera nécessaire d'implémenter des *pattern Observer* depuis l'**État** vers le **Moteur Physique**.

5.2 VERS UNE CONCEPTION LOGICIEL

Les objectifs de rendus peuvent se séparer selon différentes couches de rendu. La couche la plus basse étant celle liée à l'objectif de rendu **Monde3D**, exclusivement gérée par le moteur 3D utilisant ***OpenGL***, suivi par l'objectif de rendu **ActionneursMonde3D** car celui-ci se trouve à l'interface entre le moteur 3D et le gestionnaire de fenêtre (ici la ***SDL*** 1.2) puisqu'il pourra contenir des éléments 3D ainsi que des textures "blittés" sur la fenêtre. Finalement, au dessus de tout cela, on trouvera les **Menus** qui seront donc exclusivement "blittés" à l'aide de la ***SDL***. Face au faible nombre de chose que vont contenir les courches **ActionneursMonde3D** et **Menus**, il semble être une perte de temps que dédier des classes à chacune de ces trois couches. Donc, jusqu'à ce que le besoin ne s'en sente réellement, tout sera géré au même niveau par une celle classe.

5.3 DIAGRAMME UML & DÉTAILS

Etant donné le *pattern Model-View-Controler*, le choix a été fait d'associer une classe hérité du type *IEngine* à chacun d'eux. Ici, nous sommes intéressé par la classe *VueEngine* qui s'occupe de la partie *Vue*. Celle-ci traitera donc entièrement les trois couches de rendu précédemment évoquées.

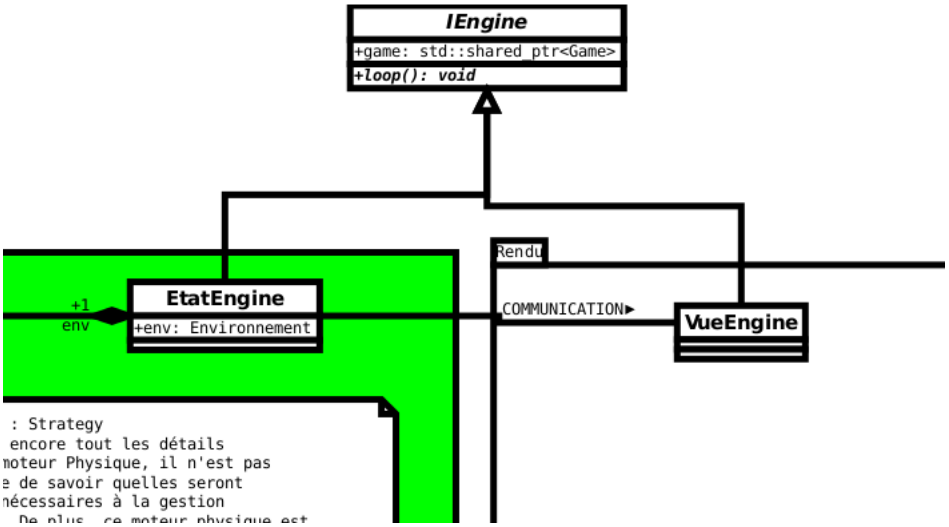

FIGURE 4 – Diagramme de classe détaillant les *IEngine*s et les prémisses de la partie *Vue*.

BIBLIOGRAPHIE