The Big Adventure rapport

ALVES Rayan COSERARIU Alain

Introduction

L’objectif de ce projet est de produire un jeu 2D vue du dessus entièrement coder en java utilisant la librairie graphique zen5 utilisant les images du jeu d’énigme Baba Is You.
Pour ce faire, nous devions répondre le plus fidèlement au cahier des charges fourni tout en gardant le code le plus maintenable possible en respectant les principe de la programmation orientée objet.
Ce rapport détaille les méthodes mises en œuvre dans une documentation technique pour implémenter les fonctionnalités relatives au jeu ainsi que les remarques prises en compte lors le la soutenance bêta.

Documentation technique

Cette section détaille et justifie l'implémentation de certaines parties du code.

Boucle principale de jeu

Comme à tout instant dans le jeu il y a des événements, les ennemis ou le joueur qui bouge en sont des exemples, nous utilisons une boucle principale qui va se répéter jusqu'à la fin du programme.
Cette boucle contient les actions qui se répètent et qui donnent l'illusion d'un jeu fluide. Elle se décompose de la sorte :

- La mise à jour de l'affichage des éléments de jeu
- La récupération des touches pressé par l'utilisateur
- La mise à jour du comportement des entitées sur le terrains
- Et enfin l'attente d'1/60eme de seconde pour garder un temps régulier entre chaque image et libéré du temps processeur pour les autres programmes.

En ayant effectuer quelques mesures de temps on se rend compte qu'une frame met en moyenne, sur nos ordinateurs personnels 3 millisecondes, à se calculer.
Pour s'approcher le plus d'1/60eme de seconde on soustrait le temps de calcul de la frame simplement en gardant en mémoire le temps système au début de la frame et en le soustrayant au calcul des millisecondes dans la fonction de pause du programme.

Entitées (Joueur, ennemies et amis)

Dans le cahier des charges trois types d'entité sont décrites, le joueur, les ennemis qui peuvent faire du mal au joueur et les amis. Comme ces trois types d'entité partagent les mêmes propriétés entre elles, nous avons essayé de procéder à la délégation.
On utilise une première classe **EntityStat** qui contient les différentes propriétés partagées, comme une position en x et en y, une vitesse, un skin, un nom, une durée d'invulnérabilité…
Ensuite pour les classes joueur et ennemies on leur ajoute un champ du type **EntityStat** en plus de leur particularités.
Cette manière de procéder nous à permis de centraliser les collisions avec les obstacles, les différentes mise à jour de variables comme la prise de dégât ou inversement le gain de vie par exemple mais surtout tous les différents champs. Un exemple concret est lors de notre implémentation des temps d'invulnérabilité, il nous a seulement fallu ajouter deux champs et une fonction de mise à jour pour la voir correctement implémentée.

Enfin on ajoute une interface **Entity** qui permet de s'assurer que chaque classe utilisant un champ **EntityStat** possède toutes les méthodes requise.

Position et boîte de collision

Dans notre programme, les grilles de jeux sont représentées par un tableau statique à deux dimensions de **FieldElement**. Donc pour pouvoir se déplacer librement les entités possèdent des coordonnées en **double**. Cela permet aussi d'avoir un niveau d'abstraction par rapport à l'affichage. La vitesse d'une entité représente le nombre de tuiles par secondes que cette entité parcourt, ensuite à chaque mise à jour on procède à un calcul simple pour déterminer ses nouvelles coordonnées en fonction de la touche pressée.

Cependant avoir simplement les positions n'est pas suffisant pour vérifier les intéractions entre les éléments de jeu. À cela on ajoute les boîtes de collision qui dans notre cas seront exclusivement carrées. Celles-ci vont nous permettre de déclarer si deux éléments dans le jeu entrent en collision. Elles sont mises à jour à chaque déplacement d'une entité. Pour avoir une expérience de jeu un peu plus agréable elles sont légèrement plus petites qu'une tuile (on les à appliquer à 80% de la taille d'une tuile, soit de manière abstraite 0.8)
Avec la programmation orientée objet et la délégation on ajoute une nouvelle classe **Hitbox** à **EntityStat**. Il ne reste plus qu'à tester les collisions lors de la mise à jour d'une frame du jeu.

Entrée utilisateur

Nous nous sommes rendu compte que l'entrée des utilisateurs de zen5 possède un défaut : si on reste appuyer sur une touche les entrées sont irrégulières d'un point de vue des frames. Pour corriger cela on a ajouté une surcouche qui permet de garder en mémoire quelle touche à été pressé et quelle touche à été relâché. Elle consiste en une HashMap<KeyboardKey, Boolean>. À chaque touche est associé un boolean qui indique si elle est pressée ou non. Ainsi à chaque frame on scanne les entrées utilisateur et on met à jour le boolean en fonction de si une touche est pressé ou si une touche est relâché. Ensuite pour les actions relatives aux touches on essaie de produire une fonction par touche, par exemple une fonction pour les actions de la barre espace ou encore une fonction pour la touche i.

Inventaire

L'inventaire est simplement une ArrayList d'**Item**. Nous avons hésité à implémenter une HashMap<Integer, Item> qui aurait permit de placer des **Item** à n'importe quelle place de l'inventaire, mais pour simplifier le développement nous somme rester sur une simple liste ou les éléments de l'inventaires sont forcément consécutif, avec le premier élément de la liste étant comme l'**Item** en main utilisable par le joueur. Une limite de 10 objets est aussi imposée, si la limite est atteinte, il est impossible d'en ramasser de nouveau.

Éléments de map

Les éléments de maps représentent les éléments qui sont placés avec la grille. Ils sont initialisés au moment du parsing de la map.
Comme il peut s'agir d'obstacles ou de décoration nous utilisons une interface **FieldElement** qui permet de regrouper ces deux classes. Cela nous aurait aussi permis d'ajouter d'autres types d'éléments comme les portes scellées ou les éléments temporaires comme les bulles.

Analyseur syntaxique

Cette partie du programme constitue de loin celle la moins réussie. Pour garder en mémoire tous les éléments de jeu nous utilisons une classe **Panel**. Elle contient le joueur, la liste des ennemies, la grille d'élément sur la carte ainsi que la liste des objets posée au sol. Donc pour pouvoir les initialiser on fait appelle au parser, dans lequel nous construisons ces objets. Or comme nos variables sont définies comme finales, nous devons d'abord les construire dans le parser puis les copier dans le **Panel**. Nous tenons à préciser que nous avons demandé conseil et que cette manière de procéder à finalement été validé par notre chargé de TP. Donc notre classe **Parser** contient aussi les champs de Panel.

Pour pouvoir construire ces variables nous parcourons le fichier map token par token et on commence par trouver un bloc, qu'on défini comme étant l'ensemble des champs décrivant un [element] ou [grid]. Ensuite dans cette sous partie du fichier on recommence à le parcourir pour en extraire les valeurs de chaque champ. Ces résultats sont stocké dans une HashMap qui associe un champ sous forme de String (damage, name, size, zone…) à la chaîne de caractère qui suit les deux points. Ensuite en fonction des champs qui ont été remplis on en déduit le type d'objet qu'on veut construire et on ramène vers les fonctions de construction de ces objets.
Ce que nous avons très mal géré sont les différents cas d'erreur possible et les informations à donner à l'utilisateur. Pour garder le compte du nombre de ligne nous devons vérifier à chaque avancement de token s' il s'agit d'un \n par exemple, ce qui complexifie pas mal le programme.
Nous pensons adopter la mauvaise logique pour lire le fichier map mais nous n'avons finalement pas réussi à trouver de meilleur solution et finalement cela nous à beaucoup retardé pour implémenter de nouvelles fonctionnalités.

Découpage des paquets

Pour le découpage des paquets nous avons eu la logique suivante, le programme est découpé en 4 grandes parties :

- L'affichage
- Les entrées utilisateurs
- Le moteur du jeu
- L'analyse d'un fichier map

Nous avons donc naturellement ajouter les paquets fr.uge.display, fr.uge.gameEngine, fr.uge.parser
Comme les entrées utilisateur ne demandent qu'une seule classe **UserEvent**, aucun paquet ne lui est dédié, de même pour les paramètres généraux du jeu (taille de l'écran, taille des tuiles, framerate…) et le paquet qui gère les options entrées par l'utilisateur.
Un autre paquet fr.uge.enums est présent, il contient simplement tous les différents enum que le programme possède (par ailleurs on leur a ajouter une fonction de vérification de présence d'un élément qui renvoie un boolean, cela permet de ne pas faire planter le programme à la première erreur dans le parseur).

Ensuite les sous paquets se sont créés naturellement pour ne pas mélanger toutes les mécaniques du programme entre elles. Par exemple, dans le moteur du jeu les entités et les objets sont séparés dans des paquets dédiés.

Affichage

Le dernier grand point à aborder est l'affichage du jeu. Pour afficher la carte nous devons d'abord opérer une translation si la carte est trop grande et que le joueur est loin d'un bord. Or si nous opérons cette translation en boucle le repère de base sera perdu, cela est problématique pour afficher l'inventaire notamment, donc on garde une copie de la transformation affine du plan. Ensuite pour afficher les images, nous les rechargeons dans une HashMap associant un skin à une image avant la boucle principale de jeu pour ne pas avoir à chercher dans la mémoire morte de la machine pour économiser les performances. Enfin le reste de l'affiche ne comprend que des calculs de position qu'on ne détaillera pas.

Amélioration apporté depuis la soutenance bêta

Plusieurs points ont été soulevés lors de la soutenance bêta. Nous détaillerons ici les points qui ont été corrigés depuis.

Découpage des paquets

L'une des premières remarques qui nous a été adressée est le mauvais découpage des paquets, nous avions trop découpé la sémantique de notre programme et en résultait trop de paquets contenant peu de classes.

Présence de variables public

Lors de notre soutenance bêta nous avions laissé quelques variables en visibilité publique puisque nous n'avions aucun intérêt de les garder privées. Depuis nous utilisons uniquement des getter pour y accéder et elles sont toutes privées.

Javadoc pas claire

Certaines de nos documentation n'étaient vraiment pas claires surtout dans le package **Parser**, nous espérons avoir mieux fait.

Main trop long

Le main à été correctement découper en sous fonctions

Refonte du parseur

Certaines fonctions du parseur vérifiaient si une chaîne de caractère est dans un enum en transformant les enum en liste. Maintenant on utilise une fonction spécifique en O(1) pour savoir si une chaîne de caractère est dans un enum

Ajout d'une documentation

Notre rendu de la soutenance bêta manquait d'une documentation ou d'un README, bien que cela soit obligatoire pour le rendu final, elle est maintenant présente.

Conclusion

Malheureusement nous n’avons pas réussi à implémenter beaucoup de fonctionnalités. Nous avons préféré essayer de correctement corriger et maintenir le code le plus propre possible, et cela ce ressent très fortement sur la quantité d'éléments que nous avons réussi à implémenter. Sur le rendu final uniquement la phase 0 est complètement fini est les éléments de la phase 1, nous avons un inventaire fonctionnel, le système de point de vue opérationnel (avec la nourriture), la carte est centré sur le joueur, et les éléments STICK et SWORD fonctionnent.
Nous pensons que le temps que nous avons perdu est principalement dû à un écart de niveau entre les membres du binôme qui nous demandaient de beaucoup revenir sur le code produit au fur et à mesure de l'avancée du projet, il était important pour nous d’essayer de garder une équité sur la répartition du travail, et nous avons échoué.
Cependant nous pensons sincèrement que le produit actuel fait office de base solide pour implémenter de nouveaux éléments. Par exemple, le système de porte et de clef peut être ajouté simplement en ajoutant un type d’obstacle et un type d’objet. Seul le parser nous limite fortement sur l’ajout de nouveaux éléments comme les amis et leur liste d’éléments à échanger par exemple.