ARCHITECTURE LOGICIELLE

Benoît Védrenne, Gaël Walter

18 janvier 2009

Chargé de TD : Damien Cassou

Université Bordeaux 1,
351 cours de la Libération,
33405 Talence Cedex,
France

The Cremi Legend Of Zelda

Résumé

La légende raconte que la grande forêt d'Hyrule est occupée par Ganon, le puissant Prince des Ténèbres. Ganon a capturé la fille du Roi qui tenta de séparer la Triforce, puissant sortilège qui protège le royaume d'Hyrule. Ses guardes maléfiques avancent à grand pas dans la forêt en détruisant tout sur leurs passages...
Il existe cependant un chevalier courageux, en ces temps de manants. On le reconnait facilement à sa chevelure blonde sous sa longue cape verte. Apprenant la nouvelle de l'invasion, le regard noir, il parti dans les haut bois de la forêt d'Hyrule pour les affronter tous et libérer la fille du Roi nommée "Zelda".

The "CREMI Legend of Zelda", c'est l'histoire de ce jeune garon, nommé Link qui doit sauver la princesse Zelda au péril de sa vie.
Une somme d'embûches l'attend. L'art de manier son épée pourra peut être l'aider dans sa quête impossible : seul contre une armée entière.

1 Introduction

Le framework de jeu plateforme dont nous disposions, nous a permis d'implémenter une version du Jeu Zelda. Ce jeu, créé par Miyamoto (auteur également de Donkey Kong et Mario), possède une version Nintendo de 1986.

Notre application s'inspire de cette version, où l'on peut guider dans une forêt le personnage Link. Nous l'avons bien sûr dénommé Cremi Legend Of Zelda car il a été principalement développé au Cremi !

1.1 Comment jouer à CLOZ ?

Le but du jeu est de guider Link dans la forêt pour l'amener à la princesse Zelda avec les touches fléchées.
On devra tout d'abord éliminer tout ses ennemis maléfiques en utilisant le coup d'attaque par la touche "Espace". Ceux ci sont très énervés car ils ont été prévenus de l'arrivé de Link, c'est pour cela qu'ils n'ont pas un comportement très logiques. Parfois, il se peut qu'un ennemi plus robuste surveille plus raisonnablement Zelda, il est donc plus difficile à tuer. Il ne faut pas hésiter à les frapper de manière répétée.

La difficulté du jeu réside dans le nombre d'ennemis à tuer, plus ou moins forts selon leurs grades, mais aussi dans les différents objets qui peuvent agrémenter le jeu. Le joueur est libre de tous les essayer. Mais comme dans la réalité, les bombes blessent.

La vie de Link démarre à "100" mais elle peut descendre très vite ! Faites attention Link, n'est pas immortel. En mourrant, vous recommencerez le jeu au début du niveau.

1.2 Quelques règles du jeu

Seul le clavier est utile au jeu, mais le menu permet d'autres actions.
Au croisement avec des ennemis, Link perd de la vie car il se fait frapper par ceux-ci. Les boss notament sont plus coriaces (de vrais brutes), et tapent plus fort. Si notre héros meurt, le niveau de jeu redémarre au début.
Depuis le menu, on peut redémarrer la partie au début, mais aussi sauvegarder la partie ou (en théorie) en restaurer une.
Lorsque tout les ennemis sont tués, il faut aller vers la princesse et le niveau est gagné, on passe alors automatiquement au niveau suivant, s'il existe. On peut trouver une arme sur le sol, Link peut attaquer avec. Il est possible que certaines actions enlèvent cette arme, et le combat de boss devient plus ardu.

1.3 Astuces

Comme dans beaucoup de jeu vidéo d'aventure, nous avons laissé des codes secrets que l'on peut chercher si l'on veut. Qu'il est agréable en étant joueur de trouver ce genre de code ! Mais n'oubliez pas que l'abus de codes nuit au plaisir de jouer.

2 Conception

La première phase du projet ft d'étudier le framework donné, ainsi que le code du jeu Pacman. Le framework nous permet de créer un jeu basé sur un univers donné, des entités évoluant dans ce monde selon des règles de collisions données. Nous avons commencé utiliser les classes du jeu Pacman pour débuter Zelda. Ce fût non seulement un moyen de comprendre plus vite comment utiliser certaines fonctionnalités du framework, mais aussi une difficulté de commencer vouloir adapter un jeu déjà fait à un autre, plutôt que de partir sur une base saine.
Ce choix nous amena à ajouter petit à petit les différentes fonctionnalités en plus que nous voulions.

2.1 Pacquetages

Au fur et mesure de l'implémentation, nous avons mis en place une arborescence de paquetages de classes, séparant bien chaque partie du code, ce qui nous a permis de nous retrouver plus facilement tout au long du projet. On retrouve "**base**", "**game**" comme dans le framework, mais aussi "**rule**', "**entity**" comme dans la version de Pacman.

Le paquetage "**zelda**" comprend plusieurs paquetages.
Le paquetage "**base**" permet de gérer l'intéraction utilisateur, sons, gestion de mouvements de personnages.
Le paquetage "**rule**" contient les règles de gestion de collisions ou blocages entre entités du jeu.
Le paquetage "**level**" permet de gérer la mise en place des niveaux du jeu, de la lecture de fichier pour les niveaux, à la création de niveaux.
Le paquetage "**game**" contient les classes nécessaires à la création du jeu (gestion de l'univers, sauvegarde de niveaux...).
Le paquetage "**observer**" permet de gérer tout les observateurs du jeu.
Le paquetage "**entity**" possède toutes les classes de toutes les entités présentes dans le jeu : personnages et décors. Dans les personnages, un paquetage est réservé aux états de Link.

2.2 Architecture détaillée

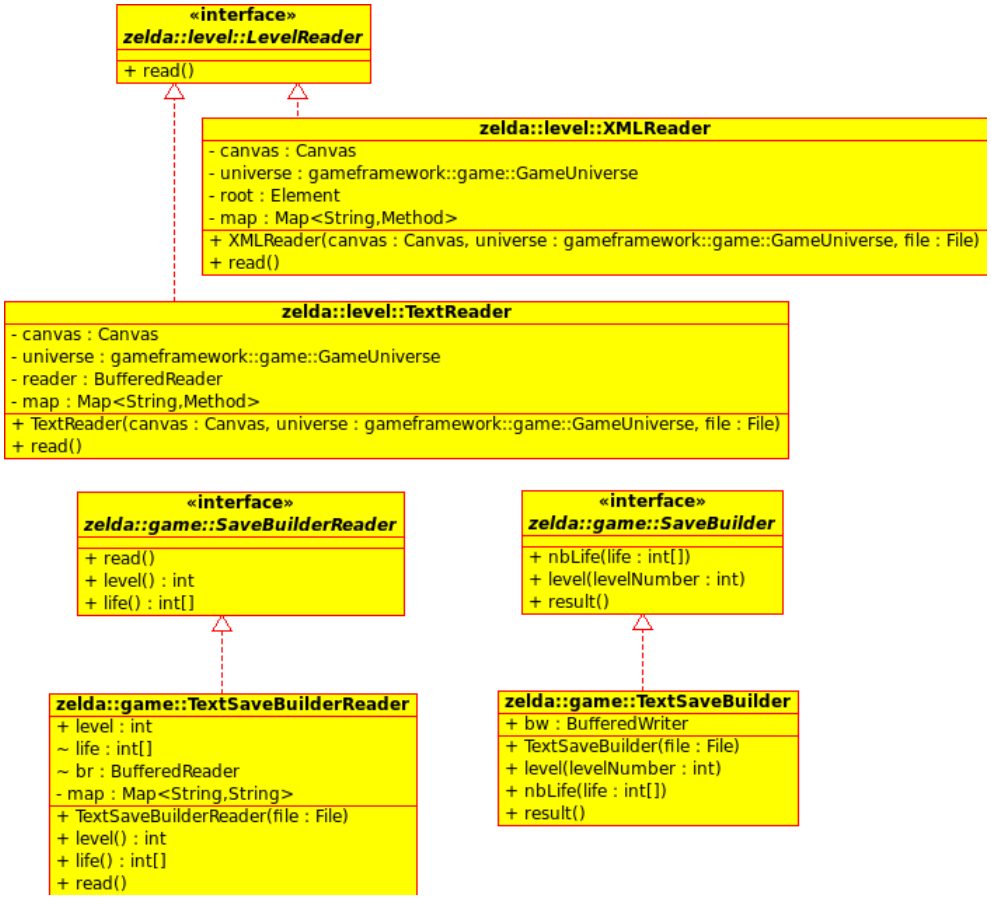
Une Fabrique Abstraite pour créer le niveau depuis un fichier texte

Nous avons choisi d'utilisé une fabrique abstraite pour permettre la création et l'ajout d'entités dans le niveau. Il était assez intressant de créer cette fabrique abstraite car ainsi nous ne faisions juste qu'une demande de création d'un certain objet qu'elle s'occupait d'instancier. Une personne voulant créer un personnage ou une autre entité a juste besoin de faire appel à la méthode create adéquate. Cependant, le problème le plus important et qui est un problème propre à ce modèle est que l'ajout de nouvelles entités est assez fastidieux : chaque nouvel objet il faut une méthode pour le créer. Dans les constructeurs de niveaux actuels il faut ajouter dans la table de hashage la méthode associée.

On reconnaît bien là la structure d'une fabrique abstraite avec ses diverses méthodes de création d'objet.

Le pattern Monteur pour sauvegarder la partie

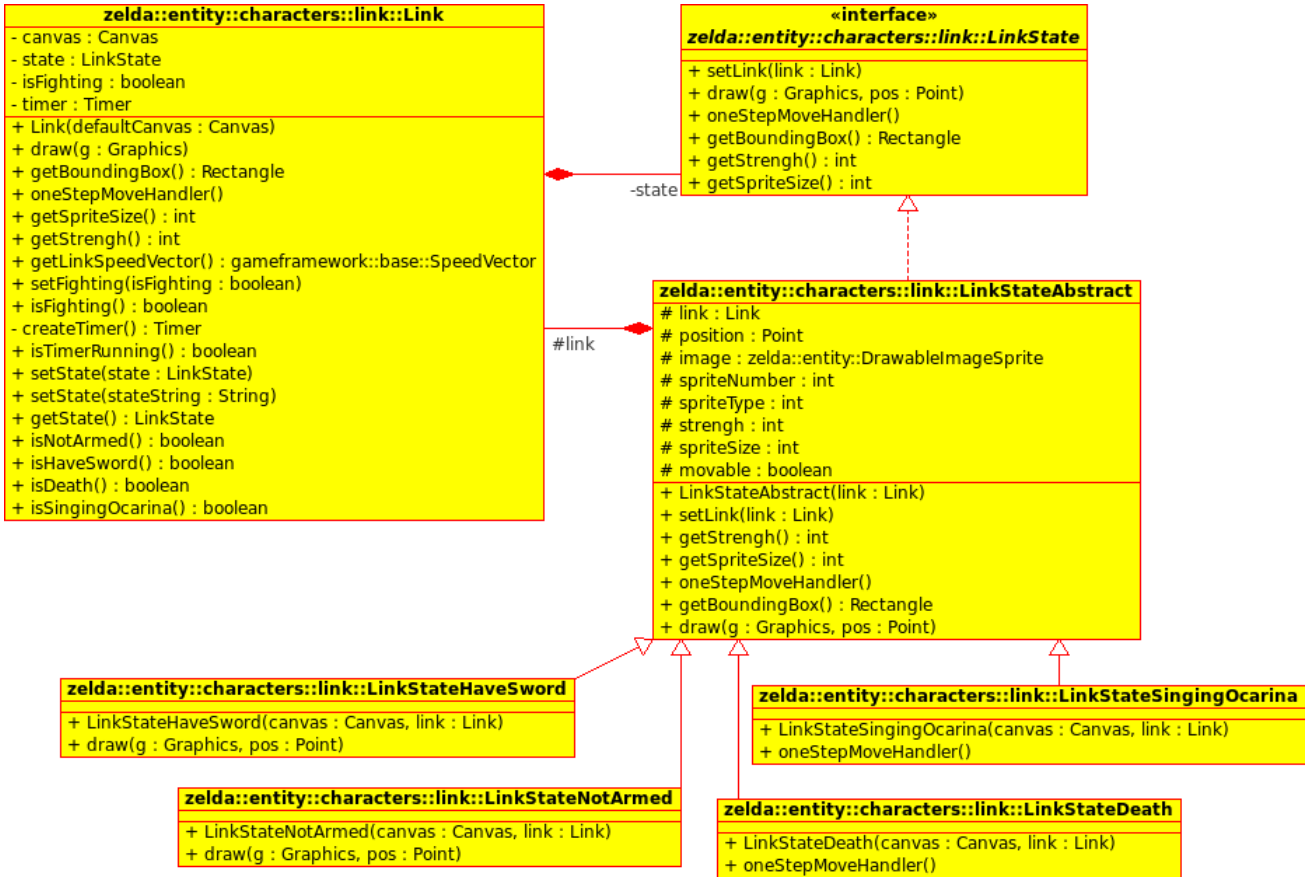
Le fait de sauvegarder la partie est une action classique de tout jeu. Il nous est apparu comme évident d'avoir une représentation sur fichier du contenu de cette sauvegarde. Nous avons donc mis en place un monteur, celui-ci en effet est particulièrement adapté à ce genre de situation. Nous avons donc créer un monteur concret qui permet d'écrire un fichier texte contenant les informations ainsi sauvegarde.
Un autre avantage de ce patron de conception est que nous pouvons ainsi rapidement modifier la représentation de cette sauvegarde. Il est tout à fait possible d'imaginer une sauvegarde au format HTML ou XML, ou on ne sait dans quel autre format. Le format de la sauvegarde est donc facilement modifiable pour n'importe quelle personne reprennant notre code et qui n'aimerait pas notre représentation.



Nous avons aussi utilisé ce modèle pour la lecture de la sauvegarde ainsi que la lecture du fichier de création de niveau et ce pour les même raisons que celles énoncées juste avant. Leur mise en place est visible sur l'image ci-dessus.
Sur cette image on voit bien les divers monteurs, on reconnaît leur méthode de création séquentielle. Il est donc facile de voir comment étendre l'interface de chacun pour pouvoir changer la représentation d'une partie de ce jeu.

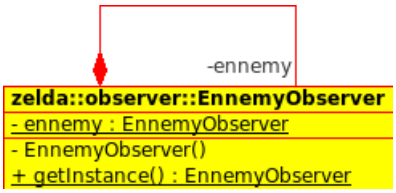
Gestion des états de Link avec le pattern Etat

L'état de Link modifie son comportement. Ces changements apparaissent de faon dynamique, selon les évnements du jeu, ou du joueur. Cette intention nous a amené à utiliser le pattern Etat pour gérer Link. Celui ci peut déléguer son comportement changeant à une hiérarchie de classes LinkState possédant des requêtes spécifiques comme l'utilisation de l'arme, le mode attaque, la mort du personnage...Sur la fin du développement, cette conception nous a permis de rajouter rapidement de nouveaux états, et d'ajuster le comportement voulu facilement.



un Singleton pour gérer le nombre d'ennemis dans le niveau

Dans le jeu, une des priorité est d'éliminer tous les ennemis afin de pouvoir finir le niveau. Afin de gérer le nombre d'ennemis, on crée un observateur unique qui on adresse en début de niveau le nombre d'ennemis à tuer. L'intérêt est ici de créer une unique instance d'observateur et de fournir un point d'accès global à celle ci depuis toutes les classes du jeu. L'utilisation du modèle Singleton fût alors évident.



3 Evolutions

Notre jeu Zelda n'est pas compltement fini, il ne comprend que quelques niveaux, comprenant quelques ennemis, et quelques objets classiques. Quelques sprites ne sont pas parfaits, ni complets. Cependant, le jeu fonctionne correctement, les fonctionnalits que nous voulions au dpart tant implmentes.

Nous aurions voulu, avec plus de temps, produire des niveaux plus riches en entités, plus agréables jouer. Le jeu semble trop court, et nous aurions voulu développer un scénario plus attractif.

3.1 Critiques

Le mouvement des gardes a été honteusement copié de celui des "ghosts" du Jeu Pacman donné, dans un premier temps, ce qui nous a permis de nous concentrer sur d'autres éléments du jeu. Finalement nous nous sommes habitués à ces mouvements primitifs qui ajoutent une difficulté au jeu. Nous pourrions créer une autre méthode de déplacement, plus intelligente, qui suivrait un déplacement précalculé. Une bonne aptitude des gardes serait de changer de méthode déplacement et aller attaquer

Link dès qu'il s'approche trop près de Zelda ou trop près d'une zone donne.

Les sons ne sont pas assez nombreux, ce qui laisse une part de silence trop grande, mais le systme de son est mis en place, ce qui nous relie aux besoins du projet même.

4 Implémentation

Problèmes rencontrés

Nous avons bien sur rencontré plusieurs points difficiles.

Gestion des niveaux

Nous avons mis en place plusieurs niveaux ainsi qu'un changement de ceux-ci, ce qui s'avéra être une tâche ardu. En effet, nous pouvons passer au niveau suivant sans problème mais choisir au hasard un niveau et le lancé est malheureusement impossible. Nous avons voulu mettre en place un systême utilisant ceci via le rechargement de sauvegarde. Avec notre implémentation, quand nous rechargeons un niveau, le niveau se charge mais celui-ci ne démarre pas. Nous avons tenté de diverse faÃ§on de faire en sorte que cela fonctionne sans résultat. En fait, le plus compliqué vient de l'utilisation des threads qui doivent être interrompus avant de passer au niveau suivant et le niveau suivant doit être lancé. Mais pour des raisons inconnus, les threads ne se lancent pas.

XML

En théorie, il est possible de créer des niveaux au format XML grâce au Monteur LevelReader, malheureusement nous avons eu des difficultés à mettre en place ce lecteur de XML. Nous n'avons pas réussi surement à cause du fait que nous n'avions jamais utilisé l'API correspondante auparavant, et les différents tutoriaux présent sur internet n'ont pas suffit à nous permettre de réussir.

Le framework de base

Le fait d'être confronté à un framework dont nous ne pouvions rien toucher, fut un challenge intéressant mais cependant parfois compliqué. Il nous est arrivé à plusieurs reprise de vouloir récupérer une chose d'une classe du framework, alors que cette chose ne nous était pas accessible ou encore modifier une variable privée inaccessible de l'extérieur. Nous avons avoir parfois était tenté par la recopie de code (ce que nous pouvons apercevoir avec la classe GameZeldaAWTImpl) mais il fallait le plus souvent trouvé des astuces. Ces astuces étaient souvent des modèles de conceptions, par exemple nous avons utilisé un décorateur pour permettre d'étendre les fonctionnalités d'une interface qui ne nous permettait pas de récupérer une variable pourtant essentielle sans laquelle le jeu aurait paru étrange aux yeux des divers joueurs.

Gestion des sprites

Une des principales difficultés fût de gérer correctement les sprites. Au départ, nous avions repris le code de Pacman. Celui ci utilisait pour Pacman une grande image contenant une série de sprites différentes. Dans le code apparaissait plusieurs fois le même identificateur SPRITE_SIZE. Il nous a fallu analyser plus en détail le code pour nous rendre compte qu'en fait, tout dépendait de la taille de l'image affichée à l'écran, de sa taille en largeur et hauteur en pixels. Durant plusieurs jours, nous nous sommes entétés à utiliser des sprites complexes, de tailles variées. L'idée fût alors de créer ses propres sprites et de s'occuper de chacun selon ses dimensions respectives.
Ainsi pour Link, on a créé un sprite sans arme, et un autre avec épée. L'utilisation de plusieurs classes permet alors d'utiliser le sprite voulu avec ses dimensions particulières, nottament le nombre de sprite sur une ligne, mais aussi son affichage à l'écran.
La difficulté fût d'abord d'apprendre à utiliser le logiciel de retouches d'images Gimp, ce qui ne fût pas sans peine, sachant qu'il nou fallait peu de temps pour créer une série de plusieurs sprites différents, calés au pixel près.
Ensuite, il a fallut déméler les SPRITE_SIZE selon leur vraie utilité.
Ensuite, il a fallu gérer l'utilisation du bon sprite selon l'état du personnage (pour Link). Link fût le personnage qui fût le plus compliqué à manier dans le sens où justement les transitions d'états

compliquent la gestion du mouvement des sprites selon la classe qui l'utilise. Il n'est pas impossible par exemple que sur la dernière version, les dernières modifications aient entrainées que l'on ne puisse plus voir le sprite de Link en train de jouer un air de musique, car on fait suivre ce mouvement particulier d'une nouvelle transition d'état vers un Link armé par exemple. Surement que ce problème sera débuggué d'ici la remise du projet (en tout cas ça a marché il y a quelques heures !).

Les sprites prirent donc un temps monstrueux dans la gestion du projet, ce qui a affaibli les fonctionnalités où le design des autres entités du décor.