Rapport
Architecture Logicielle
Jeu 2D autour d'un framework
Année 2015/2016

Raphaël Jorel, Antoine Laulan

2 février 2016

1 Introduction

Notre jeu a été produit dans le cadre de l'UE architecture logicielle. Le but étant de créer un jeu à l'aide d'un framework fourni et cela sans modifier celui-ci. Dans ce rapport nous présenterons notre jeu et nous fournirons une critique du *framework* utilisé.

Comme le sujet était libre, nous avons choisi de recréer une copie simplifiée du célèbre jeu Breakout. Simple en apparence, mais qui nous aura tout de même donné du fil à retordre sur certains aspects que nous ne soupcionnions pas. Nous dédierons une partie à l'explication de ces problèmes rencontrés.

2 Firewall Breaker

2.1 Présentation

Tout d'abord, une petite capture d'écran afin d'apprécier le rendu du jeu et des *assets* qui ont été utilisés :

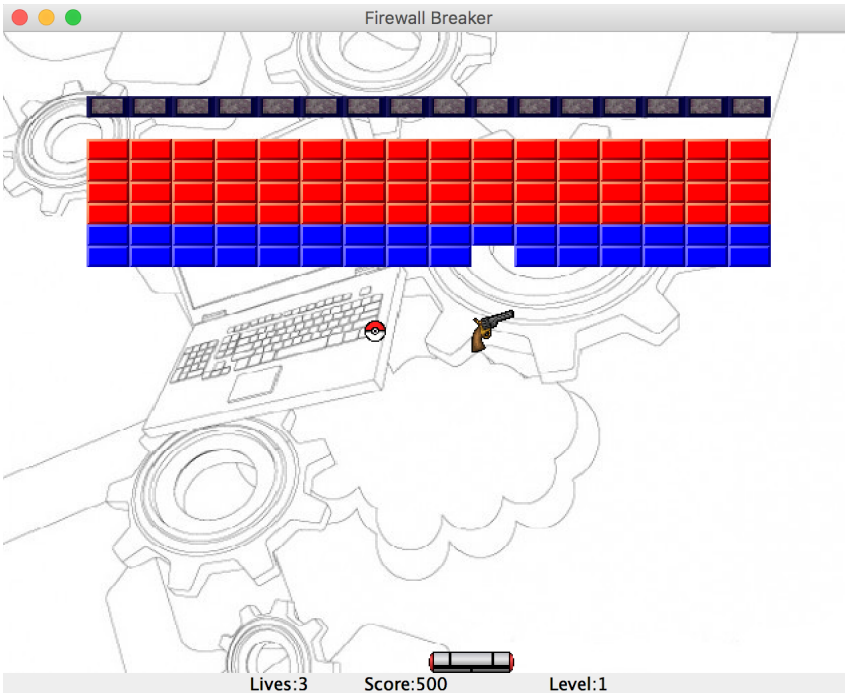

FIGURE 1 – Vue du jeu

Le joueur contrôle une palette et doit, en faisant rebondir la balle, détruire les briques pour gagner. Pour se faire il peut être aidé de divers bonus et briques spéciales. La partie s'arrête lorsque le joueur a détruit toutes les briques qui peuvent l'être, ou lorsque celui-ci n'a plus de vie.

2.2 Fonctionnalités du jeu

2.2.1 Le joueur

Le joueur ne peut faire que deux choses : aller à droite ou à gauche. Il doit rattraper la balle qui se déplace pour la relancer dans le jeu et ainsi casser les briques. Comme sus-dit, il pourra être aidé par différents bonus, et ainsi augmenter sa capacité de destruction.

2.2.2 La balle

La balle se déplace et rebondit sur les briques, les murs (invisibles sur les côtés) et le joueur. Ses rebonds gardent les mêmes angles lorsqu'elle rebondit sur les murs et les briques, mais suivant l'endroit où elle touche le joueur, elle gagne ou perd en vitesse.

2.2.3 Les briques

Il y a quatre sortes de briques :
— les basiques : elles n'ont aucun effet particulier, mis à part le fait de rapporter des points lorsque la balle les détruit,
— les incassables : elles font barrage à la balle et ne peuvent être détruites,
— les briques de bonus : elles déclenchent l'apparition aléatoire de bonus lorsqu'elles sont détruites,
— les briques explosives : elles explosent lorsqu'elles rentrent en contact avec la balle et cassent toutes les briques situées autour d'elles et ceci dans un rayon de taille aléatoire.

Lorsque toutes les briques sont cassées, le niveau est terminé. Toutes les briques cassables ne nécessitent qu'un seul coup pour être détruites.

2.2.4 Les bonus

Afin de varier un peu le jeu et de coller un peu plus à l'esprit d'un Breakout nous avons intégrer la possibilité au joueur d'utiliser des bonus. Ceux-ci ont uniquement un effet "positif" sur la parite du joueur. Voici la liste des bonus mis à la disposition du joueur :
— **life Bonus** : représenté par un coeur, caractérisé par le rajout d'une vie supplémentaire,
— **weapon Bonus** : représenté par un pistolet et caractérisé par le tir d'une salve (limitée dans le temps) de balles depuis la palette du joueur en direction des briques,
— **bomb Bonus** : représenté par une bombe. Ce bonus à pour effet de transformer une brique restante en brique explosive et de la faire exploser. L'explosion a le comportement d'une brique explosive classique.
— **fireball Bonus** : représenté par une boule de feu. Il provoque un changement d'état temporaire de la balle (*fireball*) qui ne rebondit alors plus sur les briques mais passe à travers tout à l'exception des briques incassables.

Chacun de ces bonus est associé à une probabilité, ainsi par exemple le bonus de vie supplémentaire a moins de chance d'apparaître que les autres, car le nombre de vies est un élément cruxial à prendre en compte, et que nous ne souhaitons pas que le jeu soit trop facile, cela va de soit. Nous proposerons également au joueur l'achat de futurs DLC pour obtenir de nouveaux bonus encore plus dévastateurs..

2.2.5 Gestion des parties

Il est toujours agréable de pouvoir recommencer une partie, sans etre obligé de relancer un programme. Pour cela, la fonctionnalité *new (game)* a été apportée, en plus de la possibilité de quitter. Cependant, il est impossible de suspendre une partie et de la reprendre. Il faut donc que vous soyez vraiment sûr de ne pas être dérangé et de n'avoir rien besoin de faire pendant vos parties (que l'on espérera enflammée, évidemment).

2.3 Architecture

Un bon jeu implique une bonne architecture. L'inverse est moins évidente.. Nous allons dans cette partie vous présenter l'architecture logicielle de notre jeu. Ceci sera fait à l'aide de diagrammes de classes. Nous metrons en évidence les connexions entre nos classes et les classes du framework.

2.3.1 Diagrammes de classe

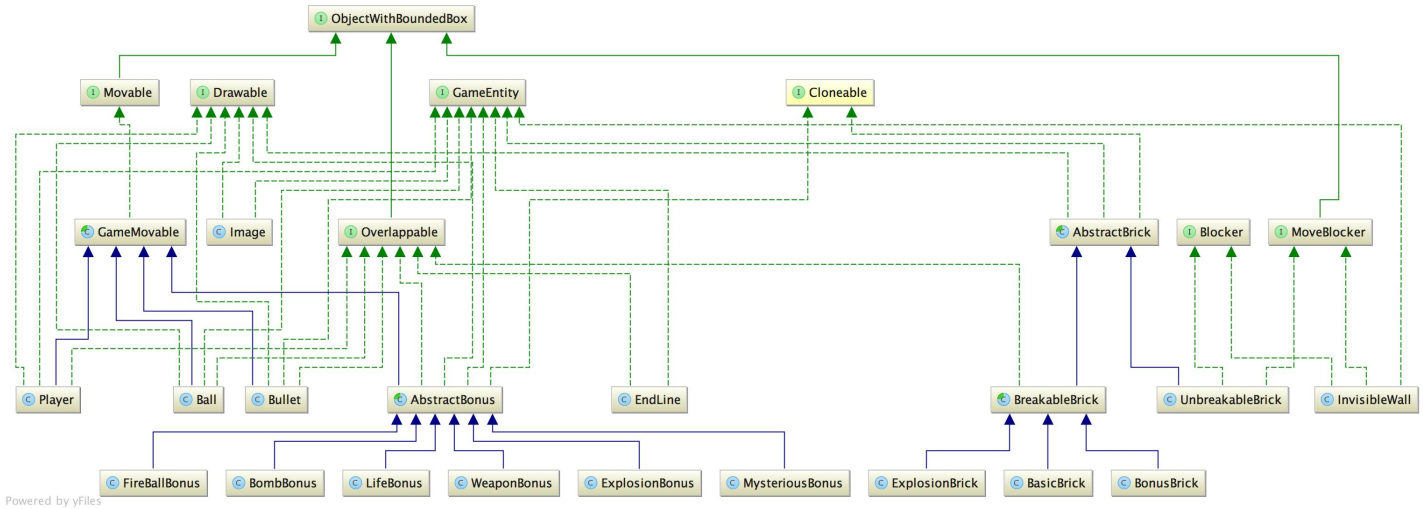

FIGURE 2 – Diagramme de classe des entités

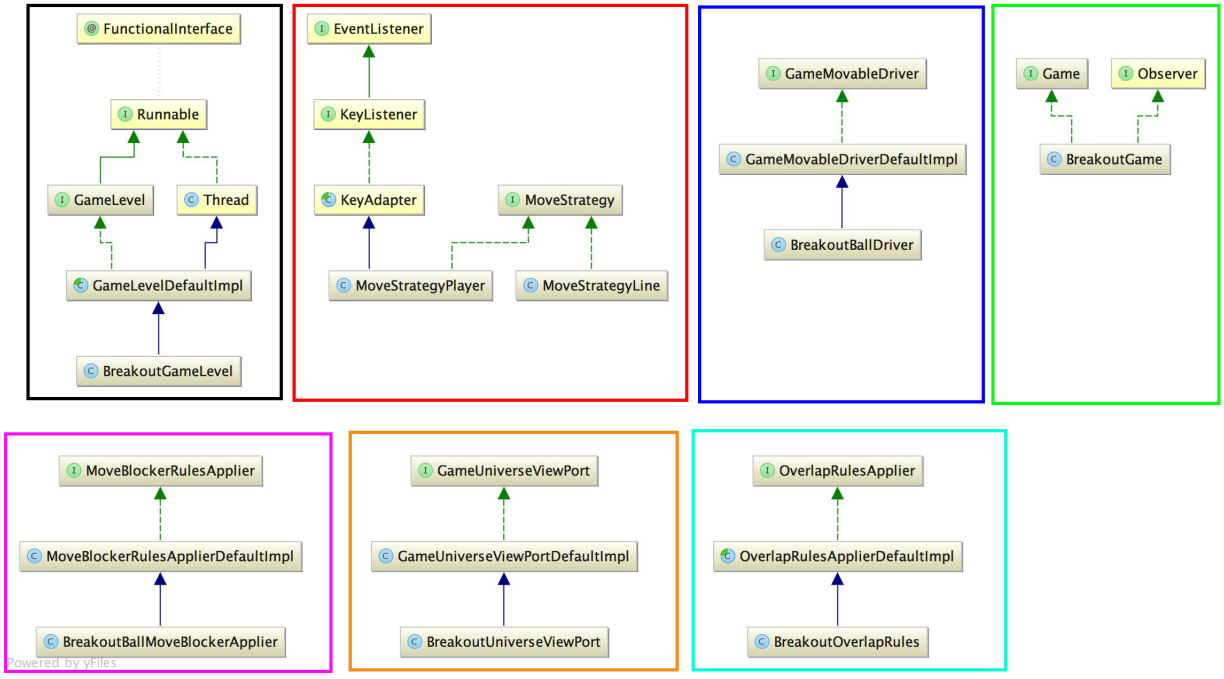

FIGURE 3 – Multi diagramme de classe

Description : Plusieurs classes forment de petits diagrammes de classes. Nous les avons regroupés dans une seule est même image. Pour les différencier nous avons utilisé un code couleur. Ce code couleur est détaillé dans la liste suivante :

— noir : gestion des levels du jeu,
— rouge : stratégie de déplacement du joueur,

— bleu : driver de la balle,
— vert : gestion de la fenêtre IHM du jeu,
— rose : move blocker pour la belle,
— orange : gestion du fond d'écran,
— bleu ciel : gestion des overlappables.

2.4 Implémentation

Après cette présentation des fonctionnalités et de l'architecture du jeu, nous allons dans la partie suivante parler de son implémentation et donc du travail qui a été effectué.

2.4.1 Les niveaux

Afin de ne pas avoir à écrire statiquement la configuration des briques d'un niveau, nous avons utilisé des fichiers externes, au format *.txt*, pour le faire. Chaque type de brique est représentée par un numéro. Théoriquement, il est possible d'avoir un nombre de brique en largeur et en hauteur aussi grand qu'on le souhaite, il faut juste préciser cela dans au début du fichier de description, mais pour des raisons de jouabilité, il est recommandé de se limiter à des grilles de 16x16. Ainsi, la balle a toujours la place de bouger entre les murs exterieurs et les briques.

Pour éviter d'instancier chaque brique une à une, nous nous sommes servis du *design pattern prototype*, cela permettant d'avoir les briques de bases dans un tableau et ensuite de les cloner en fonction du fichier de description.

2.4.2 Stratégies de déplacement

Les stratégies permettent de définir comment un objet va se déplacer. Pour notre jeu, nous n'avons eu besoin que de deux stratégies : celle du joueur et celle des autres entités.
Les autres entités se déplacent verticalement, vers le bas pour les bonus et vers le haut pour les balles de pistolet. Le joueur quant à lui bouge suivant l'action de l'utilisateur sur le clavier. Son déplacement est uniquement horizontal.

2.4.3 Fenêtre de jeu

En s'inspirant de l'exemple d'utilisateur du *framework*, nous avons créé une fenêtre de jeu semblable, mais évidemment plus appropriée au jeu de *Breakout*. La possibilité de recommencer une partie et de la quitter font parties des deux seules fonctionnalités de la fenêtre d'affichage.

Afin de gérer la fonction *new*, nous avons remanié l'exemple donné pour lui permettre de fonctionner correctement. Nous gérons les niveaux dans une file de *threads* qui attendent d'être lancés. Lorsque la file est vide, le jeu se bloque et attend. Une fois que le joueur a fait *new*, la file de *threads* est à nouveau remplie, et ces derniers peuvent être relancés. Si jamais l'utilisateur fait *new* avant la fin du jeu, le *thread* courant est arrêté, pour pouvoir repartir sur un jeu "neuf".

2.4.4 Les rebonds

Une mauvaise impression sur les rebonds peut rebuter quelque peu l'utilisateur. Nous ne pensions pas que cette partie serait la plus compliquée, mais c'en était pas loin. Nous effectuons les rebonds de façon séparées sur les briques cassables et incassables, car ces dernière sont considérées comme *blockers*. Lorsque la balle est bloquée par l'une d'elle, nous proposons au *driver* de la balle des possibilités de stratégies à essayer afin de choisir celle qui est applicable, pour que la balle puisse continuer à bouger.
Par contre, lorsqu'elle entre en collision avec les autres, c'est différent.

En ce qui concerne les murs, nous ne faisons qu'inverser la vitesse sur l'axe des ordonnées ou des abscisses suivant que le mur est horizontale ou non.

Calcul du point de collision Les murs et les briques incassables sont des *blockers*, c'est-à-dire que le système calcule avant de déplacer les entités mobiles (comme la balle) si ces dernières rentrent en collision avec les *blockers*. Et si c'est le cas, l'entité ne sera pas déplacée. Ceci pose un problème lorsque la balle est trop rapide. En effet, le système considérera qu'une entité ne pourra etre déplacée, alors qu'en raison de sa grande vitesse, elle ne sera pas collée à un mur ou une brique incassable. Pour rémédier au problème, il suffit de vérifier si la balle est effectivement collée ou pas, et ensuite calculer le point de collision si ce n'est pas le cas. Comme nous ne modifions la vitesse que sur l'axe des abscisses, ce calcul n'est pas effectué sur celui des ordonnées.

2.4.5 Gestion des briques

Les briques sont toutes issues de la meme classe abstraite. Cette classe gère quasiment tout, les briques ne font que définir des types qui seront utile pour identifier les entités qui se touchent. Principalement, elle précisent leur image qui sera affichée pour les identifier.

Pour alléger la mémoire, bien que le jeu soit petit, il ne nous a pas paru logique que chaque briques aient son image, alors que beaucoup de briques ont les memes. Pour gagner de la place, nous avons utilisé un tableau statique dans la classe abstraite qui répertorie les images et qui ne rajoutent que celles qui ne sont pas encore chargée. Les briques se voient alors assigner un numéro d'image, qui correspond à l'indice dans le tableau.

2.4.6 Bonus

Les bonus n'ont pas été la partie la plus dur à implémenter et penser, bien qu'elle demanda un peu d'imagination. Le plus dur a finalement été de comprendre bien le *framework* pour savoir comment arriver à nos fins.

2.5 Limites du jeu

Meme les grands jeux ont des limites, prenez par exemple *Assassin's Creed Unity*... quoique dans ce cas, il s'agisse plutôt de bugs.

La gestion des collisions n'a pas été chose simple, aussi il est assez remarquable par moment que la balle ne rebondit pas comme souhaité. Cela vient principalement *framework* qui permet difficilement d'avoir conscience du contexte de la balle. De plus, ce dernier permet que la balle touche plusieurs briques à la fois, ce qui peut entraîner des comportements inattendus, vu que la trajectoire de la balles sera modifiée plusieurs fois.

Pour des raisons qui nous échappe également, l'image des explosions ne disparaît pas certaines fois. Il est possible que ce soit dû à des problèmes de collisions multiples d'une brique avec une bombe et une autre élément du jeu (les balles tirées par exemple).

2.6 Difficultés rencontrées

Nous avons tout de même rencontrées plusieurs difficultés et nous en ferons la liste dans cette partie. Nous donnerons également "l'état" de cette difficulté dans la version actuelle du jeu.

3 Critiques du framework

Passons maintenant à une petite critique du *framework*... il est toujours plus facile de critiquer que de faire, donc comme nous avons fait, place à la critique (bien méritée, vous en conviendrez).

Nom des classes Mention spéciale aux noms des classes fort longs parfois. Bien que cela explicite peut-etre le rôle de chaque classe, il est parfois bien compliqué de s'y retrouver, et de comprendre le fonctionnement, car les noms ne différent pas de beaucoup par moment. De plus, les suffixes *DefaultImpl* alourdissent les noms et sont franchement inutile.

Introspection On nous a appris que l'introspection faisait perdre en performances. Et pourtant, le *framework* s'en sert pour faire appel aux fonctions de gestion des collisions. Ceci permet d'ajouter que les règles de collision qui nous intéressent, mais n'oublions pas que les performances en patissent. De plus, cette technique est utilisée à deux endroit du code, et cela est appliqué à chaque itération du jeu.

Image de fond Pour créer un niveau, il est nécessaire d'avoir un *viewport*, ce que le système fournit. Ce qui est par contre assez difficile à comprendre, c'est pourquoi la classe gérant *viewport* par défaut comporte une méthode qui permet de modifier l'image de fond, alors que finalement la méthode récupérant cette image ne fait que renvoyer une chemin vers une image ? Et cela écrit en dur en plus ! Conclusion, une classe fille seulement pour une méthode qui fait un travail inattendu...

Gestion des niveaux Pourquoi avoir gérer les niveaux comme des threads ? Bonne question... une fois lancés, impossible de les relancer, impossible à cloner, gestion plus compliquée, etc... nous supposons que cela est pour permettre à l'*EventDispatcher* de faire son travail avec l'interface du jeu, mais le travail pour gérer les niveaux est bien plus compliquée.
D'ailleurs, l'exemple donné (le *Pac Man*) ne gère pas du tout bien cela. Il suffit de rajouter un niveau au jeu pour se rendre compte que faire *new* lance deux niveaux en meme temps. Mais cela vient d'une boucle *foreach* sur les *threads* et donc la liste de niveau de retrouve parcouru à deux endroits différents. De plus, le *Pac Man* ne gère pas le *new* lorsque le jeu est terminé. Et ceci est normal, car un *thread* ne peut etre relancer. Pour pallier au problème, nous ne donnons que les chemins vers les fichiers de définition des niveaux et nous recréons des *threads* à chaque *new*. Si les *threads* étaient clonable, cela aurait pu etre éviter...

La facilité a un coût Le *framework* permet une mise en place rapide de jeux, et fournit les éléments nécessaires pour que cela soit jouable. Mais cette facilité est compensée par le fait qu'il n'est pas évident (à moins de réécrire certaines classes) de faire des choses très poussée. En effet, comme évoquée dans la partie implétementation, le système considère les collisions avec les *blockers* de façon très basique en fait. Il n'y pas de vérifications qui permettrait d'approcher les *blockers* de façon plus naturelle. C'est d'ailleurs pour cela que nous avons dû calculer nous-meme le point d'impact quand cela était nécessaire.

Une classe de collision à rallonge Loin de l'idée que les classes doivent etre courte pour etre plus facilement réutilisable, il y a peut-etre des limites. Autant pour la classe gérant la fenetre d'affichage, cela parait logique car il y a beaucoup de code pour mettre en place les éléments, autant des fois c'est évitable. Nous exposons ici la classe qui gére les collisions entre entités *overlappable*. La côté pratique, c'est que tout est au meme endroit, celui qui l'est moins, c'est qu'avec quelques centaines de lignes de codes et des fonctions qui se nomment toutes *overlapRule*, il n'est pas si évident de s'y retrouver.

4 Conclusion

En guise de conclusion, nous dirons que le projet a été tout de meme fort agréable, bien que la compréhension du *framework* ne fut pas triviale, et a été notre principal frein.
Ceci dit, faire un jeu a vraiment été agréable, de par de son côté evidemment ludique... et ça change de faire des sites web.