Projet Approche Objet

UN JEU DE PLATEAU LABYRINTHE

Guillaume CHARLET
Kenji FONTAINE
Romain ORDONEZ
Adrien HALNAUT

22 décembre 2017

Table des matières

1 Description du projet **3**

2 Architecture du code **3**

2.1 Main . . . 3

2.1.1 controllers . . . 3

2.1.2 models . . . 3

2.1.3 views . . . 3

3 Description du travail **4**

3.1 Première séance . . . 4

3.2 Premier pas . . . 4

3.2.1 Modèle graphe . . . 4

3.3 Implémentation des entités . . . 5

3.3.1 Déplacement du joueur . . . 5

3.3.2 Objets statiques . . . 5

3.3.3 Gestion des collisions . . . 5

3.3.4 Ennemis . . . 5

3.3.5 Boutons OnOff . . . 5

3.3.6 EntitySpawner . . . 6

4 Bilan **7**

5 Perspective d'amélioration **7**

1 Description du projet

Ce travail a été réalisé dans un cadre universitaire, par des étudiants en Master 1 informatique à l'université de Bordeaux. Le but de ce projet est de mettre en pratique les connaissances acquises lors du cours d'Approche Objet.
Labyrinth est un jeu de plateau dans lequel le joueur peut se déplacer et devra trouver la sortie dans un labyrinthe en franchissant divers obstacles.

2 Architecture du code

Le dossier src contient notre code source pour le projet, réparti dans différents sous-dossiers. On y trouve également Main, qui est éxécuté lorsque le programme est lancé.

2.1 Main

Description de Main

2.1.1 controllers

Master : Description de Master.

ingame :
— EntitySpawner : Instancie les entités du dossier drawable avec une position aléatoire.
— EventManager : S'occupe de détecter et de gérer les collisions entre entités (entre le joueur et un ennemi par exemple).
— GameManager : Gère le déroulement du jeu. Initialisation, gestion des déplacements des entités, ainsi que leur destruction (si un bonus est ramassé par exemple).
— Inputs : Reçoit les entrées utilisateurs afin de déplacer le joueur dans le labyrinthe.
— SpriteManager : Permet de récupérer les images utilisées dans l'interface graphique.

2.1.2 models

drawable :
— Bonus : Classe des bonbons ramassables par le joueur.
— Character : Classe utilisée par le joueur ainsi que les ennemis. On distingue le joueur des ennemis à l'aide d'un booléen.
— Door : Classe de la porte de sortie utilisée dans le labyrinthe.
— Entity : Classe abstraite héritée par toutes les autres entités. Elle définie le fonctionnement générale d'une entité.
— EntityType : Enumération des différentes entités pouvant être utilisées dans le jeu.
— OnOff : Classe des boutons permettant d'activer/désactiver un mur dans lebyrinthe.
— SpriteType : Enumérations des différentes images pouvant être associées à une entité dans le jeu.

game :
— maze/Direction : Enumération des directions différentes qu'un mouvement peut prendre.
— maze/Menu : Classe du Menu, non utilisée.
— maze/WallType : Enumération des différents états qu'un mur peut prendre.

2.1.3 views

ViewFrame : Permet d'initialiser l'interface graphique ainsi que de mettre l'affichage des entités à jour à chaque déplacement.

3 Description du travail

3.1 Première séance

Lors de la première séance, nous avons réfléchi sur la structure qu'allait prendre notre code. Nous nous sommes mis d'accord pour partir sur une architecture MVC (modèle, vue et contrôleur) afin de disposer d'une certaine flexibilité. Nous avons ensuite déterminer sur papier, la hiérarchie de nos classes, packages, etc... Puis nous nous sommes réparti le travail en groupe.
Voici le premier modèle retenu :

- Model
  - Drawable
    - Gentil
    - Mechant
  - Static
    - Switches
    - Doors
    - Bonus
  - Game core
    - Laby/Graph
    - Menu
- View
  - Rien pour l'instant
- Controller
  - Main/Master
  - In-Game
    - Inputs
    - Events
    - GameManager

3.2 Premier pas

Nous avons commencé notre implémentation par la génération de graphe (modèle) ainsi que l'affichage de la grille (vue). Le but était d'obtenir une interface graphique avec un affichage d'un labyrinthe. Pour cela, il nous a également fallu implémenter Master (contrôleur) afin de faire le lien entre le graphe généré en modèle et l'affichage. Cette partie nous a pas posé de problème particulier. Guillaume s'est occupé de la partie contrôleur, Adrien de la génération de graphe et l'affichage a été fait par Romain et Kenji.

3.2.1 Modèle graphe

Nous avons préféré implémenter notre propre module de graphes, celui-ci se rapproche de celui proposé dans le sujet, mais en un peu plus simpliste et plus destiné à une utilisation pour un jeu de plateau. Ce module est abstrait du reste du code par la classe Maze, qui s'occupe des manipulations moins atomiques sur le graphe. Ainsi, la classe Graphe et ses dépendances gèrent la manipulation de graphe, c'est-à-dire en utilisant des points (Vertex) et des liaisons (Edge) en tant que tel, et la classe Maze, utilisée par notre jeu, gère les transformations et la récupération d'informations de Graphe. Adrien s'est occupé de son implémentation principale, et Romain ainsi que Kenji de modeler son utilisation pour remplir les objectifs du jeu (respectivement calcul de parcours/distance, gestion des portes et interrupteurs).

3.3 Implémentation des entités

3.3.1 Déplacement du joueur

Une fois la génération et l'affichage du labyrinthe effectués, notre prochain objectif était d'implémenter le joueur (affichage + déplacement). Pour se faire, nous avons commencé par implémenter une classe abstraite Entity, afin de ne pas avoir de redondance dans notre code. Puis nous avons crée une classe Character héritant de Entity, en implémentant les quelques fonctionnalités propres à cette classe (comme le déplacement). Cette partie a principalement été faite par Adrien.

3.3.2 Objets statiques

L'implémentation des objets statiques que l'on peut afficher tel que les bonbons a été faite de façon naturelle. Il nous suffisait d'hériter de la classe Entity et d'implémenter les quelques fonctionnalités propres à chaque objet. Sauf pour les boutons OnOff permettant d'activer/désactiver une porte, étant lié à un certain mur, il était plus compliqué d'implémenter ces boutons. Tous les membres ont contribués à l'implémentation des différents objets.

3.3.3 Gestion des collisions

Le prochain objectif était de faire disparaitre les bonbons au contact du joueur. Pour cela nous avons créer un contrôleur EventManager dont le but est uniquement de gérer les collisions. EventManager parcourt la liste de toutes les entités en jeu et envoie un signal aux entités si il y a collision. Cette partie a principalement été traitée par Guillaume.

3.3.4 Ennemis

Les ennemis sont représentés par la même classe que le joueur (Character), la seule différence étant un booléen et leur déplacement automatisé. Deux façons nous ont été proposées pour la gestion des ennemis, et notamment pour leur déplacement. La première solution se basait sur un système de tour par tour ; le joueur se déplace, puis les ennemis. La deuxième solution était que les ennemis se déplacent en temps réel, même si le joueur ne bougeait pas. Nous avons opté pour la première solution, car nous la trouvions plus proche de ce qui était attendu. Nous avons donc implémenté l'algorithme de Manhattan afin que les ennemis puissent se rapprocher du joueur. Cet algorithme permet de déterminer la distance de chaque sommet à un sommet donné, ainsi on l'utilise principalement pour déterminer la direction dans laquelle les ennemis devraient se déplacer afin qu'ils se rapprochent du joueur. Le mouvement est effectué par le GameManager. Lorsque un ennemi touche un joueur, un signal est émis à l'eventHandler, qui mettra fin au jeu. Cette partie a principalement été faite par Romain.

3.3.5 Boutons OnOff

Après avoir implémenté les bases du jeu (joueur, ennemis, déplacements, collisions), nous devions implémenté les portes qui s'ouvraient ou fermaient à l'aide d'un bouton. Le bouton OnOff est également une entité qui hérite de la classe abstraite Entity, cependant ses fonctionnalités sont plus complexes que les autres entités telles que les bonbons. Pour se faire, nous avons ajouté un membre à la classe OnOff, qui est une arête du graphe en modèle. Ceci représente le mur qui est actionné par le bouton dans le labyrinthe. Lors d'une collision entre le joueur et le bouton, un signal est envoyé à l'EventManager, qui va envoyer un signal au modèle du graphe et modifier le type de l'arête en question (WallType). Ceci aura pour effet d'ouvrir ou de fermer une porte. La vue va ensuite mettre à jour l'affichage est modifier la couleur du mur (vert/rouge). Cette partie a principalement été faite par Kenji.

3.3.6 EntitySpawner

Afin de facilier l'ajout/suppression d'entités dans notre labyrinthe, nous avons créer une classe EntitySpawner dont le but est d'instancier une entité et de la positionner sur le labyrinthe. Cela nous permet d'alléger le code dans la classe Master et d'être plus efficace sur la gestion de nos entités.

4 Bilan

Dans l'ensemble, nous trouvons que le projet s'est bien déroulé. Nous estimons avoir répondu à la plupart des fonctionnalités attendues, même si certaines paraissent "injustes" à cause de la génération complétement aléatoire des positions. La répartition des tâches n'a pas été simple, beaucoup de parties du projet semblaient étroitement liées et nous avions du prendre le temps de mettre sur papier un modèle MVC bien séparé afin de pouvoir travailler efficacement en parrallèle.

5 Perspective d'amélioration

Une des améliorations que nous envisagions mais n'avons pas eu le temps d'implémenter était une amélioration du contrôle de l'aléatoire. Dans la version actuelle, toutes les entités sont générées à une position aléatoire dans le labyrinthe. Cela signifie que le joueur peut très bien apparaître juste à côté d'un ennemi. Une solution envisagée serait d'utiliser les valeurs obtenues par l'algorithme de Manhattan afin de faire apparaître les ennemis à une distance minimale du joueur.
Nous pensons également que certaines parties de notre code ne sont pas pertinentes. Nous pouvons touver des commentaires "TODO : à déplacer" dans certaines classes (ViewFrame par exemple), mais n'avons pas eu le temps de nous concerter pour améliorer notre code.