VI50 : Souris 3D

Tamax & Ruby

Etudiants :
Alexandre Haffner
Nicolas Monneret
Wassim Sioud

Responsables :
Stéphane Galland
Olivier Lamotte

INTRODUCTION 3

CAHIER DES CHARGES 3

SUJET 3
OBJECTIFS 3
CONCEPT GENERAL DU JEU 3
FONCTIONNALITES 4
MODELISATION 4
TECHNOLOGIE 5

DEVELOPPEMENT 6

PRINCIPE GENERAL 6
ARCHITECTURE GENERALE 7
GESTION DES CONTROLEURS 10
GENERATION DE L'ENVIRONNEMENT 13
GAMEPLAY 14
LES OBJETS 14
Bumper 14
Booster 14
Slower 14
Stopper 14
LE TERRAIN 15
LE CIEL 15
SIMULATION DE LA CHUTE 16
DETECTION DU CRASH 16
INTERFACE UTILISATEUR : OVERLAYS 17
MODELES 3D ET ANIMATION 18
GESTION DU SON 20
PROBLEMES RENCONTRES 20
PERFORMANCES 20

GUIDE UTILISATEUR 21

BUT DU JEU 21
MENU 21
COMMANDES DU MENU 21
COMMANDES DU JEU 21
UTILISATION DU CLAVIER 22
UTILISATION DE LA SOURIS STANDARD 22
UTILISATION DE LA SOURIS 3D 22
OBJETS 22

COMPILATION 22

CONCLUSION 23

Introduction

Ce document de synthèse présente l'ensemble des fonctionnalités ainsi que le processus de conception du projet pour l'UV VI50. Le groupe qui a travaillé sur celui-ci se compose de trois membres : Nicolas Monneret, Alexandre Haffner et Wassim Sioud.

Le sujet choisis implique l'utilisation de la souris 3D Connexion. Nous avons choisis de l'utiliser dans le cadre d'une intégration à un jeu d'arcade basé sur le vol.

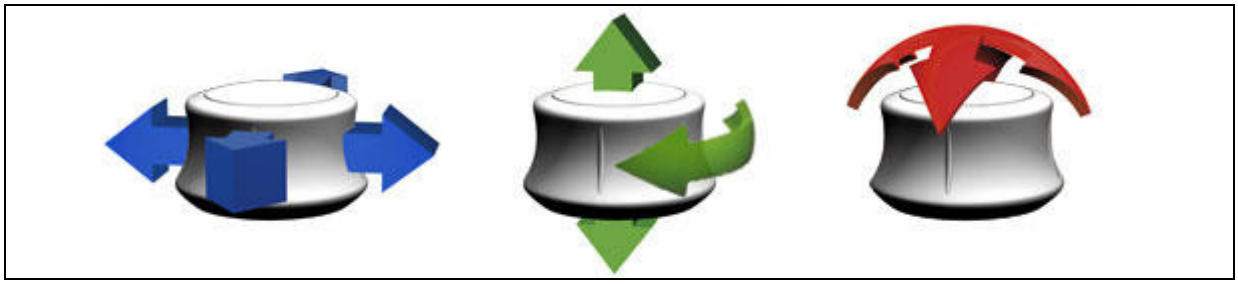
Cahier des charges

Sujet

Objectifs

Nous allons pour ce projet travailler le sujet concernant la souris 3D. Le principal objectif sera de tirer partie des fonctionnalités de la souris dans un environnement interactif en trois dimensions composé de différents modèles de personnages, d'objets, et d'environnement.

Il va nous falloir d'une façon ou d'une autre tirer partie des spécificités de la souris 3D, soit ses 6 axes de libertés visibles ci-dessous :



Afin de rendre le tout attractif autant pour l'utilisateur final que pour nous en terme de développement, nous allons réaliser un petit jeu de vol.

Concept général du jeu

Le but du jeu sera d'amener notre personnage volant le plus loin possible dans le niveau. La distance parcourue augmente le score plus ou moins rapidement en fonction des bonus acquis. Afin d'évoluer dans le monde virtuel celui-ci aura a sa disposition différents objets :

- Des propulseurs
- Des ralentisseurs
- Des multiplicateurs de score
- Des obstacles à éviter

Le personnage pourra décider de monter auquel cas il perdra de la vitesse, et inversement s'il décide de descendre. Il devra évoluer de propulseur en propulseur sans jamais toucher le sol. Lorsqu'il atteint le sol, la partie est terminée.

Fonctionnalités

Voici les différentes fonctionnalités que devra comporter le jeu :

- La gestion des collisions avec les objets de l'environnement pour faire bénéficier au personnage des effets désirés.
- Un comptage du score fonction des bonus.
- La génération aléatoire des niveaux de jeu dans un environnement rectiligne et infini.
- Plusieurs caméras :
    - Une vue première personne.
    - Une vue troisième personne (caméra TrackBall).
    - Une vue spéciale lors du passage sur certains objets (rétro par exemple).
- Le contrôle du personnage via la souris 3D (voir l'image ci-dessus). La camera sera indépendante du contrôle du personnage (distinction entre les mouvements de la tête et du corps) et l'utilisateur pourra bloquer la camera sur une position pour n'autoriser que les déplacement du personnage :
    - Les degrés de libertés bleus et rouges pour contrôler la caméra.
    - Les degrés de libertés verts pour contrôler le personnage.
    - Une touche clavier ou sur la souris (similaire au clavier) pour bloquer la camera.
- Le choix à l'utilisateur d'utiliser son clavier et sa souris standard, voir de paramétrer complètement les commandes.

Optionnellement nous pourrons implanter :

- De nouveaux objets pour le gameplay.
- Des effets graphiques évolués.
- Des sons.
- Plusieurs personnages.
- Un autre mini jeu basé sur le premier.
- La gestion des replay (avec mode fantôme par exemple).
- …

Modélisation

Nous utiliserons 3D Studio Max pour modéliser :

- Notre environnement : le sol, les objets avec lesquels le personnage rentrera en interaction.
- Le/Les personnages.
- Les animations : tourner à droite, à gauche, monter, descendre, crash contre un objet ou sur le sol…

Les personnages seront animés via animation par squelette, et les objets de l'environnement le seront éventuellement par morphing.

Technologie

- Le développement se fera en C++.
- Le moteur de rendu sera Ogre.
- Nous utiliserons le SDK C++ pour la souris 3D.
- 3D Studio Max dans sa version 2009 pour bénéficier de l'exporteur OgreMax. Nous fournirons au choix une conversion des fichiers au format 3D Studio Max 6 ou bien le viewer OgreMax permettant de visualiser l'ensemble des models et des animations.

Développement

Nous allons suivre dans cette partie notre conception du problème et les outils mis en œuvre pour le résoudre. On constatera au fil des paragraphes ce qui, du cahier des charges, a été maintenu, et ce qui s'est avéré, au final être un mauvais choix.

Principe général

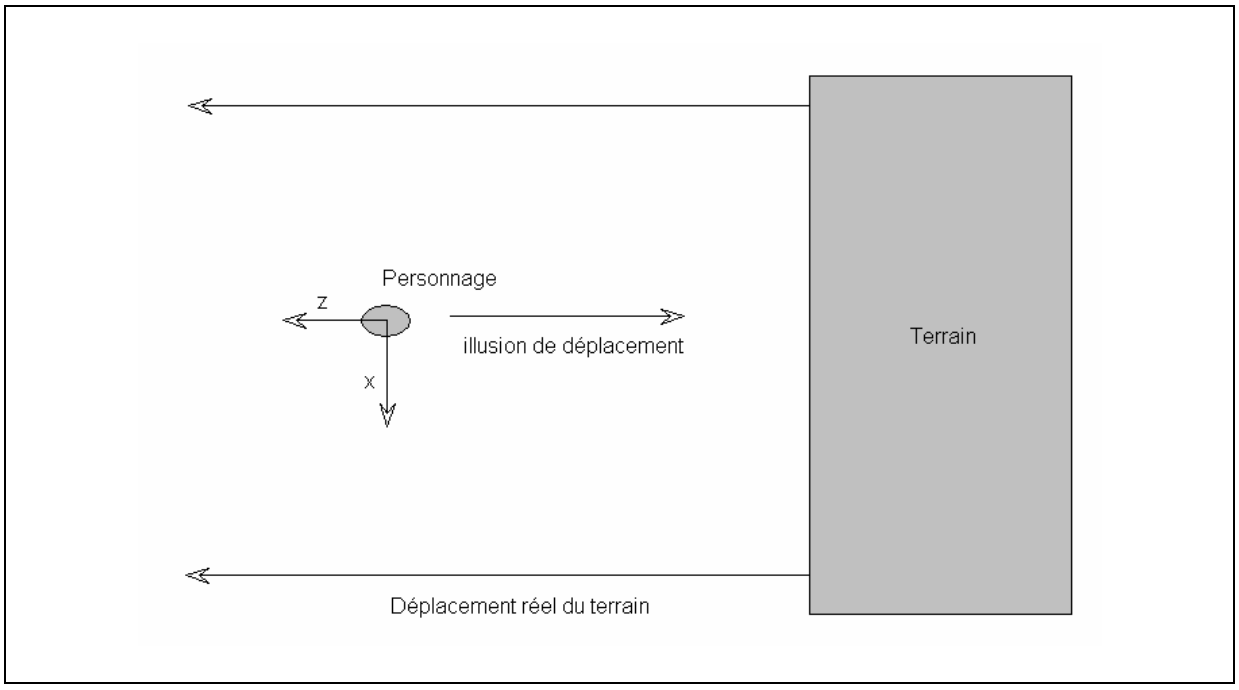
Le jeu que nous avons choisi de développer est comme nous l'avons vu précédemment un jeu arcade de vol. Le gameplay que nous voulions obtenir était celui d'un personnage volant dans un environnement infini, un environnement se générant automatiquement au fil du jeu.

Pour ce faire, nous nous sommes basés sur un concept assez ancien qui consiste à faire se déplacer l'environnement vers le personnage au lieu de déplacer le personnage lui-même, ceci afin de donner l'illusion de son déplacement.

Nous avons en réalité mixé cette approche avec une approche plus conventionnelle faisant se déplacer le personnage. Si l'on considère le repère direct avec y la verticale (repère que nous utiliserons tout au long de ce document), notre personnage sera immobile en x et z, et ne se déplacera donc que sur l'axe vertical. L'environnement en revanche se déplacera dans le plan xz afin de donner l'illusion du mouvement aussi bien en ligne droite que lorsque l'on tourne. La combinaison des deux nous donne donc un déplacement possible dans les 3 dimensions.

L'avantage principal de cette méthode est de pouvoir maintenir la création des objets de l'environnement dans un intervalle raisonnable autour du centre du monde, et de faciliter entre autre la gestion des caméras.

Si l'on observe la scène du dessus on a :

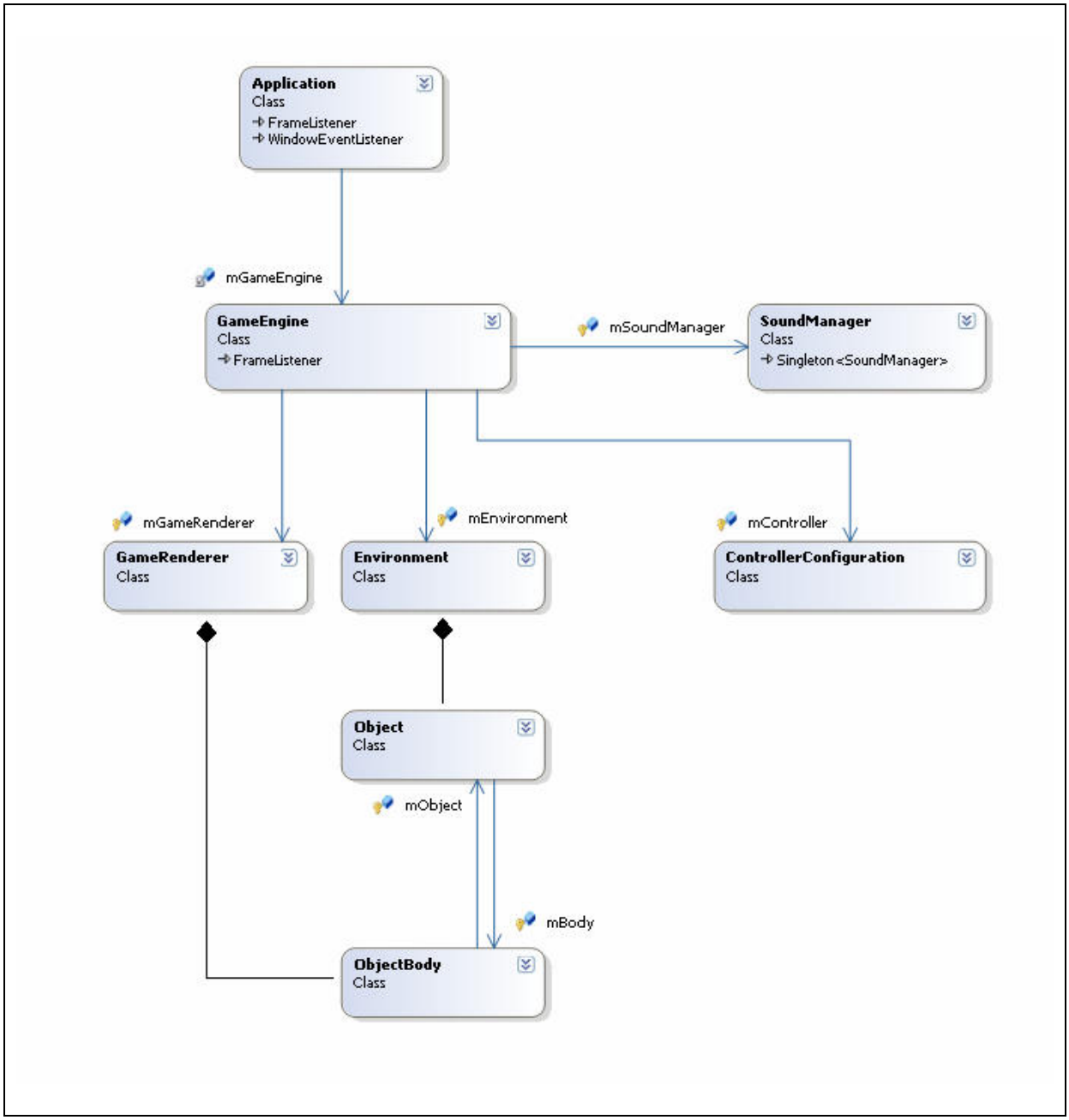


De même, on améliore l'immersion du joueur en faisant translater la texture du ciel.

Tous les objets générés sont concernés par ce principe, que ce soit le terrain ou les objets ayant trait au gameplay, qu'il nous faudra aussi bouger. On assurera une cohérence visuelle du terrain en ajustant quatre maillages ayant les mêmes jointures, afin de créer aléatoirement un pavage.

Architecture générale

Nous allons maintenant étudier l'architecture que nous avons adoptée et sur laquelle nous avons basé l'ensemble de notre travail. Voyons le diagramme de classe simplifié de l'application :

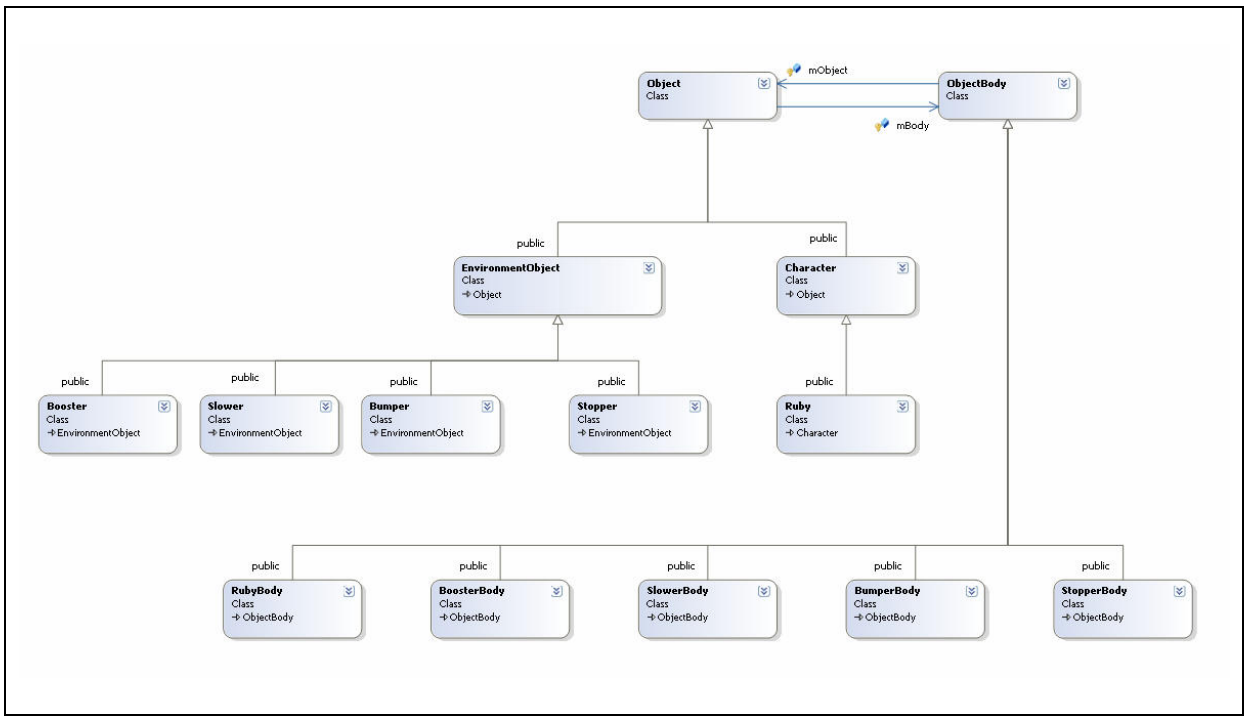


Nous avons basé notre application sur le moteur de rendu Ogre3D, lequel permet de multiples opérations de haut niveau. La boucle de rendu ogre est lancée par la classe Application, qui se charge en fait du démarrage du jeu, du fenêtrage, et de l'initialisation des différents composants. Elle contient entre autre la boucle de rendu principale (méthodes frameStarted / frameEnded) et va à chaque frame mettre à jour l'ensemble du système comme nous le verrons plus loin.

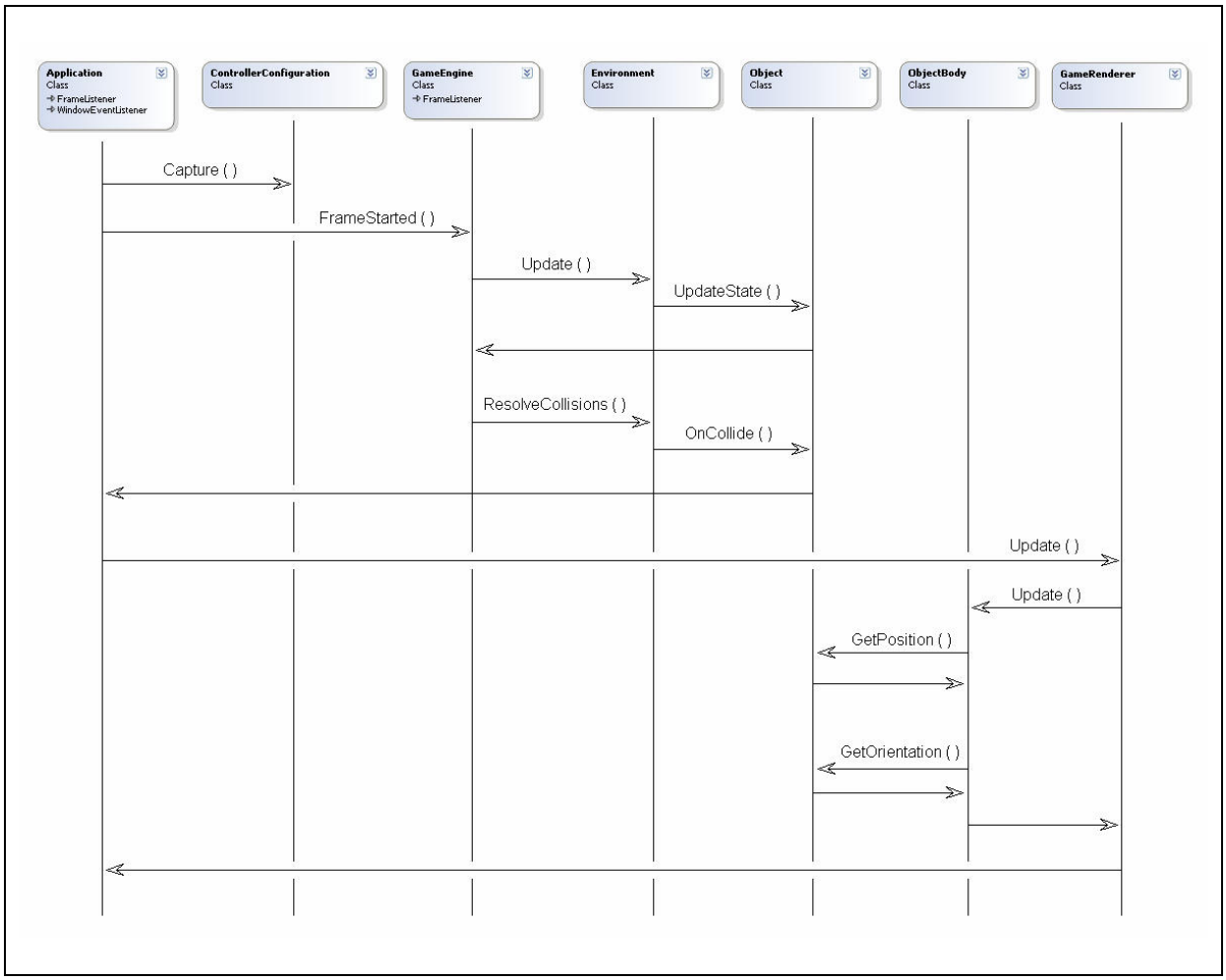
La class GameEngine (moteur de jeu) comprend l'ensemble des mécanismes de jeu tel que le mouvement du personnage, des objets en jeu (via la classe environnement), la mise à jours des contrôles, le calcul du score, … en bref l'ensemble de la partie gameplay. Elle travail directement ou indirectement sur un ensemble d'Object, lesquels sont des entités du monde, sans représentation, mais faisant parti du gameplay et donc pouvant posséder une position, une orientation, une vitesse…

La classe directement associée à celle-ci est la classe GameRenderer qui se charge en fait d'afficher l'ensemble de ce que la classe GameEngine va manipuler. C'est la partie graphique du processus, partie dont ne se charge pas le moteur de jeu. Cette classe travail sur des ObjectBody, représentations graphique des Object (le body ici n'est pas considéré dans le sens multi agent comme étant une passerelle vers l'environnement, il n'est que la représentation visuelle de l'objet). Ce sont ces ObjectBody qui vont décider quel maillage représentera quel objet, les animations qu'il comporte…

Notre application se base donc sur ces quelques classes, les dérivants au besoin (RubyGameEngine, RubyGameRenderer). Une partie importante du travail a été de créer les objets de l'environnement et de leur définir leur interaction avec le personnage. Voici l'ensemble des objets qui existent :



Voyons maintenant le diagramme de séquence général simplifié de l'application, afin de comprendre comment sont gérées les mises à jour de l'ensemble des structures :



A chaque image, la fonction FrameStarted de la classe Application est lancée. Celle-ci met alors à jour les contrôleurs (nous verrons un peu plus loin comment ceux-ci fonctionnent) ainsi que le moteur de jeu. Le GameEngine va demander à son Environement de mettre à jour l'état de l'ensemble des objets qu'il contient (leur position, leur vitesse, etc…) et effectuer la résolution de l'ensemble des collisions entre le personnage et les objets de l'environnement. La fonction OnCollide des objets est appelée s'il y a collision entre l'objet considéré et le personnage et se charge de lui appliquer les effets souhaités.

Notez que les collisions au niveau des objets (ce n'est pas vrai pour le terrain) sont gérées via des bounding box alignées sur les axes (AABB).

Une fois le moteur de jeu entièrement mis à jour, il reste à l'afficher. L'application appel la méthode update du GameRenderer lequel va mettre à jours l'ensemble des ObjectsBody, eux même allant chercher dans leur Object respectif les propriétés (position, orientation, animation, disparition, …) qu'il leur faudra représenter à l'écran.

A ceci s'ajoute la gestion du menu et la pause, la mise à jour du HUD et des caméras. La manipulation est à chaque fois basée sur le même principe.

Gestion des contrôleurs

La gestion des contrôleurs est une partie importante de notre projet, l'un de ses objectifs principaux étant de faire fonctionner la souris 3D dans ce contexte.

La souris 3D est en fait une souris ayant 6 axes : 3 rotations et 3 translations, conférant au périphérique la possibilité de contrôler à la fois la caméra et le déplacement du personnage.

Pour utiliser la souris en C++ nous avons utilisé le driver officiel dans sa dernière version. D'autres options étaient envisageables, comme par exemple extraire directement les valeurs des capteurs sur la souris sans passer par le pilote, option rapide à mettre en œuvre mais ayant pour inconvénient majeur de rendre le pilote complètement inutile et de ce fait d'empêcher toute possibilité de customisation des touches de la souris.

Nous avons donc travaillé suivant les préconisations de la marque pour l'utilisation de la souris, aussi bien en termes de programmation que de convention sur les repères.

Afin d'intégrer la gestion de la souris 3D à notre projet de façon intéressante, et surtout nous permettre de la combiner avec les autres périphériques dont nous disposons, à savoir le clavier et la souris standard, nous avons imaginé un système de configuration telle qu'il est présenté sur l'image suivante :

La classe centrale de ce système est l'AbstractController. Notre objectif a été dans un premier temps de généraliser l'ensemble de nos contrôleurs au sein de cette même classe abstraite.

Nous avons considéré qu'un contrôleur, quel qu'il soit, est capable de fournir en sortie, à tout moment, un ensemble de translations, un ensemble de rotations, et éventuellement des actions. Pour expliciter la chose, comparons les différents périphériques :

- La souris 3D de par sa nature offre 3 translations, 3 rotations, et des actions via ses touches.
- La souris standard peut offrir 3 translations en maintenant le bouton droit et en glissant (2) ajouté à la molette (1), 2 rotations en maintenant l'autre bouton et en glissant, et aucune action.
- Le clavier offre quand a lui 3 translations via les touches QSDZAE et 2 rotations via les touches Gauche Droite Haut Bas, ainsi que plusieurs actions (nous en avons défini 2).

Chacun de ces périphériques est défini dans les classes qui dérivent de la classe AbstractController et sont chargés de fournir en permanence les données en entrée au moteur de jeu chargé de gérer les déplacements du personnage (translations) et les mouvements de la caméra (rotations).

On constate la perte d'une rotation sur les 3, le « roll », soit la rotation suivant l'axe Z. Nous avons choisis de l'abandonner pour permettre l'utilisation de n'importe quelle combinaison de périphérique pour l'utilisation du jeu, mais aussi car c'est une rotation qui n'est pas fondamentale dans notre vision du jeu.

Ainsi il est possible de contrôler le personnage avec le clavier ou la souris, de gérer la camera avec la souris 3D ou le clavier…, en fait toutes les combinaisons de contrôleurs sont possibles. Ceci n'assure bien sur pas l'ergonomie et la facilité du système, aussi quelques choix s'avèreront plus judicieux que d'autres :

- Souris 3D pour le personnage, souris standard pour la caméra.
- Clavier pour le personnage, souris 3D pour la caméra.
- Clavier pour le personnage, souris standard pour la caméra.
- Souris 3D pour l'ensemble.

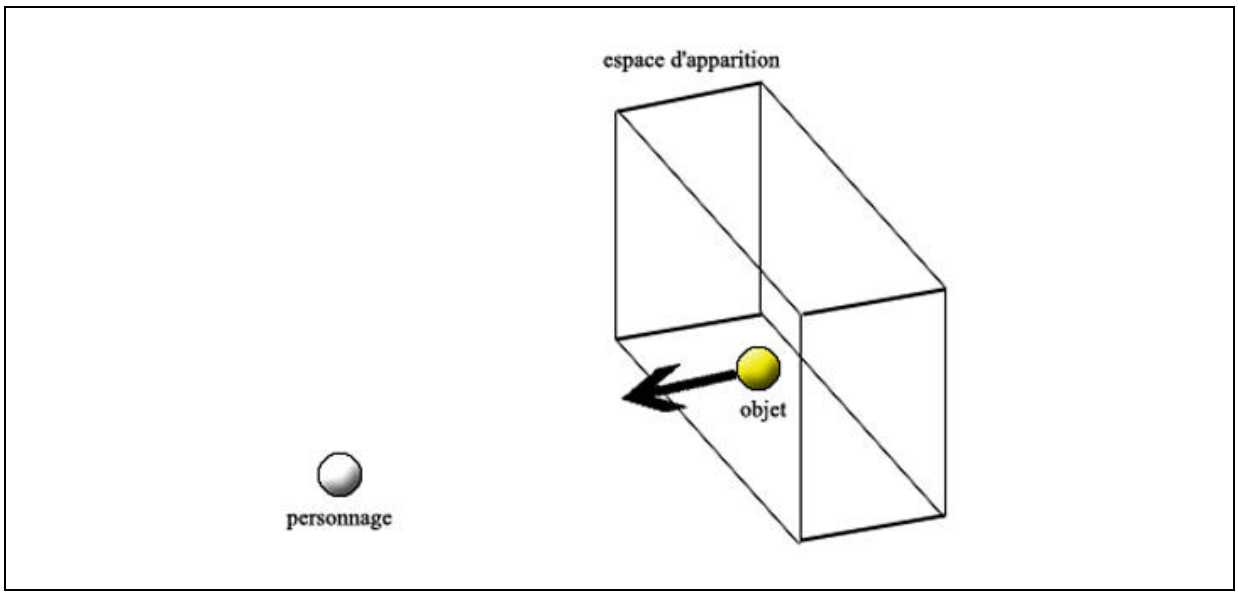
Ces différents contrôleurs sont regroupés au sein d'une configuration : ControllerConfiguration qui contient actuellement 3 contrôleurs, mais qui pourrait éventuellement en contenir plus. Cette classe se charge de la mise à jour (capture) unique sur chacun des contrôleurs distincts. En effet, capturer les touches du clavier ou de la souris plusieurs fois dans le même tour de boucle est source de problèmes. Cette classe simplifie donc la capture en s'assurant qu'un périphérique ne soit jamais mis à jour 2 fois par frame, mais assure aussi un regroupement de l'ensemble des contrôleurs en un seul endroit.

Chaque contrôleur de la configuration est assigné à une tâche précise. Dans notre application nous avons utilisé le contrôleur 1 pour les translations du personnage, le 2 pour la caméra, et le 3 pour la gestion de l'interface utilisateur, à savoir le menu et le changement de caméra.

La classe Com3dConnexion associée au contrôleur de la souris 3D est une surcouche maison de l'api permettant de gérer la souris 3D. Elle en simplifie l'accès et encapsule l'utilisation du composant COM… utilisation relativement fastidieuse…

Génération de l'environnement

C'est l'environnement qui créé les objets dans le ciel aléatoirement. Pour cela on définit une distance moyenne entre chaque génération d'objet (par exemple un objet tous les 200 mètres). Un objet sélectionné aléatoirement parmi ceux disponibles est créé et positionné dans un espace défini face au personnage à une certaine distance.



Cet espace est entièrement paramétrable via une série de propriétés dans le code. Son paramétrage influe directement le gameplay et le choix des valeurs doit être fait avec soin.

L'espace d'apparition des objets est toujours relatif au personnage sur les axes horizontaux. Si celui-ci se déplace sur la gauche ou la droite, il y aura toujours des objets qui apparaîtront en face de lui.

En revanche, cet espace possède une limite haute pour l'apparition des objets afin d'éviter au personnage de pouvoir monter indéfiniment en se servant des objets.

La taille de l'espace est suffisamment grande pour simuler le fait qu'il y a des objets dans tout l'espace aérien et pas seulement en face du joueur. Le joueur n'a donc pas l'impression d'être sur un chemin spécialement tracé pour lui.

Gameplay

Les objets

Tous les objets affectent le personnage lorsque celui-ci entre en collision avec eux. Bien que chaque objet dispose de ses propres effets, ils agissent de manière générale, sur l'accélération du personnage. Lorsqu'il y'a collision entre le personnage et un objet, un son spécifique à l'objet est joué de même qu'une animation pour certains objets.

Bumper

Le bumper simule un ressort et permet au joueur de prendre de l'altitude. Il donne une impulsion verticale vers le haut au personnage.

Booster

Le booster permet de faire accélérer le personnage en lui donnant une impulsion horizontale vers l'avant.

Slower

Le slower est l'inverse du booster et donne une impulsion horizontale vers l'arrière qui permet de faire ralentir le personnage.

Stopper

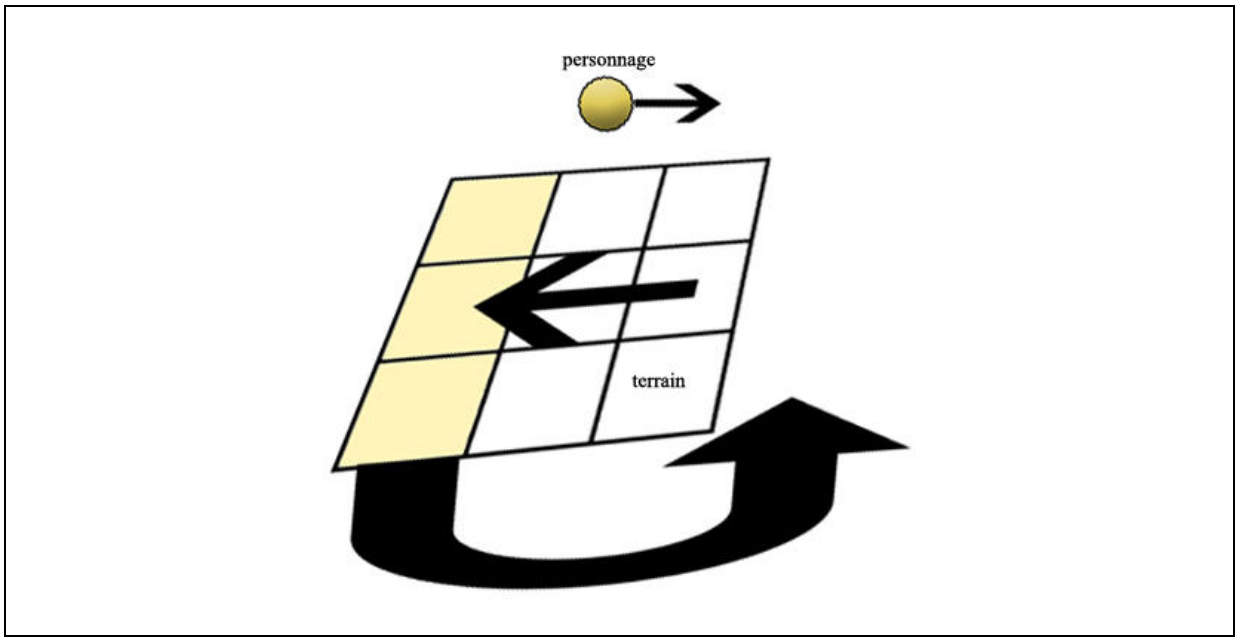
Le stopper est représenté par un mur, il s'agit d'un obstacle qui stoppe le personnage instantanément en mettant sa vitesse à zéro. Lorsque le personnage entre en collision avec cet objet, il tombe et la partie est terminée.

Le terrain

Dans le jeu, ce n'est pas le personnage qui se déplace réellement dans l'espace 3D, mais plutôt le terrain qui translate. Pour simuler un terrain de dimensions infinies, nous avons créé 4 meshs qui forment un pavage.

Lors de la création du terrain, un pavage aléatoire est créé avec les 4 meshs en respectant une pondération sur l'apparition des motifs.

Lorsque le personnage avance, nous déplaçons le terrain et effectuons un « roulement » en reprenant les meshs situés derrières et en les replaçant devant. On effectue la même opération lorsque le personnage se déplace à gauche et à droite.



Le ciel

A la manière du terrain, le ciel simule la progression du joueur dans le jeu. Il est constitué d'un plan situé à une altitude « infinie ». Ogre dispose par défaut de ce type de plan appelé skyplane. Pour simuler le mouvement, nous déplaçons ici la texture ciel sur le skyplane suivant les mouvements du personnage.

Simulation de la chute

Le comportement du personnage et la gestion de la physique sont gérés en utilisant des vecteurs d'accélération et de vitesse. On a ainsi des forces qui s'exercent sur le personnage et qui vont modifier son vecteur vitesse.

- La pesanteur qui est dirigée vers le bas et qui tend à attirer le personnage au sol.
- La portance qui est représentée par un vecteur force dirigé vers le haut. Elle est fonction de la vitesse horizontale du personnage vers l'avant. Quand le personnage a atteint une certaine vitesse horizontale, la portance compense totalement l'effet de la pesanteur.
- La répulsion du vent est un vecteur dirigé dans le sens inverse de la direction du personnage et qui tend à le ralentir. Elle est fonction du carré de la vitesse horizontale. Plus le joueur atteint des vitesses élevées, plus le vent aura tendance à le ralentir.

A ces 3 forces qui s'exercent sur le vecteur vitesse du personnage, on ajoute le contrôle du personnage par le joueur qui peut vouloir se déplacer en haut, en bas, à gauche ou à droite.

Nous gérons aussi le fait que le personnage prend de la vitesse horizontale lorsque le joueur demande à descendre et perde de la vitesse lorsque le joueur fait prendre de l'altitude au personnage.

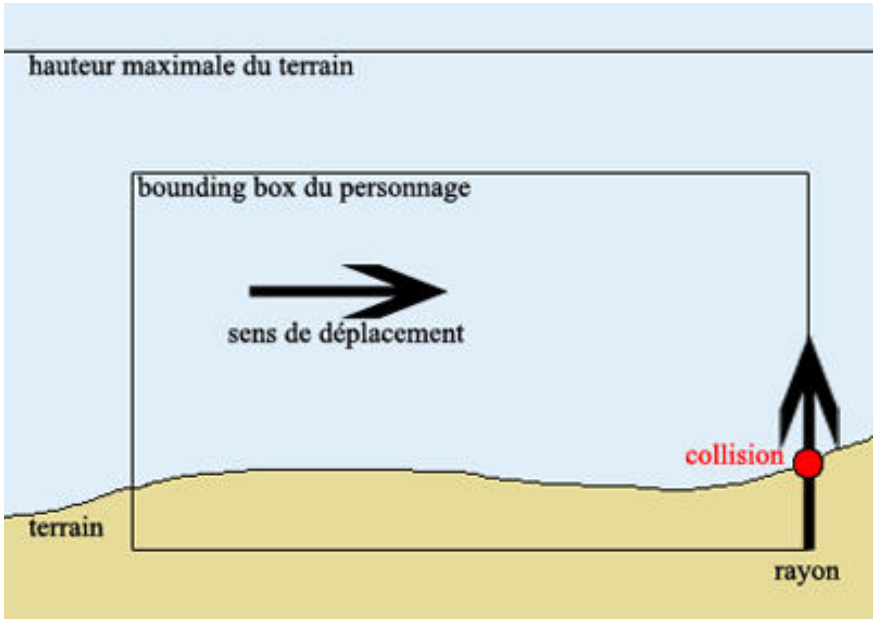
Détection du crash

Pour détecter la collision entre le personnage et le terrain et ainsi mettre fin à la partie, nous utilisons un lancer de rayon. Cette détection est réalisée en plusieurs étapes afin de minimiser les calculs à chaque frame.

- Une première étape consiste à tester si le personnage est au dessus de la hauteur maximum du terrain.

- S'il est dessous, on teste si la bounding box du personnage est en collision avec chacune des bounding box des meshs composant le terrain.

- Si on est en collision potentielle avec un mesh, nous lançons un rayon depuis le minimum de la bounding box du personnage vers le haut. Si ce rayon intersecte un des polygones du mesh, le personnage est donc sous le terrain. Le jeu stoppe et c'est la fin de la partie.



La détection de collision au niveau du polygone nous permet de créer un terrain avec beaucoup de dénivelé ou avec des obstacles comme des arbres. Le personnage a ainsi la possibilité de voler entre les arbres et au raz du sol.

Interface utilisateur : overlays

Ogre permet de définir des menus et des affichages à l'écran très rapidement grâce à des scripts externes appelés Overlays.

Nous avons utilisés 4 overlays :

- Le menu : Il est divisé en 2 parties et permet de sélectionner le contrôleur à utiliser pour la caméra et le contrôleur à utiliser pour diriger le personnage. Ces contrôleurs peuvent être le clavier, la souris ou la souris 3D. A partir du menu, le joueur peut aussi commencer une nouvelle partie ou quitter l'application.

- Le HUD : Il affiche les informations pertinentes en cours de partie : le score, la distance parcourue, la vitesse et l'altitude du personnage.

- Le masque en vue subjective : Il permet de simuler la vision qu'a le personnage derrière son masque.

- L'écran de fin de jeu : Il affiche le score final et la distance parcourue lorsque la partie est terminée.

Modèles 3D et animation

Une partie du projet a été consacré à la création de ressources, et entre autre les modèles 3D et leurs textures, ainsi qu'à la mise en œuvre de leur animation. Nous avons utilisé 3D studio max dans sa version 2009 pour bénéficier du plugin d'exportation vers les .mesh OgreMax.

Nous avons, dans la mesure du possible appliqué les règles de la modélisation polygonale, en gardant à l'esprit qu'il ne fallait pas abuser des polygones. Plusieurs essais on été nécessaire pour obtenir des models plus ou moins plaisants, ainsi que le passage par une phase d'apprentissage de l'animation avancée sous 3D studio max.

Nous avons utilisé différentes techniques :

- L'animation nodale, directement sous ogre.
- L'animation par squelette de nos personnages, et de certains objets (bumper).
- L'animation par morphing pour permettre au mesh de se déformer.
- La simulation de tissu pour générer l'animation de la cape de notre personnage.

Ainsi que plusieurs techniques de placage de textures récentes, comme le pelt mapping, en complément des techniques standard de développé et autre mapping UVW.

Ruby, modélisation polygonale, animation par squelette et simulation de tissu.

Un ange en converse, modélisation polygonale et morphing des ailes.

Gestion du son

Pour gérer le son, nous utilisons la librairie FMOD à travers une classe (trouvée sur le wiki Ogre) adaptée pour ogre : SoundManager.

Les sons sont spatialisés, nous avons donc défini un nœud du scenegraph comme étant le nœud « oreilles ». Chaque son créé est associé lui aussi à un nœud du scenegraph. Ceci permet de connaître la distance du son par rapport à l'oreille ainsi que sa vitesse et ainsi de générer les effets voulus (effet Doppler par exemple).

Nous avons intégré un son continu (le bruit du vent), et un son différent est associé à chaque type d'objet, qui se déclenche lorsque le personnage entre en collision avec lui.

Problèmes rencontrés

Nous avons été confrontés à différents problèmes tout au long de l'élaboration de ce projet. Nous ne connaissions ni Ogre, ni Fmod, et très peu sur 3D studio max. Les concepts abordés en VI50 et IN55 nous ont largement suffit à appréhender facilement Ogre, et la documentation officielle s'est avéré être une aide suffisante en ce qui concerne ogre.

L'exportation des models a en revanche posé des problèmes, par la méconnaissance du système, et les caprices des pivots !

Dans l'ensemble nous ne sommes pas restés bloqué longtemps.

Performances

La performance de notre jeu est à la mesure de notre connaissance d'Ogre. Bien qu'il soit souvent évident de savoir si une opération sera lourde ou non pour le temps réel, notre connaissance limité d'ogre nous a peut être amené à faire des choix non optimaux concernant ce projet ; avec notre gestion du terrain peut être.

Dans l'ensemble l'application semble tourner correctement sur des configurations vieillissantes. Nous avons pris soin d'appliquer au maximum les méthodes vues en cours, avec par exemple l'utilisation de LOD dans notre projet.

De la même manière le fait de débuter en modélisation polygonale nous a peut être amené à générer des modèles trop complexes en terme de géométrie (relativement à la qualité graphique).

Guide utilisateur

But du jeu

Le but du jeu est d'obtenir le meilleur score dans le temps imparti.
Le score est calculé en fonction de la distance parcourue et de la vitesse de vol.
Des bonus sont accordés si le personnage vole au raz du sol ou s'il dépasse la vitesse du son.

Menu

Nouveau jeu avec Tamax	Commencez une nouvelle partie avec le personnage Tamax (non disponible pour l'instant).
Nouveau jeu avec Ruby	Commencez une nouvelle parie avec le personnage Ruby.
Configuration	Cette option ouvre la page du menu qui vous permet de régler les contrôleurs.
Quit	Sélectionnez cette option pour quitter le jeu.
Character Device	Sélectionnez le contrôleur pour manipuler le personnage.
Camera Device	Sélectionnez le contrôleur pour manipuler la caméra.

Commandes du menu

Echap	Retourner au jeu.
Flèches	Se déplacer dans les options.
Entrée	Changer ou sélectionner une option.

Commandes du jeu

Echap	Mettre en pause et afficher le menu.
Espace	Changer de caméra.

Utilisation du clavier

Z	Faire perdre de l'altitude au personnage.
S	Faire prendre de l'altitude au personnage.
Q	Se déplacer vers la gauche.
D	Se déplacer vers la droite.

Haut	Baisser la caméra.
Bas	Monter la caméra.
Gauche	Tourner la caméra à gauche.
Droite	Tourner la caméra à droite.

Utilisation de la souris standard

Click gauche + déplacement	Déplacement de la caméra autour du personnage.
Click droit + déplacement	Translation du personnage.
Molette	Eloignement ou rapprochement de la caméra.

Utilisation de la souris 3D

Rotation autour de X	Rotation de la caméra (pitch).
Rotation autour de Y	Rotation de la caméra (yaw).
Rotation autour de Z	Inutilisée.
Translation en X	Déplacement du personnage en X.
Translation en Y	Déplacement du personnage en Y.
Translation en Z	Avancer/Reculer la caméra.
Bouton 2D	Bloquer les rotations (donc la caméra).

Objets

Accélérateur	Cet objet vous permet de prendre de la vitesse.
Bumper	Cet objet vous permet de prendre de l'altitude.
Ralentisseur	Cet objet vous ralentir, il faut donc l'éviter.
Vitre	Entrer en collision avec une vitre vous stoppera et vous fera perdre la partie.

Compilation

Pour compiler le projet vous aurez besoin des bibliothèques de fonctions Fmod et Ogre. Celles-ci sont disponibles sur leurs sites internet respectifs. Pour compiler le projet Visual C++ fournis, vous devrez définir la variable d'environnement FMOD_HOME avec le chemin vers l'api après avoir installé Fmod.

Nous avons mis en place système de compilation conditionnel pour permettre la compilation du jeu sans la souris 3D et ainsi gagner au chargement (en effet le composant cherche pendant

quelques secondes la souris sans la trouver au démarrage sinon). Pour ce faire il suffit de compiler en utilisant le define : NO_3DMOUSE.

Afin d'exécuter le programme, il est probable qu'il soit nécessaire d'ajouter une DLL si vous ne la possédez pas, et éventuellement les DLL de Visual Studio 2008 téléchargeables à cette adresse :

http://www.microsoft.com/downloads/details.aspx?familyid=A5C84275-3B97-4AB7-A40D-3802B2AF5FC2&displaylang=en

N'hésitez pas à me contacter en cas de problème.

Conclusion

Ce projet a clairement été l'un des projets les plus intéressants à mener à l'UTBM. Il est pour nous, friands et passionnés de jeux vidéo, une bonne mise en jambe pour notre futur d'ingénieur.

Nous avons eu l'occasion de toucher à l'ensemble du processus, tant au niveau modèles 3D que développement logiciel.

Nous aurions en revanche aimé pouvoir implanter un second petit jeu (Tamax) pour faire la paire Tamax&Ruby, mais nous avons manqué de temps.