ENSSAT
LANNION

Projet

Programmation Orientée Objet

# Boulder Dash

Colin Leverger
Valerian Saliou

Informatique, Multimédia et
Réseaux
*Promotion 2017*

Destinataire : Damien Lolive

27 juin 2015

# Table des matières

# 1. Introduction

Dans le cadre de la deuxième partie du module de programmation-objet dispensé à l'ENSSAT, spécialité Informatique, Multimédia et Réseaux, nous avons été amenés à développer un jeu dans le langage java en binôme. Nous avons imaginé et développé un moteur de jeu en clonant le célèbre "Boulder Dash" ; nous avons utilisé pour ce faire un paradigme de programmation connu sous le nom de *modèle MVC*. Ce modèle permet de construire son application en séparant totalement les données des vues, dans un souci d'architecture rationnelle et optimisée.

Le présent rapport explicite la méthode que nous avons suivie pour développer ce jeu. Nous allons tout d'abord y présenter le jeu, pour situer le contexte et les enjeux. Une seconde partie donnera un aperçu de la modélisation et de la structure interne de l'application. La troisième partie sera consacrée à l'aspect purement technique de la réalisation, et des fonctions importantes y seront explicitées. Nous conclurons enfin en quatrième partie en prenant du recul sur l'application et sur les choses qui auraient pu être développées d'une autre manière.

# 2. Présentation du jeu

Boulder Dash est un jeu vidéo conçu par Peter Liepa et Chris Gray, et édité par First Star Software à partir de 1984. [1] Ce jeu est rapidement devenu une référence dans le domaine des jeux d'arcades. Le gameplay accrocheur et assez simple a fait énormément d'émules. Le but du jeu est de mouvoir un personnage au sein d'un plateau 2D ; ce personnage évolue dans un terrain rempli d'éléments tels que de la terre, des murs, etc. Il peut traverser la terre pour bouger et se rendre à un autre endroit du plateau. Le gameplay est basé sur la recherche d'éléments rapportant des points, les Diamants. Mais le chemin du personnage est bien sûr semé d'obstacles et d'ennemis... Il faut donc que le joueur essaie de bouger son héros, Rockford, sans heurter de pierre et sans croiser d'ennemis. Il est demandé dans le sujet de créer une copie de ce jeu. Il est aussi demandé de développer un éditeur de jeu ; ce dernier permettra à l'utilisateur de créer ses propres parcours du combattant (et pourquoi pas de les partager avec ses amis ?) Certains éléments du jeu sont animés. En effet, quand le personnage bouge, on doit avoir l'illusion à l'écran qu'il "marche" dans la direction voulue. On appelle ce procédé *l'animation de sprites* ; cela permet de rendre le jeu plus dynamique et surtout plus réaliste.

Certains éléments du plateau sont destructibles, d'autres non. Il faut définir avec soin les caractéristiques de chaque objet à modéliser pour ne pas oublier de détails dans l'implémentation. Nous avions pour cela un site web répertoriant les différents attributs de chaque objet.

Pour avoir une interface ergonomique et simple d'utilisation, il était demandé dans le sujet de décomposer son application en 3 parties graphiques : un panel permettant de créer une nouvelle partie / charger une partie / mettre sur pause, un autre panel permettant d'afficher le plateau de jeu et un panneau affichant les informations du jeu (score, highscore, etc.) Il ne nous était évidemment pas demandé de créer un gameplay et un jeu commercialisable. Le but ici était surtout d'asseoir nos connaissances en programmation-objet, et surtout de nous permettre d'appréhender et de manipuler le modèle MVC.

1. Source : Wikipedia

# 3. Conception & choix de réalisation

## 3.1 Le modèle

Comme expliqué plus haut, le modèle MVC permet de décomposer son application en 3 parties bien distinctes, qui vont communiquer entre elles. Les 3 parties seront les suivantes :

- La vue : ce qui affiche les éléments graphiques.
- Le modèle : toutes les données de l'application sont stockées ici.
- Le contrôleur : il lie la vue et les données et coordonne le processus d'affichage / lecture de données.

Le package Model sera composé de classes visant à modéliser et représenter les différents éléments affichables sur notre terrain de jeu. Concrètement, une super classe "DisplayableElementModel" sera présente, et tous les éléments affichables hériteront, avec leurs attributs propres, de cette classe. Les attributs seront simples : "isDisplayable", "canMove", "isDestructible"... tout ceci pour représenter les actions possibles par l'élément affichable. Il y aura une classe par élément affichable, et ces classes hériteront toutes de "DisplayableElementModel".

Toujours dans ce package Model, il était nécessaire d'avoir une classe "LevelModel". Il s'agit en effet de stocker les informations sur le niveau en cours, tel que la disposition des différents éléments sur le plateau, pour pouvoir les afficher. C'est dans cette classe que le modèle de jeu évoluera au fil du temps et des actions de l'utilisateur. Dès que le personnage bougera, le terrain devra évoluer, et sera donc mis à jour dans cette classe. Le terrain et ses éléments seront stockés sous la forme d'un tableau à double dimension. Chaque indice du tableau représentera un élément affichable. Une classe "GameInformationModel" se chargera de stocker les informations présentes sur chaque partie distincte : les scores, les diamants restants...

La matrice ci-dessous représente le terrain de jeu tel que nous le stockons ; nous l'affichons en console uniquement à des fins de débug.

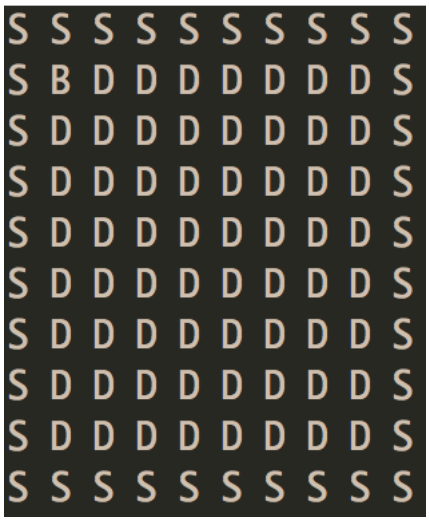


Figure 3.1 – Ground View

À noter, pour stocker le terrain de jeu nous sommes partis sur la lecture et l'écriture d'un fichier XML. Il nous semblait en effet intéressant de modéliser le terrain sous la forme d'une matrice de 30x30 carrés de 16px, et d'indiquer dans un fichier XML où étaient placés chaque élément. Ceci implique le développement d'un parseur XML et d'un interpréteur de fichier de jeu. Il s'agit en effet de ressortir les informations fournies par le fichier XML sous forme de tableau, directement exploitable dans le programme. Valérian s'est chargé de développer l'aspect XML.

Le package Controllers sera composé de plusieurs éléments. Un contrôleur va se charger de lier le model et l'affichage du jeu en lui même. Il s'agit en effet de détecter les actions sur les boutons du jeu, tels que sur le bouton "Nouvelle Partie" ou "Quitter", et d'agir en conséquence de manière décentralisée.

Un second contrôleur permettra de gérer les éléments clavier et de détecter sur quelle flèche le joueur appuie. Pour le déplacement du personnage virtuel, il faut évidemment détecter le sens de déplacement voulu, et c'est cette classe qui s'en occupera. Une troisième et quatrième classe contrôleur permettra de gérer l'éditeur de niveau. Deux contrôleurs se chargeront de modifier les éléments dans le modèle en fonction des évènements : chute des boulders, déplacement du personnage... Ces deux contrôleurs seront en fait des threads qui scruteront le terrain et effectueront des modifications en fonction des différents paramètres : s'il y a du vide sous ce diamant, le faire tomber, etc. Un ultime contrôleur se chargera de la navigation entre les différentes vues que nous avons mises en place.

Le package views se chargera quant à lui d'afficher le modèle, avec l'aide des contrôleurs qui lui renverront des informations. C'est dans ce package que sera rafraîchie l'image du jeu et que les interfaces seront dessinées. Il faut bien évidemment que les vues implémentent "Observers" ; dès que le modèle est mis à jour, il faut que les vues se mettent elles aussi à jour !

## 3.2 Méthodologie de développement

Dans un souci de productivité, nous nous sommes réparti les tâches comme suit :

### 3.2.1 Première étape de développement

- Valérian : pose de la structure globale du projet, initiation du git, parseur & intégration XML.
- Colin : création du modèle pour chaque personnage, initialisation de l'affichage, création des fenêtres.

À la fin de cette étape, nous pouvions faire bouger un élément dans le plateau de jeu, et notre personnage principal était capable d'évoluer dans la terre.

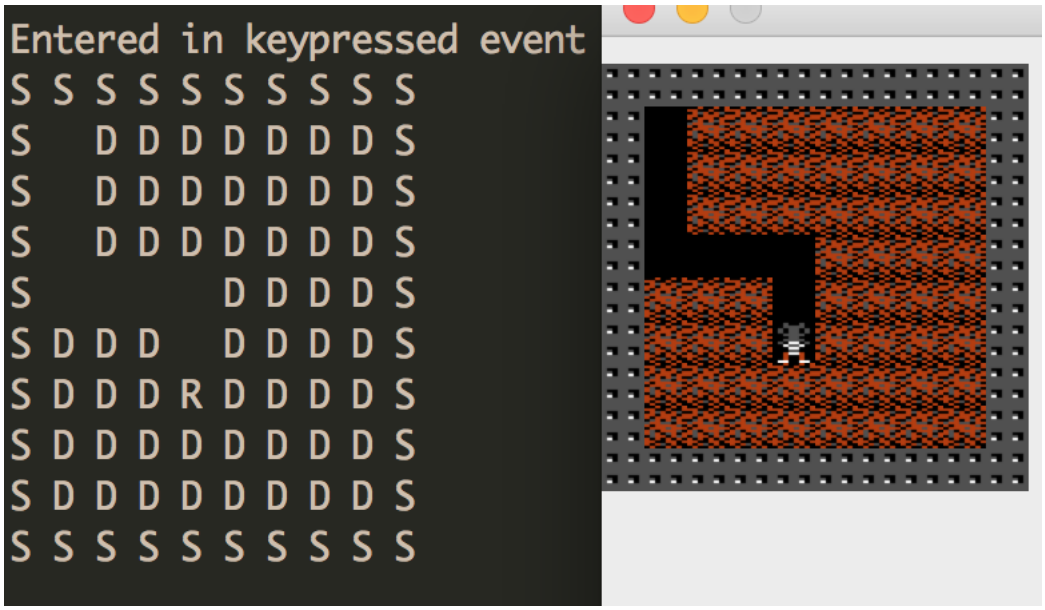


Figure 3.2 – Ground View with Rockford

On peut observer ci-dessus la concordance entre la matrice stockée dans la clase "LevelModel" et l'affichage du personnage.

Une fois que toutes ces tâches furent terminées, nous avons fait un point, mergé le code et fait un refactoring.

### 3.2.2 Seconde étape de développement

— Colin : Animation des sprites. Finalisation de l'interaction contrôleur / vue et gestion des évènements.
— Valérian : Création de la vue d'éditeur de niveau.

Lors de cette deuxième étape, Colin s'est chargé de parfaire le modèle ; nous nous sommes vite rendu compte qu'il était très important d'avoir un modèle cohérent et bien conçu. Nous allons, lors du scan de la matrice, utiliser du polymorphisme pour l'affichage et l'actualisation de chacun des éléments de la matrice. il s'agit en effet de mettre à jour la position d'un objet boulder s'il tombe, etc. Comme tous les éléments affichables héritent d'une superclasse "DisplayableElementModel", il fallait que tous les attributs utiles pour définir un objet et le faire bouger en conséquence soient définis. Lors de l'avancée de cette étape, nous avons dû effectuer plusieurs refactoring, car il nous manquait certains attributs...

### 3.2.3 Troisième étape de développement

— Finalisation jeu et collisions.
— Finalisation interface : il s'agit ici de mixer les différentes vues en une seule et d'effectuer les derniers réglages d'ergonomie.
— Finalisation éditeur de niveaux.

### 3.2.4 Dernière étape ... Les tests !

Pour tester notre application, nous avons validé une à une les différentes possibilités de jeu ; il fallait vérifier un certain nombre de critères. Le diamant qui tombe sur le personnage va le tuer, le rocher qui tombe sur un mur en brique va casser le mur, etc. De ces tests sont ressortis quelques problèmes et nous avons pu les résoudre en conséquence. Nous avions pensé qu'il était intéressant pour le projet de faire des tests unitaires. Nous nous y sommes pris un peu tard, et il était devenu très compliqué de l'intégrer à notre projet ; en effet, le workflow classique est plutôt de développer les tests au début de l'application, avant même les fonctions, etc. La cohérence du modèle n'est donc pas testable via JUnit. Peut-être dans une prochaine version ?

Remarque amusante :

Nous avons mis notre projet en ligne sur Github. Comme nous avons suivi les standards internationaux et codés en anglais, notre projet a été repéré par un étudiant israélien, qui nous a proposé

d'ajouter à notre jeu une intelligence artificielle (pour gérer les mob par exemple.) To be continued ? (À noter, nous livrons évidemment un projet 100% Colin-Valérian ; le développement de l'IA se fera sur une branche à part.)

Github du projet : https ://github.com/valeriansaliou/boulder-dash

# 4. Développement technique

## 4.1 Boucle de jeu & thread

Il existe plusieurs solutions pour la boucle de jeu. La plus optimisée nous semblait l'utilisation de threads. Cette solution est certainement la plus légère et la plus adaptée pour un jeu de ce type. L'avantage des threads est que Java les gère très bien, et qu'ils peuvent lire "simultanément" [1] dans une même variable sans qu'il y ait de conflit. Dans notre cas, il était nécessaire d'effectuer plusieurs scans du tableau à deux dimensions contenant les différents éléments affichables pour mettre à jour les vues. Nous avons donc utilisé un thread par tâche : un thread pour update l'affichage des sprites, un thread pour le mouvement de Rockford et un dernier pour la chute des objets. À noter, il était possible de n'utiliser que deux threads mais cela nous posait un problème : en effet, les rochers doivent tomber moins vite que Rockford quand il court ; si Rockford et les boulders sont dans le même thread, cette condition ne pourra pas être respectée (a moins d'utiliser des algorithmes (trop) complexes...)

## 4.2 Gestion des sprites

Certains objets affichables sont animés. Pour afficher les sprites, nous avons rafraîchi tous les éléments du tableau toutes les 25 ms. L'animation est donc fluide et gérée au sein d'un seul thread.

La procédure est la suivante :

— Un thread qui s'effectue toutes les 25 ms se lance dans le modèle "levelModel". Dans ce thread, le tableau a deux dimensions de DisplayableElementModel représentant l'état actuel du terrain va être rafraîchis toutes les 25ms via la méthode "this.updateSprites(x, y) ;". Cette méthode permet d'utiliser la fonction "update() ;" contenue dans chaque objet Dis-

1. casi simultanément en vérité...

playableElementModel.
- Utiisation du polymorphisme :
  - Si l'élément concerné est un élément animé, alors on va updater l'image avec une fonction présente dans la sous-classe : "RockfordElementModel", "DiamondElementModel" ou "MagicWallModel".
  - Sinon, l'image reste la même ! Pas d'action à effectuer...

Nous avons pris la liberté de lancer un thread au sein du modèle, car il s'agissait pour nous d'une modification typiquement indépendante du joueur. Les sprites vont en effet s'actualiser même si le joueur n'effectue aucune action...

## 4.3 Gestion des objets qui tombent

La gestion des objets qui tombent est incluse dans le contrôleur "BoulderAndDiamondController". Il s'agit là encore d'un thread qui effectue de manière périodique un rafraîchissement du tableau.

La méthode peut donc se décomposer comme suit :

- Scan du tableau.
- Mise à jour de la position des éléments.
- Pause ...
- ... Et rescan.

À noter, plusieurs particularités lors de cette fonction :

- Pour avoir une chute linéaire et contrôlée des éléments, il était important de scanner le tableau *à l'envers* (x– / y–). En effet, en scannant le tableau dans le sens habituel, un rocher tombait en un tour de boucle autant de cases vides qu'il avait en dessous de lui... Trouver la solution à ce problème nous a pris un certain temps.
- Nous avons géré les intersections entre les objets en nous basant sur les noms des sprites des différents objets. Autrement dit, pour ne pas utiliser "d'instanceOf()", nous avons par exemple vérifié si l'objet en dessous de ce "boulder" avait pour spriteName "black" (vide), ou "Rockford", etc. C'est la manière la plus propre et la plus rapide que nous avons trouvée pour récupérer le type d'objet / le surrounding de l'objet à traiter.
- Nous n'avons pas utilisé la fonction "rectangle intersect" suggérée dans le sujet. Nous avons préféré utiliser un certain nombre de vérifications "if based" ; ce n'est certainement pas la technique la plus élégante mais elle reste tout à fait fonctionnelle. Le problème de cette technique est qu'elle implique de créer une multitude de "if" embriqués. (et ça, on n'aime

pas trop)
— A chaque modification de place d'un objet dans le modèle est associée une fonction : moveThisBoulderToLeft(x,y) ; , makeThisBoulderFall(x,y) ; ... Les modifications de position en eux mêmes sont donc effectuées dans le modèle.

## 4.4 Gestion de Rockford

Rockford a son contrôleur dédié, "RockfordUpdateController" ; celui-ci dispose d'un troisième thread qui, une fois lancé, va mettre à jour la position de Rockford dans le modèle si ce dernier a bougé. Rockford est aussi contrôlé par le contrôleur "GameKeyController".

Nous pouvons décomposer la mise à jour de la position de Rockford comme suit :

— Quand on appuie sur une flèche directionnelle, la position de Rockford au sein de "RockfordUpdateController" est modifiée et un flag "hasMoved" est mis à true.
— Si le flag "hasMoved" est vérifié, le thread modifie la position de Rockford dans le modèle.

L'inconvénient de ce mode de fonctionnement est qu'il fallait stoker la position de Rockford a la fois dans le modèle et dans le contrôleur, il y a donc une redondance...

## 4.5 Les vues

Nous avons utilisé le mécanisme d'Observer/Observé dans ce projet. La vue observera donc le "LevelModel" et se rafraîchira à chaque modification de ce dernier. Comme nous animons les sprites toutes les 25ms, la notification de la vue se déroulera après chaque rafraîchissement.

## 4.6 Chargement d'un niveau

Le chargement des niveaux est réalisé via l'outil helper LoadLevelHelper. Il permet de désérialiser un niveau stocké sous la forme d'un XML, vers le format de représentation interne de la carte du jeu.

L'outil parse l'XML en utilisant les librairies XML du W3C fournies avec Java 1.8 (org.w3c.dom.*). Le fichier XML n'est lu qu'une seule fois, lors de l'initialisation du niveau cible. Ensuite, le jeu se déroule depuis la représentation interne de la carte, qui stocke des objets du modèle. Celle-ci mute depuis son état initial lorsque le jeu évolue.

Il est à noter que le système de chargement n'est pas parfait. Il est très facile d'obtenir des erreurs de chargement, notamment en raison de la désynchronisation de la taille de la map (éléments <width /> et <height />) avec les données de la grille (élément <grid />). Il suffit qu'il n'y ait pas suffisamment de noeuds enfants dans cette même grille pour que le chargement retourne une exception, ce qui empêchera le modèle d'être affiché par le système d'affichage du levelModel en raison de NullPointerException.

## 4.7 Sauvegarde d'un niveau

La sauvegarde du niveau se déroule de la même façon que le chargement, de façon inversée, au sein de l'helper LevelSaveHelper. Le processus est plus fastidieux, car la création d'éléments XML imbriqués est, en Java, très verbeuse.

Partant du principe qu'une carte chargée en tant que modèle ne mute qu'en cas de placement d'un nouveau bloc sur la carte, aucun état intermédiaire (déplacement d'un bloc lié au jeu) ne peut venir perturber la sauvegarde.

Ainsi, on admet que le modèle se trouve dans un état "propre" avant sauvegarde. Le système de sauvegarde boucle simplement sur la donnée et converti les objets modèles en leur représentation XML (un nom, et une nature, et dans certains cas si oui ou non l'objet est convertible après passage dans un Magic Wall - c'est le cas des Boulders).

## 4.8 Gestion du placement des objets dans l'éditeur

Un choix a été fait de ne contrôler l'éditeur que via le clavier. Ce, pour des raisons de productivité de l'utilisateur, qui contrôle déjà une bonne partie du jeu via le clavier.

L'utilisateur peut sélection l'objet sur le sélecteur de gauche, puis il sera autofocus sur la carte de jeu en édition. Ensuite, une simple pression de la touche ESPACE remplacera le bloc surligné. Le déplacement est réalisé avec les flèches du clavier.

Pour tracer une ligne d'éléments, il est pratique de ne pas à avoir à appuyer plusieurs fois sur la barre d'espace. Ainsi, un mode de retenue peut être activé par pression de la touche MAJ du clavier, pour placer l'objet sélectionné en continu.

D'un point de vue technique, l'éditeur réaliser de très simples opérations sur le modèle. En effet, la touche ESPACE pressée, il convertit la valeur du bloc en élément model (de superclasse DisplayableElementModel), puis l'insère en position courante dans la grille.

Lorsque la sauvegarde est demandée, il suffit de passer le modèle muté au LevelSaveHelper, pour sauvegarde immédiate.

## 4.9 Gestion de l'XML pour stocker les niveaux

Le XML a été un choix judicieux pour ce projet, en raison de la possibilité d'éditer directement les niveaux de manière simplifiée, lors du développement du mode "jeu", avant même que l'éditeur ne soit réalisé (qui aurait été trop complexe si nous étions partis de zéro sur cet aspect).

## 4.10 Structuration du projet

Le projet a été structuré judicieusement en regroupement de classes selon leurs rôles :

- Bridges - "Ponts" vers des librairies, permettant une utilisation simplifiée au sein du code.
- Controllers - Contrôleurs des fenêtres, ainsi que des handlers d'évènements claviers.
- Exceptions - recense les exceptions spécifiques au jeu, pouvant être levées au sein du jeu, et uniquement en son sein.
- Helpers - Utilitaires non liés aux modèles ou aux contrôleurs, peuvent être instanciés de plusieurs endroits distincts.
- Models - Modèles, cartographiant les blocs de jeu, ainsi que la matrice de la map de jeu.
- Views - Vues, spécifiant les fenêtres de jeu ainsi que les parties des formulaires utilisés.

Ceci est bien entendu valable pour les projets de petite envergure, tel celui-ci. Un projet plus large serait bien mieux structuré par sous-packages, regroupement les vues, modèles, helpers, factories directement liés au package.

# 5. Conclusion

| FONCTIONNALITES | ATTENDUE | NON ATTENDUE | IMPLEMENTATION |
|---|---|---|---|
| | | | |
| **Editeur de niveau :** | | | |
| Placement objets sur jeu | x | | x |
| Sauvegarder niveau | x | | x |
| Charger niveau sauvegardé | x | | x |
| Placer un seul rockford par niveau | x | | x |
| Avertir & informer l'user sur ses actions | | x | x |
| | | | |
| **Jeu :** | | | |
| Déplacement personnage | x | | x |
| Terre qui se détruit | x | | x |
| Boulder qui tombe | x | | x |
| Bouder qui glisse | x | | x |
| Diamant qui tombe | x | | x |
| Diamant qui glisse | x | | x |
| Score | x | | x |
| Nombre de diamants restants | x | | x |
| Ennemis | | x | |
| Annimation sprites | x | | x |
| Pause | x | | x |
| New Game | x | | x |
| Perdre / gagner | x | | x |
| Rockford ecrasé par objets tombant | x | | x |
| Expanding Walls | | x | x |
| Spawn porte de sortie aléatoirement | | x | x |
| | | | |
| **Developpement :** | | | |
| Modèle MVC | x | | x |
| Observer/Observable | x | | x |
| Modèle cohérent | x | | x |

Figure 5.1 – Recette du projet

Voici la recette de notre application. Les éléments que nous avons tenté de développer, mais que nous n'avons pas réussi à finaliser, car manque de temps sont les suivants :

— Problème d'affichage du prompt à la fin du jeu : une fois sur deux, la popup ne s'affiche pas

lorsque le joueur gagne. Nous savons pourquoi : comme ce qui spawn la popup est dans une view qui est "Observer", le modèle n'a pas le temps de notifier la vue lorsque le joueur gagne, et la popup n'est donc pas affichée. Nous avons essayé de re-notifier la vue après le changement d'état du jeu (passe de gameRunning = true a gameRunning = false), essayé de mettre une temporisation, mais cela ne fonctionne pas.

— Un certain nombre de détails purement fonctionnel du jeu n'ont pas été implémenté pour cloner à 100% le jeu original : certaines chutes ne fonctionnent pas exactement de la même façon, le jeu ne possède pas de Timer, les boulders ne se changent pas tous en diamants dans le jeu ... Nous nous sommes en quelque sorte rapproprié le jeu et nous l'avons construit en prenant certaines petites libertés.

L'application que nous avons développée est fonctionnelle et remplit bien le cahier des charges. Le pattern MVC semble bien respecté, et nous avons fait attention à utiliser aussi le patern Observer/Observable. Certains détails peuvent être discutables. Par exemple, la solution que nous avons choisie ne donne pas de coordonnées X,Y a nos objets affichables. Autrement dit, nous avons fait le choix de stocker les coordonnées de chaque objet instancié dans l'objet ground[x][y] de la classe "LevelModel" plutôt que dans l'objet lui même ; il aurait été possible de développer complètement différemment, et tout baser sur le polymorphisme et sur les propriétés internes (position x,y) de chaque objet stocké au sein même de l'instance de l'objet. Notre solution implique une classe "LevelModel" qui doit se charger de toutes les modifications du modèle, et cela en fait une classe très conséquente : près de 830 lignes de code !

Une modification possible serait aussi de rendre la taille de notre map générée modifiable par l'utilisateur. Actuellement, la map est fixée à 30x30.

Le développement de ce projet nous a permis de nous plonger dans le monde objet et java en profondeur ; il nous a permis de gagner en autonomie dans le développement et la lecture de documentation technique, et nous a aussi bien amusés : développer un jeu est quelque chose de motivant et de très visuel. Nous espérons que le projet continuera d'être suivi sur github !