PROJET : UN JEU LABYRINTHE

22 décembre 2017

Université de Bordeaux

ORTIZ LOZANO Jeniffer
DEMOULINS Louis
MASSAMIRI Michel
PERUZZETTO Enzo

Introduction

Le présent rapport traîte du projet de programmation orienté objet qui nous a été demandé de faire pour notre premier semestre de master informatique de l'université de Bordeaux. Pour ce projet, il nous a été demandé de programmer en java un jeu de labyrinthe, librement inspiré de PAC-Man. Pour ce faire nous avons eu à notre disposition les bibliothèques de java et une bibliothèque annexe permettant de travailler avec des graphes. Nous avons utilisé les bibliothèques java vu en cours et en TD, à savoir JavaFX.

Le but de ce projet était de nous faire travailler les concepts de programmation orientée objet que nous avons pu étudier durant le semestre, comme l'héritage, le polymorphisme ou encore les design patterns.

Nous traiterons dans ce rapport de plusieurs points. Nous mettront tout d'abord en valeur les différents Design patterns que nous avons utilisé tout en explicitant leurs utilisations repesctive au sein du projet. Dans un second temps nous parlerons des classes qui en sont pas necessairement triviales au projet, ou dont le fonctionnement necessite clarification. Enfin, la dernière partie sera consacrée à l'arbre de nos classes et un lien vers la documentation de notre projet.

Table des matières

1 Répartition du travail

Dans un premier temps, nous avons commencé par réfléchir à la structure globale du projet à quatre. Puis, en ayant commencé tous ensemble le constructeur, nous nous sommes ensuite répartit les taches du projet. Jeniffer et Enzo se sont d'abord occupé de la partie affichage du jeu (qui, nous le verrons plus tard, est représenté par la vue/view). Michel et Louis ont pris en main la partie données de l'application(qui sera représenté par le modèle/model.

Dans l'ensemble du projet, malgré quelques moments d'entre-aide ou nous sommes un peu sortit de nos tâches respectives, nous avons respecté ce schéma de répartition du travail (c'est à dire Michel et Louis s'occupant principalement du modèle et Enzo et Jeniffer s'occupant de la partie Controlleur et vue).

Pour plus de précision, Louis s'est chargé de la gestion du personnage dans le modèle, de la mise en place de ce qu'on appèlera les entitées (éléments du jeu, nous en parlerons plus loin), et de ce qui est dans l'ensemble relatif à ces entitées.

Michel s'est occupé de la partie algorithmique du jeu(algorithme Manhattan), de la modélisation du graphe pour que ça soit cohérent(classe Labyrinth, Edge et Vertex).

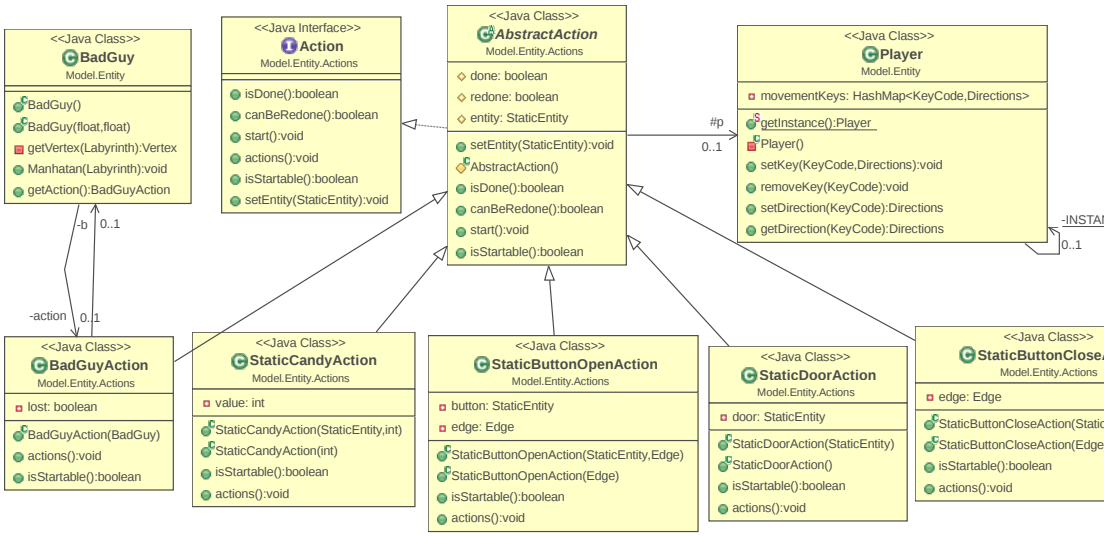
Après avoir fini donc les tâches qui ont été repartis entre nous, nous avons développé tous les 4 la classe GameController.

2 Architecture du projet

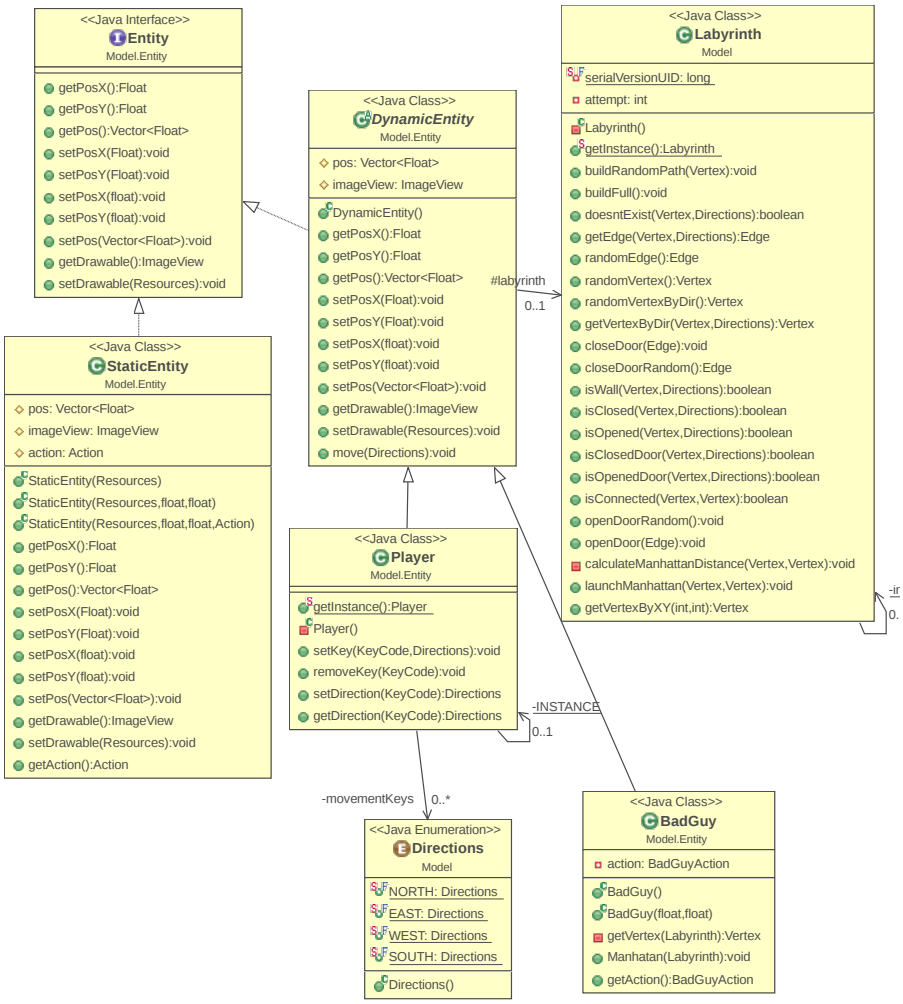
2.1 Diagrames UML

Modèle vue contrôleur

Vue général du modèle



Inferface Action



Interface Entity

2.2 Design patterns utilisés

Pour ce projet nous avons principalement utilisé deux designs patterns. A savoir Le singleton et le MVC (model-view-controller).

2.2.1 MVC

Le MVC a été utilisé pour découper en trois notre projet. Les deux parties les plus importantes (en terme de code) ont étés le controlleur et le modèle. Le modèle contient toutes les données et leurs traitement respectifs. C'est dans cette partie que nous avis mis, par exemple, les entitées, la gestion des ressources exterieurs, etc. Le controleur gère le traitement des entrées utilisateur, le temps et l'intelligence artificiel. La vue, quant à elle, gère uniquement l'affichage du jeu.

Ci-dessous le détali de cette mise en ouvre :

Game Controller : Cette classe collectione atous les contrôleurs (Player et Bad Guys Controller) de l'interface graphique. Chaque fois que l'utilisateur active le contrôleur, il appelle le controleur ou la méthode dans le modèle pour lui dire de changer son état. Les écouteurs sont de type Event, qui détectent une action de l'utilisateur pendant la parti et appellent la méthode du modèle qui prend en charge cette action.

ViewFrame : Cette classe coordonne l'apparence de l'interface du jeu. Elle décide où vont les contrôles et les affichages (de stages, arêtes, candys, player etc.) Lorsque le modèle a changé d'état, la vue se réaffiche en appelant des méthodes dans le modèle qui renvoient toutes les informations que le la viewframe doit afficher. En effet, elle contient les détails sur l'apparence du jeu.

Model : Cette classe est le pont entre notre GameController et la ViewFrame. Le joueur actif le contrôle, qui appelle la méthode dans le modèle pour lui déplacer dans le labyrinthe ; après cela, la classe ViewFrame appelle d'autres méthodes dans le modèle qui lui fournissent les informations qu'il affiche).

Comme nous avons pu le voir dans la partie précédente, ce design pattern nous a été très utile lors du partage des taches.

2.2.2 Singleton

Le singleton permet de creer des instances uniques de classes et assure qu'il ne peut pas y avoir plusieurs instances d'une même classe.

Nous avons beaucoup utilisé le singleton lors de ce projet. En effet les trois parties du MVC, mais aussi le joueur, le ResourcesManager, et la classes Labyrinth sont des singletons.

2.3 Explications de classes

Dans cette partie nous tâcherons d'expliquer le fonctionnement et/ou l'utilité de certaines classe dans ce projet.

2.3.1 ResourceManager

Le resource manager a une utilité qui permet d'améliorer considérablement les performences du programme. En effet, cette classe, qui est un singleton, permet de ne charger qu'une seule fois chaque ressource. La précision "ressource" est importante, car si dans ce cas précis, il s'agit d'images, si on souhaite ajouter des musiques, des shaders, des bruitages, des vidéos, ou tout types de fichiers externes au programme qui necessitent d'être chargés en mémoire, on pourrait le faire via cette classe en la généralisant.

Cette classe permet donc de charger des images via une méthode load, et de les décharger via une méthode unload. Une fois chargées, les images sont stockées dans une Map, qui est indexée par des éléments d'une énumeration nommée Resources. On peut ensuite accéder à ces images via la fonction get, qui retourne une copie du pointeur sur cette image. Ainsi elle n'est pas dupliquée en mémoire. On peut donc afficher plusieurs fois la même image sans avoir a la charger plusieur fois, ou l'avoir dupliqué en mémoire.

2.3.2 Entity

Entity est une interface qui permet de définir le fonctionnement global des entitées. A savoir gerer leurs positions ou leurs images. Deux classes implémentent cette interface, à savoir AbstractDynamicEntity et StaticEntity. Ces deux classes représentent respectivement les entitées pouvant se déplacer et celle qui sont fixe sur la carte.

Le point que nous aimerions appuyer ici est le système de déplacement que nous avons mis en place pour le player. Nous nous somme posé la question de la manière dont nous pourrions faire en sorte qu'avec un code générique nous puissions choisir les touches du clavier qui permettent de déplacer le joueur sans avoir à modifier les conditions dans le controlleur. Ainsi, la solution que nous avons trouvé est de créer une Map qui prend comme entré des $JavaFX.scene.input.KeyCode$ et qui retourne une $Direction$ (qui est une enumeration des quatres directions). Ainsi, il ne reste plus qu'a lier n'importe quelle touche avec une direction, et, lors de la boucle de jeu, quand on récupère un évennement de type keyPressed, on peut juste vérifier si l'entrée $Map[event.keycode]$ existe. Si tel est le cas, alors on peut déplacer le joueur dans une direction. Sinon, on ne déplace pas le joueur.

2.3.3 Action

La classe action permet de définir les comportements des différentes entitées du jeu. On définit les conditions qui font que le comportement doit être fait ou non, on définit si ce comportement doit se répéter, puis on code le comportement en tant que tel, le tout dans une classe fille de la classe AbstractAction. Un fois ceci fait, on attache notre action à notre entité, et une fois que toutes les conditions de son lancement sont réunies, elle se déclanche.

Par exemple, dans le cas de la classe BadGuy, la condition de déclancehement de l'action est que le joueur et le BadGuy doivent être au même endroit (i.e. avoir les même coordonnées). Une fois que cette

conditon est remplie, on lance l'action qui est d'informer le jeu que le joueur vient de se faire "manger", et que par conséquant la partie est terminé, le joueur ayant perdu.

2.3.4 Vertex

On a dû redéfinir les sommets du graphe pour les rendre cohérents vis à vis de notre jeu. Donc, chaque sommet a une position dans le Labyrinthe avec un ID privé qui est essentiel pour calculer les distances de l'algorithme de Manhattan.

Cette classe a ainsi des méthodes permettant de vérifier si un sommet est à l'interieur des bordures du jeu.

2.3.5 Edge

Donc, après avoir définit notre propre classe Vertex pour les sommets du graphe, il est temps de d'implémenter notre arête du graphe et de déterminer ses caractéristiques pour qu'elle corresponde à notre jeu.

La classe Edge hérite de la classe DefaultEdge afin d'utiliser les propriétés principales d'une arête et aussi cela nous permet de changer le comportement de certaines méthodes en utilisant le Polymorphisme. Notre arête est composé d'un Type(Opened door, Closed door ou bien Corridor) de plus. Puisque, nous voulons comparer les arêtes du graphe, La classe Edge a une méthode compareTo qu'on la redéfinit en implémentant la classe Comparable<Edge>.

2.3.6 Labyrinth

Cette classe est la classe qui modélise le graphe souhaité pour le jeu. Afin de définir une telle classe, on a dû réfléchir au différentes classes qui sont déjà écrites pour manipuler des graphes. Suite à ça, nous avons choisi d'éténdre la classe SimpleGraphe en utilisant le concept de la généricité pour indiquer au Labyrinth que on implémente un graphe ayant la classe Vertex comme sommet et la classe Edge comme arête.

Cette classe donc, s'en sert également de la technique Singleton afin d'allouer qu'une seul instance de Labyrinth(de graphe) dans la mémoire.

Le Labyrinth est alors construit aléatoirement en appelant la méthode 'buildRandomPath', ainsi que c'est dans cette classe qu'on trouve l'implémentation de l'algorithme Manhattan. Bien évidamment, cette classe définit plusieurs méthodes afin de réagir le mieux avec n'importe de changement possible dans le jeu. Donc elle fournit au Controlleur les méthodes qu'il a besoin, pour pouvoir interagir avec les actions.

3 Développement du projet

Nous avons abordé ce projet d'une manière que nous pouvions développer en tant que groupe et sur les spécifications demandées. Pour cela, nous utilisons les points suivants :

3.1 Quelles caractéristiques inclura le jeu

Dans ce point, nous étions entrés dans la planification. Notre objectif dans cette étape était de comprendre ce qui nous a été demandé, nous avons aussi réfléchi par rapport aux fonctionnalités du jeu, lesquelles inclure et lesquelles rejeter. Nous avons fait des maquettes sur papier des classes du projet et de l'algorithme pour construire le labyrinth pendant le premier TD.

3.2 Comment nous avons fait l'implementation du jeu ?

Dans ce point nous avons discuté de comment nous allions l'implémenter ? Quelles classes devrions-nous écrire en premier ? Avec quels designs patterns nous allions travailler ?

3.3 Quel était notre temps de travail dans le projet et comment nous faisions la répartition de tâches ?

Nous avions réfléchi au temps nécessaire pour faire ce projet de manière hebdomadaire. Ainsi, chaque semaines, nous pensions aux caractéristiques que nous allions écrire. Notre objectif dans cette étape était de trouver un calendrier pour réussir à rendre le programme dans les temps. De plus, ce point était vraiment important pour nous aider à apréhender le temps qu'il nous fallait pour écrire un programme de ce type.

3.4 Comment modéliser le mieux possible notre architecture(structure) des classes afin d'avoir le maximum possible de cohérence ?

Afin d'avoir une vision globale de l'architecture de nos classes, on a dû faire plein de schémas pour présenter le mieux les solutions possibles(structure des classes, quel design pattern utiliser, etc..).

Éventuellement, nous avnos alors envisagé le MVC et une structure des classes qui permet d'avoir le maximum possible de cohérence en utilisant les notions 'Interface' et 'Abstract' dans Action. On avait procédé de cette façon pour la couche Model(séparer la couche Model en plusieurs sous couches).