Le dernier labyrinthe avant la fin du monde

Rapport – Projet P2

Droxler Arnaud – Perez Joaquim

16/06/2016

Professeurs : Rizzotti Aïcha, Stähli Joaquim

Abstract

Ce rapport décrit le déroulement du projet P2, de l'élaboration du cahier des charges, selon les contraintes posées par le cours relatif au projet, à la finalisation du programme. Il synthétise le travail effectué, l'architecture finale du programme, les problèmes rencontrés ainsi que les solutions trouvées et les améliorations possibles. Il relève également la manière dont nous avons géré le temps mis à disposition afin de réaliser ce projet.

Table des matières

1. Introduction

L'application « Le dernier labyrinthe avant la fin du monde » est créée dans le cadre du projet P2-Java de la section développement et multimédia de la Haute-Ecole Arc. Le but est de développer une application graphique en Java, le choix du sujet est libre à partir de cette consigne.

Nous avions envie de créer un jeu vidéo qui ressemblait au vieux jeu vidéo en fausse 3D du début des années 90.

L'idée est de créer un « doom-like » en utilisant la méthode du ray casting pour créer notre rendu en fausse 3D nous étions 4 à être motivé par ce projet, nous nous sommes séparé en 2, un groupe va s'occuper du rendu en ray casting, l'autre groupe va créer la logique du jeu (gestion du clavier, déplacement du joueur, des monstres…). Ce rapport présente la conception de la logique de ce projet.

Un des buts de ce projet est de nous familiariser à la collaboration entre étudiants et d'approfondir nos connaissances en java.

2. Méthodologie de travail

Ce chapitre décrit la planification de notre projet et la méthodologie de travail utilisée.

2.1 Méthode de travail

Pour ce projet nous étions deux, nous avons décidé d'adopter la méthode d'agile d'extreme programming. Nous avons commencé par créer une liste d'objectif priorisé, cette liste était mis à jour à chaque début de séance en fonction de notre avancement dans les taches et des choix effectuer. Malheureusement nous n'avons pas maintenu cette liste à jour dans le wiki, nous l'avons de manière orale la plus par du temps.

2.2 Répartition des tâches

Vu que nous avions choisi de travailler en extreme programming, nous étions toujours sur un même pc. Un de nous écrivait le code et l'autre vérifiait que le code écrit soit correct, les rôles s'inversaient d'une séance à l'autre. De ce fait, nous avons une bonne compréhension de tout le code. Nous avons travaillé comme cela sur les ¾ du projet, mais vers la fin, nous nous sommes rendu compte que nous manquions de temps pour que le projet atteigne nos objectifs, nous nous sommes donc séparés et nous nous sommes attribué les taches restantes.

3. Gameplay

Le gameplay de notre jeu est très simple. Le but est d'atteindre la sortie du labyrinthe. Mais avant il faudra trouver la clé pour pouvoir ouvrir la porte qui mène à la fin du niveau. Pour compliquer les choses, nous avons rajouté des ennemis qui vont se balader aléatoirement dans la carte. Ces monstres tuent le joueur instantanément lorsqu'ils le touchent. Pour se défendre, le joueur a une arme qui tue les monstres instantanément aussi.

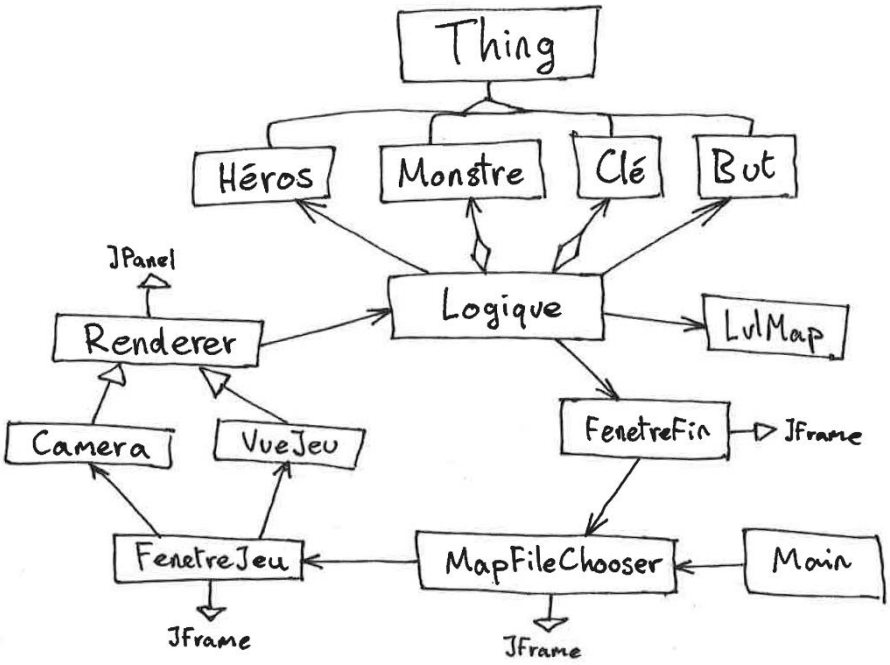
4. Architecture

4.1 Description des classes

Ce chapitre explique en détail la structure de l'application. Les classes principales du programme y sont listées et explicitées.

Les classes outils, comme « Vector2D », « AlgoPiergiovanni », « ImageParser » et « GeometricTools », ne sont pas présentées, car secondaires.

4.2 Diagramme de classes



4.3 Structure du programme

Logique est la classe centrale du programme. C'est cette classe qui s'occupe de déplacer le héros et les monstres. Logique dérive de l'objet Java KeyListener, ce qui lui permet d'écouter les entrées clavier. Elle contient une référence sur la map et gère les collisions entre le héros et les murs de la map.

La classe abstraite Renderer permet d'implémenter son moteur de rendu. Ici, nous avons fait notre moteur de rendu « VueJeu », qui affiche la carte et les Things depuis une vue aérienne en 2 D. Puis nous avons intégré le moteur de rendu « Camera » créé par le groupe de Maxime Piergiovannie et Vincent Chaperon.

4.4 Description des classes

4.4.1 Main

C'est le point d'entrée du programme, elle ne fait qu'instancier un objet MapFileChooser

4.4.2 MapFileChooser

Cette classe hérite de JFrame, elle affiche une boîte de dialogue permettant à l'utilisateur de choisir un niveau à jouer. Si l'utilisateur annule, un niveau par défaut est chargé.

L'objet Swing JFileChooser fournit une boîte de dialogue de choix de fichier.

La classe instancie un objet FenetreJeu en lui passant le chemin absolu du fichier choisi.

4.4.3 FenetreJeu

Cette classe crée une instance de la classe Logique en lui fournissant le nom du fichier map à charger. Elle créer ensuite deux Renderer, un « Camera » et un « VueJeu » en leur passant comme paramètre l'objet logique instancié précédemment.

FenetreJeu est une JFrame, elle contient l'objet Camera. Pour afficher le rendu VueJeu, une autre JFrame est instanciée dans FenertreJeu.

La classe contient aussi un MouseListener, afin que l'utilisateur puisse viser avec la souris dans le rendu 3D de l'objet Camera.

4.4.4 Renderer

Classe abstraite qui contient la référence sur l'objet Logique. Elle déclare une méthode abstraite renderThings(), qui devra être implémentée dans ces implémentations (donc dans Camera et VueJeu dans notre cas).

Sinon, cette classe ne contient qu'une méthode « anmier () » qui démarre un thread qui appelle la méthode repaint () toutes les 50 millisecondes (20 fois par secondes). La méthode repaint () va appeler la méthode paintComponent() des implémentations.

4.4.4.1 Camera

Il s'agit du moteur de rendu créé par le groupe Piergiovanni-Chaperon.

4.4.4.2 VueJeu

C'est notre propre moteur de rendu. La map et les Things y sont directement dessinés à l'aide d'un objet AWT Graphics2D, en redéfinissant la méthode paintComponent() de JPanel. Logique

C'est le « moteur » de l'application. Cette classe implémente elle aussi un timer dans un thread qui, lui, s'occupe de déplacer et gérer les collisions des monstres et du héros. C'est la méthode updateDeplacement() qui bouge le héros en fonction des touches claviers enfoncées. Pour connaitre en tout temps les touches enfoncées, la classe utilise un HashSet. À chaque évènement keyPressed, on ajoute le code de la touche dans le hashSet, on le supprime à l'évènement keyReleased.

4.4.5 Thing

Il s'agit d'une classe abstraite qui définit tout élément à afficher. Les things ont une vitesse, un vecteur position et un vecteur direction. La méthode getSprite() doit être redéfinie dans les implémentations de thing. Cette méthode doit retourner l'image qui sera affichée dans le moteur de rendu.

4.4.5.1 Monstre

C'est un Thing qui est construit avec une vitesse aléatoire et qui renvoie une image effrayante à l'appel de getSprite().

4.4.5.2 Héros

C'est le thing que contrôle le joueur. Il a une vitesse fixée à la création.

4.4.5.3 Key et Goal

Ce sont des Things sans vitesse, donc fixes. Elles ont juste un « sprite » différent.

4.4.6 LvlMap

C'est une classe abstraite qui représente la carte. Elle contient une méthode inWall (double, double) qui retourne vrai ou faux selon si les coordonnées données pointent sur un mur. Cette méthode doit être redéfinie dans les classes qui spécialisent LvlMap.

4.4.6.1 ImageLvlMap

C'est une classe qui spécialise LvlMap. La map est représentée sous forme d'image dans cette implémentation, plus de détails dans le chapitre Protocole.

4.4.7 FenetreFin

JFrame appelée par la classe Logique lorsqu'elle détecte que le héros entre en collision avec soit le but, soit un monstre. Dans le premier cas, elle affiche gagnée, dans le second, perdu. Elle permet à l'utilisateur de recommencer une partie en construisant un nouvel objet MapFileChooser.

5. Algorithmes

Ce chapitre explique comment nous avons résolu quelques problèmes complexes de programmation.

5.1 Collisions

La gestion des collisions était l'un des points chauds de ce projet. En effet, une collision réaliste entre le héros et les murs s'est avérée plus compliquée que prévu.

5.1.1 La méthode collapse ()

Les collisions entre le héros et les différents objets sont gérées par la méthode collapse () de la classe Logique. Cette méthode prend en argument un point et un rayon, elle retourne vraie si la position du héros est comprise dans le carré formé autour du point reçu en argument. Le rayon définit la moitié de la largeur du carré.

5.1.2 Collisions des monstres avec les murs

Dans la méthode updateMonsre(), pour chaque monstre, on sauvegarde sa position dans une variable oldPos. On déplace ensuite le monstre selon un angle aléatoire. On teste si le monstre touche le joueur avec la méthode collapse (). Finalement, on teste avec la méthode inWall de la map si un monstre est dans un mur. Si c'est le cas, le monstre est remis à son ancienne position. Aussi, son vecteur direction est inversé pour qu'il « rebondisse » contre le mur (sans ça, le monstre restait collé au mur).

5.1.3 Collisions du héros avec les murs

Pour la collision du joueur avec les murs, c'est un problème bien plus compliqué. On ne voulait pas simplement remettre le jouer à son ancienne position s'il est dans mur. Le but est de le faire longer les murs.

Le problème est de savoir où est-ce qu'il faut replacer le héros une fois qu'on a détecté qu'il était dans un mur. On doit donc déterminer d'où vient le héros, donc dans quelle direction il est en train de se déplacer.

Pour ce faire, on distingue 4 cas dans la méthode moveAlongWalls() : haut-gauche ; haut-droite ; bas-gauche ; bas-droite. Selon la direction, la méthode testAndMove() aura un autre comportement. L'algorithme est difficile à expliquer, car chacun de ces 4 cas a encore 3 scénarios possibles, il y a donc 12 cas au total.

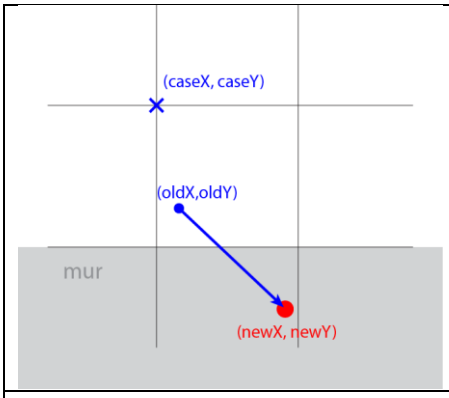
5.1.3.1 Pseudo-code

```
updateDeplacement() :
        on déplace le héros
        s'il est dans un mur :
                moveAlongWalls()

moveAlongWalls() :
        (newX, newY) ←nouvelle position du héros (dans un mur)
        (oldX, oldY) ←dernière position du héros (hors du mur)
        (caseX, caseY) ←dernière position arrondie à l'entier inférieur
        si on va vers la gauche :
                lockX ← caseX
        sinon : (on va vers la droite)
                lockX ←caseX + 0.99
        si on va vers le haut :
                lockY ← caseY
        sinon : (on va vers le bas)
                lockY ← caseY + 0.99
        testAndMove(newX, newY, lockX, lockY)

testAndMove(newX, newY, lockX, lockY) :
        si (newX, lockY) n'est pas dans un mur :
                Déplace le héros en (newX, lockY)//glisse horizontalement
        sinon, si (lockX, newy) n'est pas dans un mur :
                Déplace le héros en (lockX, newy)//glisse verticalement
        Sinon :
                Déplace le héros en (lockX, lockY)//bloque le héros dans le coi
```

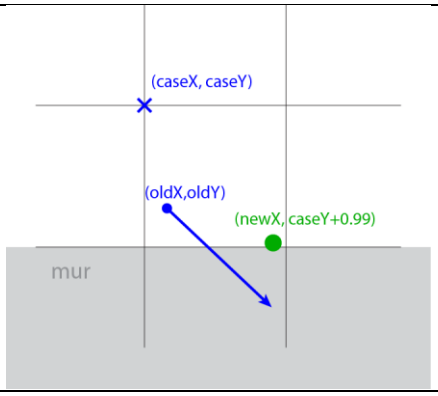
5.1.3.2 Déroulement de l'algorithme

Pour l'exemple, nous allons dérouler l'algorithme pour un des 4 cas, les autres ne variant que très peu. Le cas est « Le héros va en bas à droite ».

Scénario 1 : Le héros entre dans le mur depuis le haut, il peut glisser vers la droite.

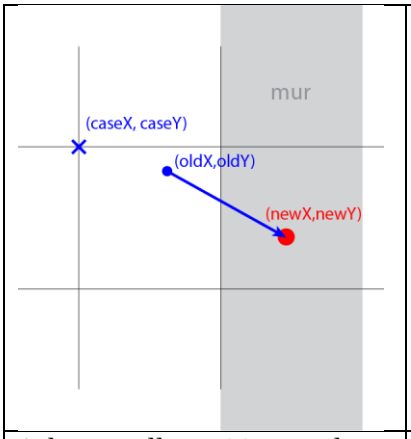
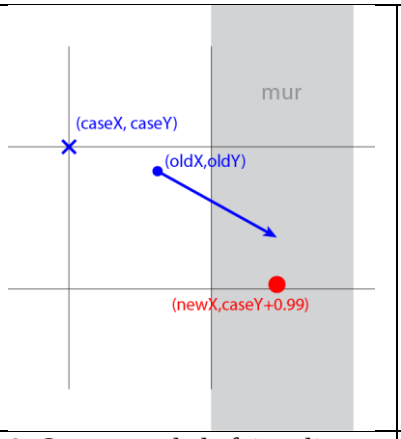
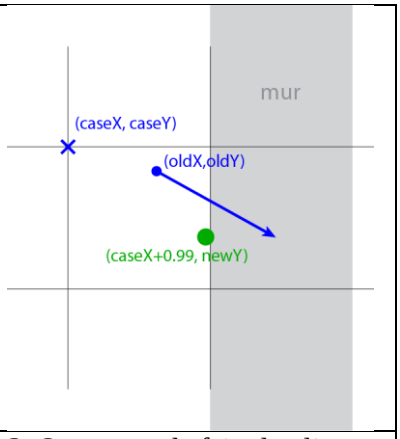


1. La nouvelle position est dans un mur, il faut rediriger le héros dans le droit chemin.

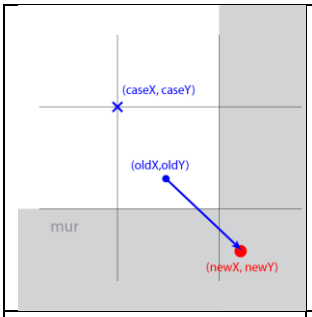
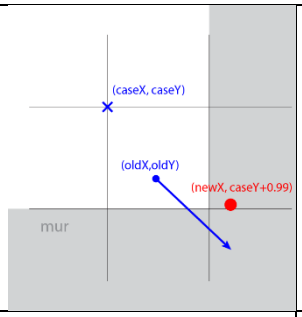
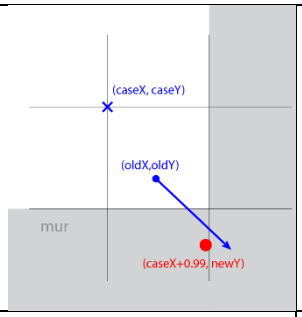
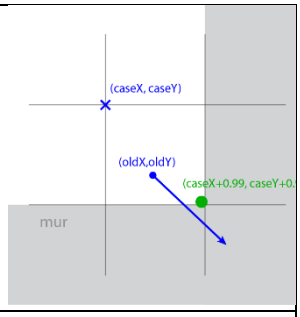


2. On essaye de le faire glisser horizontalement. C'est bon.

Scénario 2 : Le héros entre dans le mur depuis la gauche, il peut glisser vers le bas.

		
1. la nouvelle position est dans un mur.	2. On essaye de le faire glisser horizontalement. On est toujours dans un mur, ça ne va pas.	3. On essaye de faire le glisser verticalement. C'est bon.

Scénario 2 : Le héros entre dans le mur depuis la gauche ou le haut, mais il ne peut glisser ni vers la gauche, ni vers le bas

			
1. la nouvelle position est dans un mur.	2. On essaye de le faire glisser horizontalement.	3. On essaye de faire le glisser verticalement.	4. On est bloqué, on se met dans le coin.

5.2 Tir

Lorsque le héros tir, il faut que le premier monstre en face de lui soit touché, et pas les autres. Pas non plus ceux qui sont derrière un mur. Cela paraît bête, mais ça pose quand même quelques problèmes d'algorithmique.

Déroulement de l'algorithme :

- D'abord, nous obtenons la distance entre le héros et le mur en face de lui grâce à l'algorithme de ray-casting développé par le groupe Piergiovanni-Chaperon.
- Ensuite nous créons une « ligne de tir » entre le héros et le mur.
- Puis, pour chaque monstre :
    - On crée un carré autour de lui qui représente sa « hit-box »
    - On teste si la ligne de tir intersecté la hit-box
    - Si c'est le cas, la ligne est arrêtée sur le monstre, on mémorise ce monstre comme le « monstre à enlever »

- S'il y a un monstre à enlever, on le retire de la liste des monstres

Cet algorithme n'est pas idéal parce qu'il itère sur tous les monstres dans tous les cas. On pourrait imaginer utiliser l'algorithme de ray casting pour détecter le monstre le plus proche en face du héros, comme on le fait pour les murs.

5.3 Concurrence

Dans notre programme, il y a un thread qui manipule des données, et deux autres threads qui utilisent ces données pour en faire un rendu graphique. Le problème principal est la gestion des monstres. Lorsqu'on tir sur un monstre, la logique le supprime de la liste des monstres. Cependant, les moteurs de rendu itèrent sur cette même liste pour dessiner les monstres. Il y a donc un problème de concurrence de type lecteur-rédacteur.

Pour l'instant, ce problème n'est pas correctement géré, on se contente de faire des try-catch pour passer outre les exceptions. Cependant, nous sommes conscients du problème et il fait partie de la « to do list

6. Interaction avec l'utilisateur

6.1 Déplacement du joueur

Pour déplacer le personnage, nous avons fait hériter la classe logique de la classe KeyAdpter. Cela nous permet de redéfinir la méthode keyPressed(). En fonction de l'évènement, on peut faire avancer, reculer, « straffer » à droite ou à gauche (se déplacer en pas chassés). Comme dans la plupart des jeux, nous avons utilisé les touches w, a, s, d pour faire bouger le personnage.

6.2 Déplacement du curseur de tir

Pour déplacer le curseur, on a deux méthodes possibles

6.2.1 Clavier

Pour le curseur nous utilisons les flèches directionnelles de droite et de gauche pour diriger le joueur. On va utiliser la méthode que pour le déplacement on appellera la méthode de rotation à droite ou à gauche.

6.2.2 Souris

Pour la souris, on va regarder la position du curseur dans la fenêtre. Si le curseur sort de la fenêtre sur l'un des côtés gauche ou droite, on le replace dans la fenêtre du côté opposé de l'endroit où il est sorti. C'est le deltaX entre un évènement souris et l'autre qui détermine de combien de degré il faut ajuster la direction du joueur. Donc plus on bouge la souris rapidement, plus le héros bouge la tête rapidement.

6.3 Tir

Pour déclencher le tir, on a deux méthodes possibles

6.3.1 Clavier

Avec le clavier, la barre espace permet de déclencher le tir.

6.3.2 Souris

Avec la souris, on utilise le clic gauche de la souris la méthode moussePressed() sera utiliser pour déclencher le tir, cette méthode est implémentée dans la classe FenetreJeu.

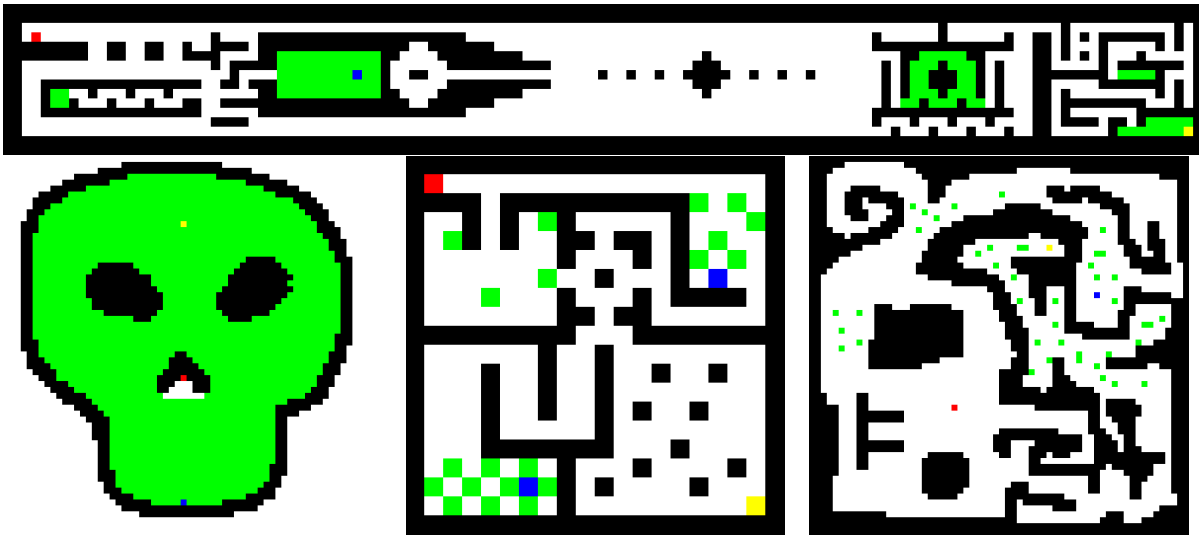
7. Protocole

7.1 La map

Dans la phase de conception, nous avons longuement discuté sur la du fichier qui représenterait la map. Cet objet doit contenir une matrice qui représente les murs, ainsi que les emplacements des objets comme la clé, la porte, le héros et les monstres.

Nous avons survolé des solutions comme TiledMap, mais nous avions envie de créer notre propre solution de gestion de map, donc nous n'avons pas retenu cette idée. Ensuite nous avons pensé à établir un protocole et écrire un parseur pour sauvegarder les maps en XML. Cette solution était satisfaisante, mais il était laborieux d'éditer les cartes, il fallait renseigner chaque valeur de la matrice par un 1 ou un 0 pour indiquer s'il y a un mur ou pas.

Finalement, nous avons opté pour une représentation sous forme d'image, éditable facilement dans n'importe quel éditeur d'image. Voici quelques exemples de fichier map sous forme d'image :



Pour qu'une carte soit valide, elle doit respecter les conditions suivantes :

- Contenir un pixel rouge (#FF0000) qui sera le point de départ du héros
- Contenir un pixel jaune (# FFFF00) qui représente le but
- contenir au moins un pixel bleu (# 0000FF) qui représente une clé
- Être fermée par des pixels noirs (#000000) qui seront les murs.

Les pixels verts représentent les monstres, on peut en mettre à volonté.

7.2 La classe ImageParser

Pour créer une map à partir d'une image, la classe Logique utilise la classe outil "ImageParser". Cette classe contient une méthode statique getMap() qui reçoit en paramètre un nom de fichier image et qui retourne un objet LvlMap.

Pour parser la map, elle itère sur chaque pixel et elle affecte les attributs de l'objet LvlMap en fonction des couleurs qu'elle rencontre.

7.3 La méthode inWall() de la classe ImageLvlMap

Cette méthode va simplement consulter le pixel x, y selon le paramètre fournit et retourne vrai si le pixel est noir.

8. Problèmes rencontrés

8.1 Eclipse et Git

Un des problèmes qui nous a le plus handicapé est la gestion du répertoir git et du workspace eclipse. N'ayant jamais configuré un worksapce avec un répertoire git, nous ne savions pas comment faire et nous n'avons pas fait les manipulations dans le bon sens. Le problème majeur étant que les fichiers de config eclipse était versionnés, mais après avoir ajouté un gitignior, le workspace ne s'ouvrait plus. Nous avons donc tout recréé dans le bon ordre, heureusement nous n'avons pas perdu de code dans cette opération.

8.2 Format de la Map

Nous avons longtemps hésité sur le format du fichier map et fais de nombreux test. Cela nous a fait perdre beaucoup de temps, il aurait été préférable de plus réfléchir lors de la conception et de s'en tenir à un seul choix. Au lieu de cela nous avons créé plusieurs parseurs pour différents format (XML, XPM, PNG) pour au final choisir le format PNG.

9. Améliorations potentielles

Le moteur du jeu est fonctionnel, cependant, de nombreuses améliorations sont possibles. Faute de temps, nous avons dû renoncer à l'implémentation de plusieurs fonctionnalités que nous nous réjouissions d'ajouter au projet.

9.1 Éléments de gameplay

Par exemple, quelques mécaniques de jeu :

- Ajout de point de vie et de point d'armure et d'item dans la carte pour rendre les points de vie et d'armure.
- Rajouter des ennemis différents, des boss de fin.
- Créer une véritable IA pour les monstres.
- Des armes différentes avec des propriétés différentes.

9.2 Multijoueur

Une des améliorations possibles est de créer un mode multijoueur. Les joueurs se retrouveraient dans une partie et pourraient se battre dans une arène à la manière d'un Couter Strike. Nous pensons d'ailleurs réaliser cela durant la HES d'été.

9.3 Éditeur

Une bonne idée serait de créer un éditeur de thing, cela permettrait de créer ces propres asset à ajouter dans le jeu, l'éditeur permettrait au joueur de créer un thing lui assigne une image et une couleur à parser dans la map. En parallèle on pourrait créer un éditeur de map, à la place d'utiliser un logiciel comme Photoshop ou Gimp. Cela éviterait de créer des maps avec un mauvais code couleur.

10. Conclusion

Les objectifs sont remplis. Nous avons bien géré le travail de groupe en gardant une bonne communication et un partage des tâches efficaces depuis le début du projet. Les connaissances de chacun étaient complémentaires et l'entraide automatique. Ce projet nous a permis d'approfondir au maximum nos connaissances en Java et d'une manière générale, la programmation graphique.

Nous sommes satisfaits du résultat final. Le moteur du jeu est fonctionnel et le programme est modulable. Il est tout à fait envisageable d'ajouter des fonctionnalités telles que de la vie ou des armes. Nous voyons donc des perspectives d'avenir pour ce projet et nous réjouissons d'en continuer le développement.

Neuchâtel, le 16.06.16

Droxler Arnaud Perez Joaquim

11. Bibliographie

La principale source d'informations que nous avons utilisée est la documentation officielle de Java, accessible sur le site d'Oracle : https://docs.oracle.com/javase/7/docs/api/