

Compte-Rendu de Projet Logiciel Transversal

Zhang Wenyuan - Denamganai Kevin

3 décembre 2015

TABLE DES MATIÈRES

i	OBJECTIF	4
1	PRÉSENTATION GÉNÉRALE	5
1.1	Inspiration	5
1.2	Règles du jeu	5
1.2.1	Toribash	5
ii	DESCRIPTION ET CONCEPTION DES ÉTATS	7
2	DESCRIPTION DES ÉTATS	8
2.1	Éléments fixes	8
2.2	Éléments mobiles	8
2.3	Etat général	9
3	CONCEPTION LOGICIELLE	10
3.1	Composition d'élément	10
3.2	Fabrique d'élément	10
3.3	Environnement	10
3.4	Attributs de la classe IElementMobile en fonction du moteur physique	10
iii	RENDU : STRATÉGIE ET CONCEPTION	13
4	STRATÉGIE DE RENDU D'UN ÉTAT	14
4.1	Pattern : Modèle-Vue-Contrôleur	14
4.2	Gestion du temps	14
5	CONCEPTION LOGICIEL	15
5.1	Extension de l'état	15
5.2	Vers une conception logiciel	15
5.3	Diagramme UML & détails	15
iv	RÈGLES DE CHANGEMENT D'ÉTATS ET MOTEUR DE JEU	17
6	HORLOGE GLOBALE	18
7	CHANGEMENTS EXTÉRIEURS	19
8	CHANGEMENTS AUTONOMES	20
9	CONCEPTION LOGICIELLE	21
10	CONCEPTION LOGICIELLE : EXTENSION VERS LA PARALLÉLISATION	23
v	INTELLIGENCE ARTIFICIELLE	24
11	STRATÉGIE	25
11.1	IA minimale	25
11.2	IA moyenne : recherche dans un graphe	25
11.3	IA avancée : déplacements optimaux	25
12	APPROCHES	26
12.1	Génération de trajectoire	26
12.2	Déplacement du personnage/robot	26
13	CONCEPTION LOGICIELLE	27

TABLE DES FIGURES

FIGURE 1	Illustration de l'animé Robotics;Notes et du jeu <i>KillBallad</i> .	5
FIGURE 2	Illustration " <i>in game</i> " du jeu Toribash .	6
FIGURE 3	Diagramme de classe des états, partie 1.	11
FIGURE 4	Diagramme de classe des états, partie 2.	12
FIGURE 5	Diagramme de classe détaillant les <i>IEngines</i> et les prémisses de la partie <i>Vue</i> .	16
FIGURE 6	Diagramme de classe en rapport avec les commandes dont les traitements entraînent les changements externes.	22

Première partie

OBJECTIF

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.

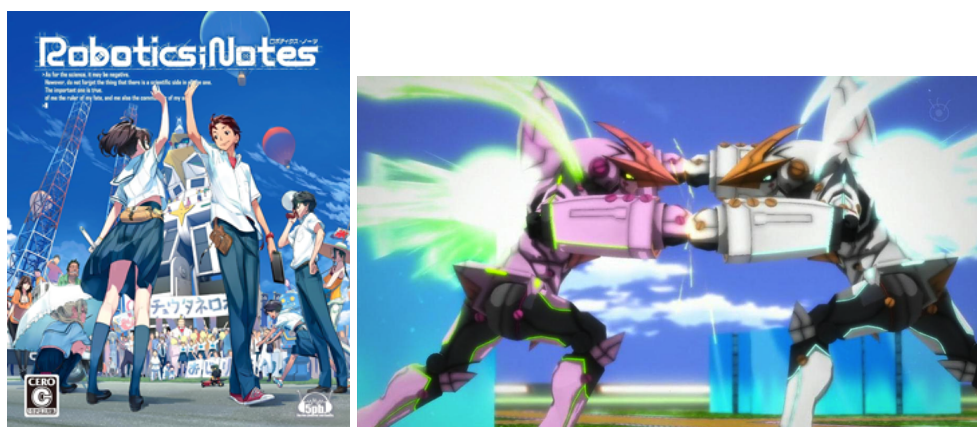


FIGURE 1 – Illustration de l'animé *Robotics;Notes* et du jeu *KillBallad*.

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 est 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 compliquer le projet, nous commencerons par nous concentrer sur le Mode **Course** qui présente le moins de difficultés.

1.2.1 *Toribash*

Nous ne connaissons 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

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 ÉLÉMENTS 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 ÉLÉMENTS 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 éléments 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.

CONCEPTION LOGICIELLE

Le diagramme des classes pour les états est présenté dans les *Figures 3* et *4*, 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 compos* 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 jeu et en même temps d'accéder facilement à ces éléments dans le but de mettre à 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.

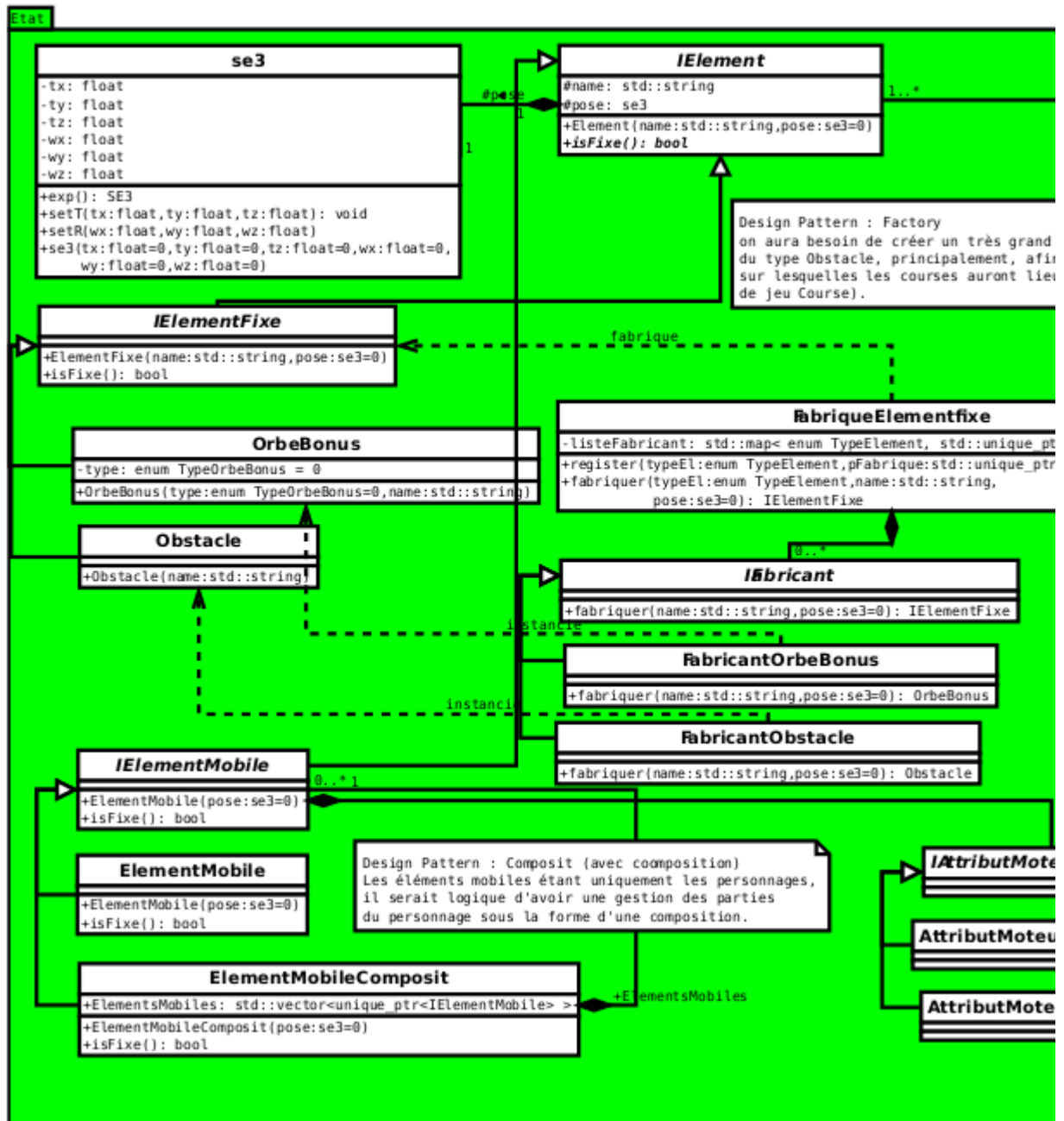


FIGURE 3 – Diagramme de classe des états, partie 1.

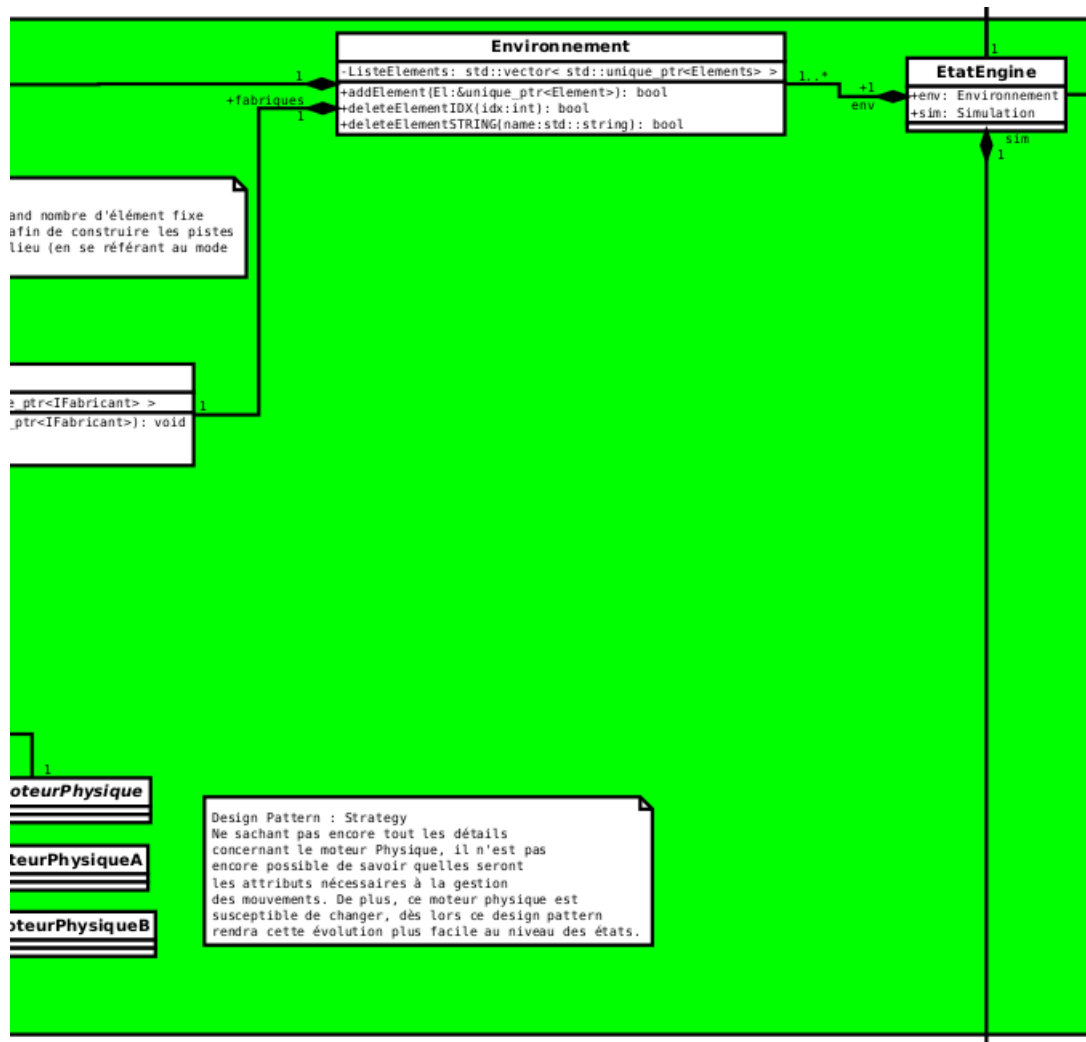


FIGURE 4 – Diagramme de classe des états, partie 2.

Troisième partie

RENDU : STRATÉGIE ET CONCEPTION

STRATÉGIE DE RENDU D'UN ÉTAT

4.1 PATTERN : MODÈLE-VUE-CONTRÔLEUR

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

- **Monde3D** : représenter le monde 3D avec lequel le joueur interagit 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 qu'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 mise 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.

CONCEPTION LOGICIEL

5.1 EXTENSION DE L'ÉTAT

Se plaçant dans le cadre du *pattern Model-View-Controller*, 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 couches **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 seule classe.

5.3 DIAGRAMME UML & DÉTAILS

Etant donné le *pattern Model-View-Controller*, 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.

5.3 DIAGRAMME UML & DÉTAILS

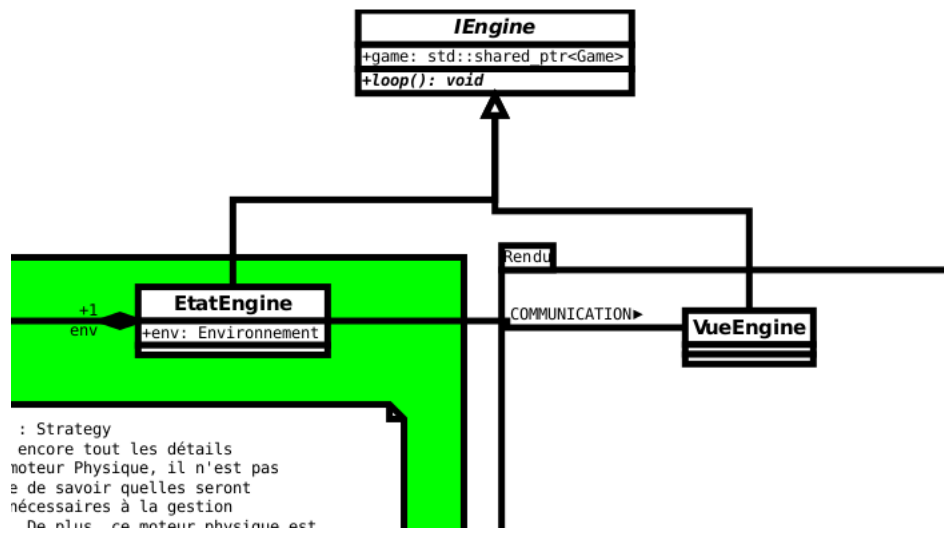


FIGURE 5 – Diagramme de classe détaillant les *IEngines* et les prémisses de la partie *Vue*.

Quatrième partie

RÈGLES DE CHANGEMENT D'ÉTATS ET MOTEUR DE JEU

HORLOGE GLOBALE

Concernant la simulation physique entre deux tours de jeu, celle-ci sera faite avec une certaine discrétisation temporelle qui, dans un premier temps pour des raisons de simplicité, sera fixe mais qui par la suite pourra être variable si le besoin s'en fait sentir afin de la simulation soit plus adéquate, par exemple, à la réalité.

Là où il se pose est un réel choix est de savoir si la durée de simulation entre deux tours de jeu doit être fixe ou non. En effet, on pourrait imaginer par exemple que la simulation pourrait s'arrêter nécessairement dès qu'une collision entre un élément de jeu et le personnage/robot apparaît, afin d'octroyer aux joueurs des moments de jeux cruciaux. C'est une évolution qu'il nous faudra envisager mais pour le moment, toujours dans un objectif de simplicité, nous fixerons un temps de simulation fixe entre 0.5 seconde et 1.0 seconde. En dehors des phases de simulation physique, il n'y a aucune évolution spontanée de l'état du jeu. On ne fera que enregistrer les commandes des joueurs et modifier les paramètres de la simulation physique en conséquence mais l'état ne changera pas. Donc, aucune mention vis-à-vis du temps à gérer.

CHANGEMENTS EXTÉRIEURS

Les changements extérieurs ici seront provoqué par les commandes mises à la disposition des joueurs. On noteras :

— Commandes MENU :

1. "Charger un niveau" : mise en place d'un terrain de jeu avec ses obstacles et orbe bonus, à partir d'un fichier.
2. "Nouvelle partie" : en assumant le chargement d'un niveau au préalable, par défaut ou non, tout est remis à l'état initial.
3. "Sauvegarder la partie" : en assumant qu'une partie est en cours, on sauvegarde l'intégralité de l'état du jeu dans un fichier.
4. "Charger une partie" : à partir d'un fichier de sauvegarde, on charge l'état de jeu sauvegarde et lance le jeu.

— Commandes MODE :

1. "Pause" : mise en pause du jeu.
2. "Reprendre la partie" : reprise après mise en pause.
3. "Quitter la partie" : retour au menu principal.

— Commandes JEU : répartie entre des commandes qui vont influencer le Modèle et d'autres qui vont influencer la Vue :

1. **Modèle** : Changer les états des articulations du personnage/robot : gérer à l'aide la souris ou de touches claviers réservés qui vont faire passer l'articulation d'un état à un autre parmi "Contracter", "Maintenir", "Relâcher" et "Tendre" ou encore "Tourner vers la droite", "Tourner vers la gauche" pour certaines articulation au niveau du buste.
2. **Vue** : Changer la pose de la caméra : gérer à l'aide de la souris également pour le contrôle d'un caméra du type "*Trackball*" centrée sur la position du personnage/robot.

CHANGEMENTS AUTONOMES

Les changements autonomes sont les résultats des changements extérieurs ainsi que de la simulation physique. Ils se produisent uniquement pendant la phase de simulation physique entre les tours de jeu. Nous ne détaillerons pas ici les détails du moteur physique étant donné la complexité mise en jeu. Toutefois, nous pouvons citer les sources principales et leurs apports pour détailler l'architecture du moteur physique. En premier lieu, on s'est référé à [1] en ce qui concerne l'architecture du moteur physique. Ensuite, au sein de cette architecture, on retrouve une partie chargée de la détection des collisions qui est inspirée de [2] et [3]. Ensuite, pour ce qui est des "solvers" chargés de trouver des solutions aux différentes contraintes posées afin de passer d'un état à un autre, on s'est largement inspiré de [4]. Vous trouverez dans l'annexe à ce rapport un fichier PDF nommé "*DiagrammeClasse-Etat-Physique-Rendu.pdf*" qui présente convenablement ce moteur physique.

CONCEPTION LOGICIELLE

Afin donc de gérer les seuls changements externes, les classes suivantes, héritantes toutes de **ICommand** , ont été ajoutées :

- **NewCommand**
- **LoadCommand**
- **SaveCommand**
- **LoadMapCommand**
- **PauseCommand**
- **GoOnCommand**
- **QuitCommand**
- **RigidBodyCommand**
- **CameraCommand**

Elles seront générées par la classe **Game** qui surveillera la souris et les appuies des touches du clavier, et lorsque l'ensemble précédent de **ICommand** aura été traité entièrement par les différents **IEngines** concernés, la classe **Game** se chargera de dispatcher convenablement les **ICommand** s qui étaient en attentes. C'est ensuite en interne des différents **IEngines** que les actions à réaliser seront traitées. La *Figure 6* présente le diagramme de classe en rapport.

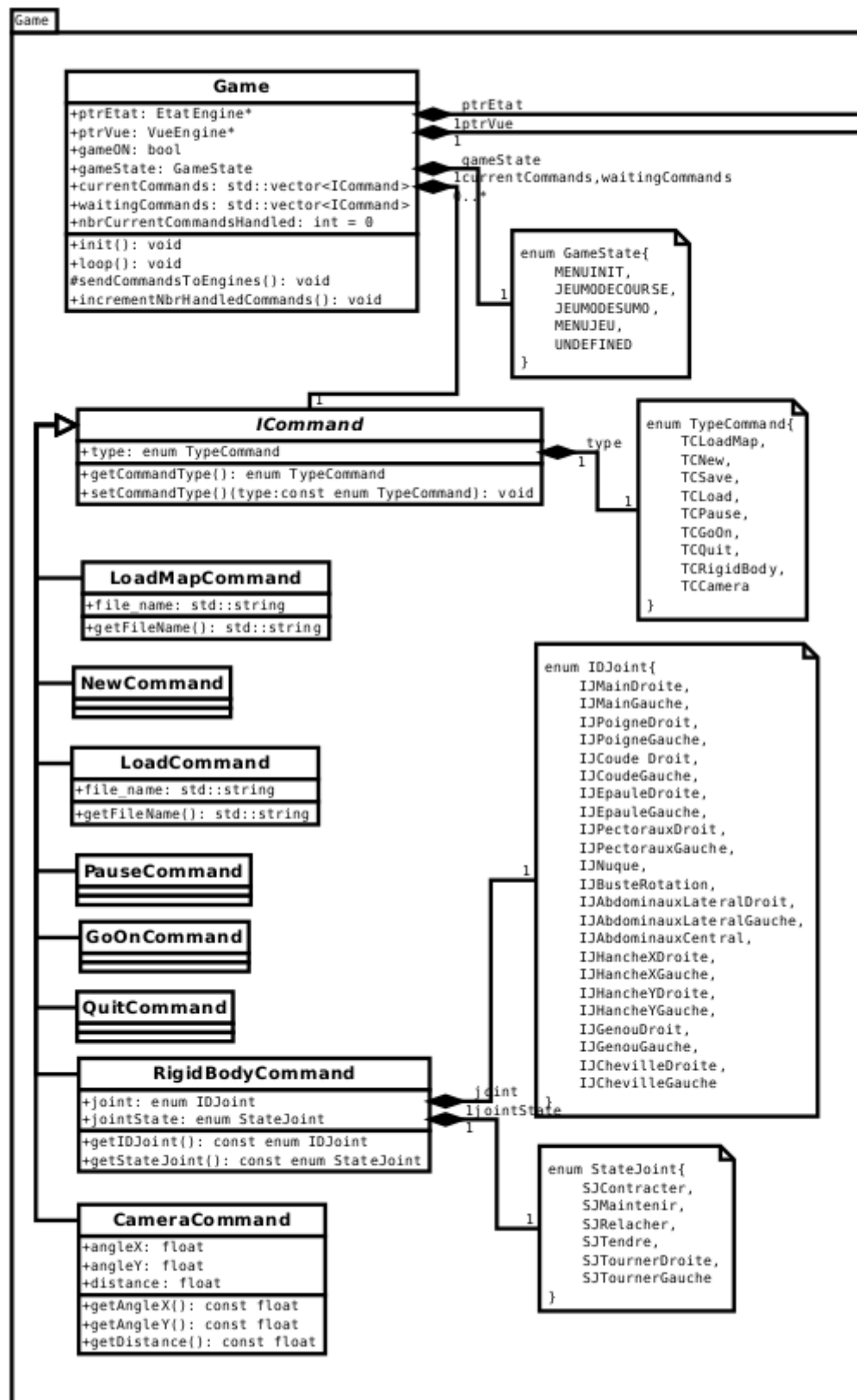


FIGURE 6 – Diagramme de classe en rapport avec les commandes dont les traitements entraînent les changements externes.

CONCEPTION LOGICIELLE : EXTENSION VERS LA PARALLÉLISATION

N'ayant pas encore de parallélisation des différents **IEngines**, l'observation des entrées est faite par la classe **Game** mais, par la suite, nous encapsulerons sûrement ce mécanisme dans un **IEngine** dédié qui sera exécuté sur un fil d'exécution qui lui sera propre. Egaleme nt, probablement que les différentes tâches et actions entraînés par ces flux de **ICommand** seront également gérées de manière *multithreadée*.

Cinquième partie

INTELLIGENCE ARTIFICIELLE

STRATÉGIE

Etant donné l'optique de compléter en premier lieu le mode de jeu "Course", nous ne parlerons ici que de l'IA associée à ce mode de jeu. L'objectif, pour un joueur, étant ici de déplacer son personnage/robot le plus rapidement possible en direction d'un but précis dans une carte, on peut identifier deux tâches critiques :

- Trouver une trajectoire à suivre, la plus **courte** possible, évitant les *Obstacles* et utilisant des *Orbes Bonus* afin de **s'avantager** au maximum face à l'adversaire.
- Déplacer son personnage/robot le plus **efficacement** possible en suivant la trajectoire trouvée précédemment.

On peut donc mettre en évidence des critères sur lesquels juger le niveau d'une IA :

- Qualité de la trajectoire générée : est-ce la trajectoire la plus courte ?
- Qualité de l'interaction avec l'environnement : la trajectoire comporte-t-elle des *Orbes Bonus* ?
- Qualité du déplacement du personnage/robot en terme de rapidité et de distance parcouru par action, par exemple.

Et du niveau de satisfaction de ces critères nous allons pouvoir dériver plusieurs niveaux d'intelligences artificielles.

11.1 IA MINIMALE

L'IA minimale se contentera de suivre une trajectoire en ligne droite allant de la position présente au but et évitant de manière *hasardeuse* les obstacles. Aucune interaction avec l'environnement ne sera faite et les déplacements seront lent.

11.2 IA MOYENNE : RECHERCHE DANS UN GRAPHE

L'IA moyenne pourra suivre une trajectoire bien meilleur car déterminée à l'aide d'une recherche dans un graphe discrétisant l'espace des positions de la projection sur le sol du centre de gravité du personnage/robot.

11.3 IA AVANCÉE : DÉPLACEMENTS OPTIMAUX

L'IA avancée sera une IA moyenne dont les déplacements du personnage/robot seront plus rapides et plus optimaux. L'interaction avec l'environnement sera possible, sous réserve de la proximité des *Orbes Bonus* avec la trajectoire optimale générée par la recherche dans le graphe explicité ci-avant.

APPROCHES

Il nous reste à expliciter les différentes approches que l'on pourra suivre afin de remplir les deux objectifs mentionnés plus tôt.

12.1 GÉNÉRATION DE TRAJECTOIRE

La mise en place d'algorithme tel que A^* ou, plus simplement, *Dijkstra* permettra une génération de trajectoire adéquate dont le degré de discrétisation de l'espace pourra être un paramètre important afin de niveler la qualité de la trajectoire générée selon les niveau d'IA, d'une part, et également ce pourra être un paramètre critique avec lequel il va falloir savoir jouer en terme de complexité algorithmique.

12.2 DÉPLACEMENT DU PERSONNAGE/ROBOT

Typique, lorsque l'on s'attaque au contrôle d'un robot humanoïde, les algorithmes évolutionnaires semblent assez indiqués. On pourra d'ailleurs imaginer que l'on pourra retenir des individus à différentes générations de populations afin d'obtenir différents niveaux d'IA relativement à ce critère de déplacement. Dans un premier temps, toutefois, on pourrait imaginer l'écriture en dure d'un contrôleur - sous la forme d'une suite de commande à appliquer, qui aurait été validées par un joueur expert au préalable - permettant de déplacer le personnage/robot de quelques pas et l'IA utiliserait cette abstraction avec des paramètres de directions et de vitesses à jauger selon ses besoins du moment.

CONCEPTION LOGICIELLE

BIBLIOGRAPHIE

- [1] Helmut G arstenauer. *A Unified Framework for Rigid Body Dynamics*. 2006.
- [2] Ian Millington. *Game Physics Engine Development*. 2007.
- [3] Christer Ericson. *Real-Time Collision Detection*. 2004.
- [4] Brian Vincent Mirtich. *Impulse-based Dynamics Simulation of Rigid Body Systems*. 1996.