Faculté
des Sciences

Projet d'informatique BAB1 : Boby Is You

Victor Hachard

2018

Table des matières

1 Présentation

Dans le cadre de la première année de bachelier en Sciences Informatiques, un projet de programmation doit être réalisé. Ce projet est un jeu vidéo qui doit être inspiré par BabaIsYou[1]. L'objectif est de réaliser ce jeu vidéo en se basant sur les cours enseignés et d'y appliquer des notions fondamentales telles que l'héritage et la programmation orientée objet [POO].

2 Méthodologie

Pour plusieurs raisons le travail n'a pas été réalisé en groupe, ces raison on été citées lors de la réunion pour les changements de groupe.

La première étape a été de créer un UML du projet en faisant abstraction de l'interface graphique. Ensuite cela a été de développer le coeur du jeu : la modélisation d'un niveau et le système de règles qui sont expliqués par la suite dans la section Décision. Pour implémenter cela, le cours de Programmation et Algorithmique II de M. Quoitin[2] sur la POO a été utilisé. Enfin l'ajout de la partie graphique qui appelle à chaque entrée clavier de l'utilisateur la partie modèle, est expliqué dans la section Décision.

3 Décisions

3.1 Décision sur les packages

Application du design paterne modèle vue qui sépare la partie graphique de la modélisation. Dans la partie modèle, il y a la logique du jeu qui communique directement avec la partie graphique. De plus il y a le package ressources qui comprend des sous-packages : CSS, images, musiques, niveaux, police.

3.2 Décision sur la modélisation d'un niveau

La modélisation d'un niveau est constituée d'un tableau 2D de taille variable qui est une matrice. Cette matrice est constituée d'une liste de taille variable. Cette liste est constituée d'éléments. Chaque élément est défini par un `TypeElement`, des règles et la direction de déplacement du `TypeElement`. `TypeElement` est défini à la section Décision sur la structure des `TypeElement`.

Le tableau 2D de taille variable permet d'être très flexible avec de nouvelles règles, des games modes, ce qui n'aurait pas été possible avec une liste de taille fixe.

L'avantage est une implémentation facile, une très grande liberté et un code très modulaire.

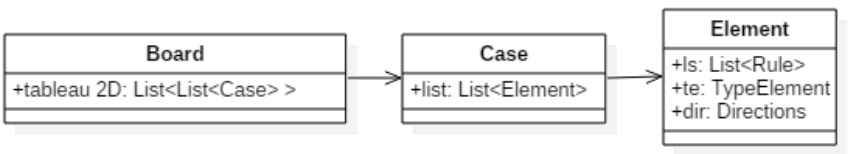

Figure 1 – Illustration de la modélisation d'un niveau

3.3 Décision sur la structure des TypeElement

Un système d'héritage pour la structure des `TypeElement` n'a pas été implémenté. Cette approche en revanche est bien plus modulable et permet la surcharge de méthode.

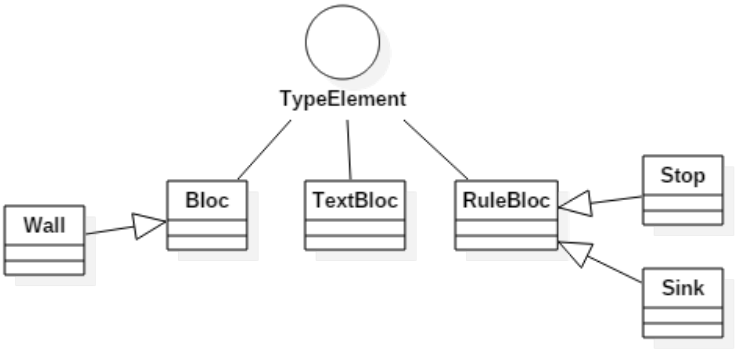

FIGURE 2 – Illustration de la modélisation des `TypeElement` (héritage)

Une énumération a été préférée dans ce cas pour éviter le surplus de classe et un code plus concis. L'énumération d'un `TypeElement` est définie par un ordre de priorité à l'affichage, un nom, un `Type`. `Type` est aussi une énumération qui permet de faire la distinction entre les différents `TypeElement`, par exemple un Type Block, un Type Rule, un Type Text. Si le `TypeElement` est un Text il aura en plus un `TypeElement` et si c'est une Rule il aura une `Property`. `Property` est définie dans la section Décision sur la modélisation des règles.

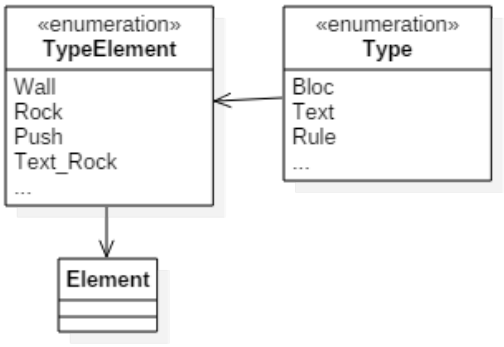

FIGURE 3 – Illustration de la modélisation des `TypeElement` (énumération)

Voici ci-dessous l'énumération `TypeElement`. Pour des raisons de clarté, tout n'est pas implémenté.

```
public enum TypeElement {

    ROCK(2,null,null,Type.BLOCK),
    LAVA(1,null,null,Type.BLOCK),
    PLAYER1(3,null,null,Type.PLAYER),
    TEXT_FLAG(3,TypeElement.FLAG,null,Type.TEXT),
    TEXT_EMPTY(3,TypeElement.EMPTY,null,Type.TEXT),
    PUSH(3,null,Property.PUSH,Type.RULE),
    STOP(3,null,Property.STOP,Type.RULE),
    YOU(3,null,Property.YOU,Type.RULE),
    KILL(3,null,Property.KILL,Type.RULE),
    IS(3,null,null,Type.CONNECTER);

    static TypeElement fromString(String element) {...}
}
```

3.4 Décision sur la modélisation des règles

La modélisation des règles est basée sur un modèle d'héritage : toutes les règles ont une classe parent `Rule` qui est constituée d'une liste chaînée statique. Cette liste chaînée est parcourue à chaque déplacement du joueur. Chaque élément de la liste chaînée est activable ou désactivable, cela évite de parcourir inutilement des règles. Chaque règle contient la méthode `check` qui vérifie si la règle doit s'appliquer et être exécutée. Toutes les règles sont définies par une énumération `Property` pour un souci d'organisation.

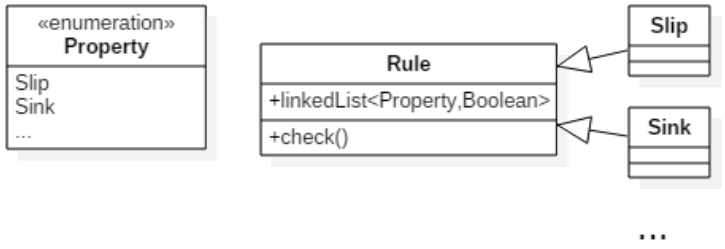

FIGURE 4 – Illustration de la modélisation des règles

Voici ci-dessous la classe parent `Rule`. Pour des raisons de clarté, tout n'est pas implémenté.

```
public abstract class Rule {

    private static Hashmap<Property,Boolean> ...;

    static void setActivity(Property, Boolean) {...}

    public boolean check(Position, Directions, TypeElement) throws WinException {
        boolean res = true;
        if (isActive(this.getProperty()))
            res = work(pos,dir,te);
        if (nextRule == null)
            return res;
        return res && nextRule.check(pos,dir,te);
    }
    abstract boolean work(Position, Directions, TypeElement) throws WinException;
    abstract Property getProperty();

}
```

Voici ci-dessous un exemple de la classe `Win`. Pour des raisons de clarté, tout n'est pas implémenté.

```
public class Win extends Rule {

    @Override
    public boolean work(Position, Directions, TypeElement) throws WinException {
        if "En fonction de la direction, trouve dans la case suivante Win."
            throw new WinException();
        return true;
    }
    @Override
    Property getProperty() {...}

}
```

3.5 Décision sur la structure des fichiers

Les fichiers suivants sont créés à la racine du projet.

- `achievement` sauvegarde de la progression des succès, une sauvegarde automatique a lieu tous les 50 déplacements.
- `crash.log` sauvegarde des erreurs produites par le jeu.
- `config.txt` sauvegarde des préférences et de l'avancement de l'utilisateur, une sauvegarde a lieu après une modification dans le menu `Paramètre` et lors d'un passage de niveau.

4 Points forts

4.1 Optimisations

L'implémentation du projet est entièrement en POO. Ce qui facilite l'ajout de règles, de blocs, de games modes.

Les optimisations les plus importantes en terme de consommation en mémoire ou processeur sont citées ci-dessous.

4.1.1 Optimisation générale

- Une liste `allElement` de taille variable se trouve en attribut de `Board`. Cette liste comprend tous les éléments trouvables dans le niveau chargé, la liste 2D qui contient le niveau est complétée avec des éléments définis dans `allElement`. Cela permet une réduction de charge en mémoire et cela évite de parcourir la liste 2D pour trouver ou changer un élément.
- Le calcul des positions des `IS` n'est exécuté qu'une fois à la construction du niveau, puis si le joueur déplace un `IS` la position est éditée. Cela évite encore de parcourir une liste 2D.
- Tous les niveaux, images, musiques utilisés ne sont chargés qu'une seule fois et gardés en mémoire. Pour éviter d'avoir des lags et chargements.

4.1.2 Optimisation du système de règles

- Le calcul des règles est refait à chaque fois que le joueur déplace quelque chose. Cela permet d'éviter de recalculer les règles si aucune modification n'est apportée sur le plateau. Une solution plus avancée serait d'implémenter le design paterne observeur/observé.
- La vérification et l'exécution des règles. Comme cité à la section Décision sur la modélisation des règles et avec l'exemple des classes `Rule` et `Win`. A chaque déplacement du joueur la méthode `check` dans `Rule` est exécutée. `Rule` comme cité à la section Décision sur la modélisation des règles est constituée d'une liste chaînée qui garde en mémoire toutes les règles du niveau, seules les règles actives sont vérifiées, si une des règles vérifiée s'exécute, les autre ne seront pas vérifiées.
- Lors d'un déplacent du joueur, la vérification des règles de la case suivante et effectuée. Comme dans chaque `Case` il y a une liste d'éléments, la vérification s'exécute sur l'ensemble des éléments. Mais si un des éléments possède les règles en question, nul besoin de continuer pour voir si les autres l'ont aussi. Cela permet un gain de temps important car ces vérifications se produisent très régulièrement.

4.2 Fonctionnalité supplémentaire

— Un game mode : déplacement limité qui est optionnel.
— Un mode coopératif : le premier niveau le démontre comme exemple.
— Nouvelles règles : slip, fly, move, melt, hot, kill, tp, sink, strong et weak.
— Nouveaux blocs : love, lava, ice, skull.
— Nouveaux niveaux.
— Un nouveau connecteur : `AND`.
par exemple : `WALL AND ROCK IS MOVE AND KILL`
— Succès : A chaque fois que l'on se déplace, gagne, recommence, un succès est débloqué. De plus un menu regroupe toutes les statistiques du joueur.
— Musiques : une musique de fond, des bruitages lors des passages de niveau, des déblocages de succès ...

5 Points faibles

5.1 Erreurs d'optimisation

Les erreurs citées sont les plus importantes en terme de consommation de mémoire ou de rapidité d'exécution. Toute les solutions proposées n'ont pas été implémentées par manque de temps ou connaissance.

— L'affichage du niveau dans la partie graphique est refaite à chaque déplacement du joueur. Une solution pourrait être envisagée : l'implémentation du design pattern observeur/observé pour ne rafraîchir que les images qui auraient changé d'état.
— Le calcul des positions du joueur est refaite à chaque déplacement. Une solution moyenne serait de garder en mémoire la position du joueur et l'éditer comme pour les `IS`. Mais le joueur peut changer de `TypeElement` ou de nombre : donc il faudrait détecter quand cela arrive et recalculer toutes les positions.
— La vérification de la position des règles n'est pas implémentée proprement. La vérification n'est qu'un enchaînement de "if", "else" sur cent lignes.

6 Erreur connue et difficultés rencontrées

6.1 Erreur connue

Aucune erreur connue n'a été trouvée, des erreurs peuvent exister dans certain cas avec la règles `MOVE`.

6.2 Difficultés rencontrées

La première difficulté rencontrée fut le système de règles plutôt complexe, lors de la vérification de la position des règles qui n'est pas implémentée proprement, cité dans les Erreurs d'optimisation. Et lors de la création de l'interface graphique, JavaFX[3] qui est très pauvre en documentation, ce qui cause la deuxième difficulté.

7 Remerciements

Je tiens à remercier toutes les personnes qui ont contribué au projet et relu ce rapport.

Tout d'abord, pour les nombreux échanges sur la structure et l'implémentions du projet, Pierre Woi, étudiant à HE2B ESI en deuxième année en Informatique Industriel

Ensuite, pour la correction du rapport, Daniel Hachard, poète.

De plus pour l'environnement du jeu, les musiques, Clément Roart, étudiant à Umons en BAB3 Mathématique. Pour les textures Johan Nevroux, étudiant à l'Human Academy en première année de Manga.

Pour finir, je remercie pour l'aide général qu'il a pu apporté, Thomas Lavend'Homme, étudiant à Umons en première année en Science Informatique.

A Vocabulaire

POO La programmation orientée objet.

UML Langage de Modélisation Unifié.

Connecteur Élément qui connecte un bloc à une règle.

B Guide

B.1 Contrôles

— `pageup` , `pagedown` : augmente, diminue le volume général du jeu.
— De-bug : `P` , `M` permettent de passer les niveaux.
— De-bug : `O` permet de remettre à zéro les games modes [1].
— De-bug : `L` permet de remettre à zéro les règles [2].

Par défaut : pour se déplacer en tant que `YOU` , les touches sont : `UP` , `DOWN` , `LEFT` , `RIGHT` et pour recommencer un niveau la touche est `R` . Cela peut être modifiable dans le menu `Paramètre` .

En coopération pour se déplacer en tant que `P1` , les touches sont les mêmes que `YOU` . Pour le `P2` , les touches sont : `T` , `G` , `F` , `H` , ces touches ne sont pas modifiables.

La touche `ESC` permet de retourner au menu.

B.2 Fichier

Les fichiers : `achievement` , `crash.log` , `config.txt` sont créés à la racine du projet au premier lancement du jeu, leur suppression remet tous les paramètres, succès et l'avancement du joueur à zéro.

B.3 Ant

Attention la commande `ant clean` supprimera aussi les fichiers : `achievement` , `BIS.log` , `config.txt` .

B.4 Ajout d'un niveau

Ajoutez un fichier "TXT" dans code/src/common/ressources/maps. Puis dans la classe java `Levels` ajoutez dans la liste ligne 146 le nom dans le tableau de `char` .

Dans un fichier de niveaux, il faut impérativement inscrire à la première ligne la dimension de la carte. Ensuite optionnellement un titre (exemple : title JeSuisUnSuperInformaticien). Toujours optionnellement une ligne de configuration avec un entier qui représente le déplacement maximum quand le game mode déplacement limité est actif (exemple : config 50).

1. Attention les games modes peuvent ou non se réactiver.
2. Attention les règles se réactiveront au prochain déplacement ou changement de niveau.

Références

[1] HEMPULI : Babaisyou. https ://hempuli.itch.io/baba-is-you, 2017.

[2] Bruno QUOITIN : *Programmation et Algorithmique II*. Presses Universitaires de Mons, 2018.

[3] Oracle CORPORATION : Javafx. https ://www.oracle.com, 2008.