# PROG2 : Projet PONG

Tom Bachard Guillaume Barbier Victor Careil

24 avril 2018

**Classification ACM** D.m

**Mots-clés** Programmation ; Java ; JavaFX ; Programmation événementielle ; PONG

**Résumé**

Le paradigme de programmation événementielle est très souvent utilisé lorsqu'un utilisateur interagit de manière intermittente avec l'application. Grâce à la bibliothèque JavaFX de Java (anciennement Swing), nous mettons au point un moteur de jeu permettant de jouer au très célèbre jeu PONG.

## 1 Introduction

### 1.1 PONG

PONG est l'un des premiers jeux vidéos commercialisés. Inventé en 1972 par Nolan Bushnell et Allan Alcorn, ce jeu propose de faire s'affronter tantôt deux joueurs, tantôt un joueur et un ordinateur, dans un match de tennis de table vu du dessus. C'est le premier jeu qui devient populaire, en attestent les 8000 bornes d'arcade vendues par *Atari* l'année de son invention.

Lors d'un affrontement joueur contre joueur, le jeu doit être capable de gérer les déplacements des deux raquettes en même temps, tout en calculant la trajectoire de la balle, afin que cette dernière soit cohérente et permette une bonne expérience de jeu. Pour cela, nous développons notre PONG en Java, en nous servant de la bibliothèque JavaFX (anciennement Swing), spécialisée dans le traitement d'événements interactifs concurrents.

### 1.2 Programmation événementielle

La programmation événementielle est un paradigme de programmation qui, par opposition à la programmation séquentielle, s'appuie sur les événements et leur réactions qu'elle reçoit. Ce qu'on définit comme « événement » est très large, en effet, sont considérés comme tels les actions suivantes :

— l'ajout d'élément dans une liste ;
— le changement d'état d'un booléen ;
— le clic d'une souris ;
— la pression d'une touche de clavier…

On remarque que ces points, et notamment les deux derniers, sont très intéressants dans la réalisation d'un jeu tel que PONG. En effet, considérer les positions des différentes raquettes, ainsi que la balle, comme des événements permet une implantation simple et élégante de ce jeu.

Enfin, il est à noté que la programmation événementielle est généralement utilisée avec des langages de haut niveau, ce qui justifie notre utilisation de Java.

# 2 Concepts généraux

## 2.1 Les *timelines*

La principale source d'animation dans notre jeu sera gérée à travers les *timelines*. En effet, cette bibliothèque permet d'animer les représentation graphiques de nos objets. Il suffit, par exemple pour la balle, de calculer son prochain point d'intersection, le temps où elle réalisera la collision, et il n'y a plus qu'à faire avancer la balle le long du segment entre son point actuel et son point de collision.

## 2.2 Utilisation des *bindings*

Afin de répondre au mieux au problème posé, nous avons décidé d'utiliser les *bidings* proposé par JavaFX. En effet, ces derniers permettent de lier nos objets, et surtout leur propriétés, aux événements extérieurs, notamment liés aux joueurs. Sans entrer directement dans les détails, on voit bien que l'on pourra, entre autres, lier la position des raquettes en fonction des touches pressées facilement grâce à ce système.

## 2.3 Séparation du rendu et des calculs

Dans l'optique de rendre notre code solide et modulable, nous appliquerons le principe de « séparation des deux mondes ». En effet, le monde des calculs mathématiques, trajectoires, rebonds, ..., et celui du rendu des figures à l'écran, balle, raquettes, ..., ne doivent pas se croiser. Les classes graphiques ne doivent rien calculer, cela permet de changer radicalement la manière de calcul sans avoir à toucher à ces classes. On pourra même attester de cette robustesse dans notre code puisqu'après la seconde démonstration, nos collisions ont été entièrement refaites, sans avoir à toucher à quoi que se soit dans nos classes graphiques.

# 3 Implémentation de notre PONG

Lors de la réalisation du jeu, nous avons séparé les classes en deux catégories, comme expliqué précédemment. D'une part, nos classes de calculs, vivants dans un monde purement mathématique, et d'autre part nos classes graphiques, qui appellent les fonctions des premiers classes afin de dessiner les objets aux bons endroits.

## 3.1 Classes du moteur de jeu

Nous avons choisi de créer un petit moteur de jeu lors de ce projet, principalement de rendre les collisions des balles avec les murs ou raquettes. On pourra noter dans le sous groupe de fichiers *core* des énumérations et des interfaces, tant pour la lisibilité du code que pour sa modularité.

### 3.1.1 Classe *Vector2D*

Ce classe contient une classe abstraite de vecteur en deux dimensions, utiles pour tous nos calculs. On y retrouve les fonctions classiques d'addition, multiplication, de norme, mais aussi des plus spécifiques à notre jeu telles que des rotations ou de calcul de normal.

#### 3.1.2 Classe *Engine*

Le fichier *Engine* est la pièce principale du moteur. En effet, c'est lui qui contient l'entièreté des constantes liées au jeu. De plus, des méthodes classiques aux moteurs de jeu nous permettent de gérer facilement le lancement du jeu, son arrêt, les conditions de victoires, *etc*, dans les autres fichiers.

C'est aussi ce fichier qui gère la scène qui contiendra les objets que l'on générera et déplacera lors du jeu. On notera que les différents *bindings* sont créés à ce moment du processus.

Enfin, c'est au travers de ce classe que sont créés et mis à jour les affichages des scores et autres textes, tels que celui de victoire, durant la partie.

#### 3.1.3 Classe *World*

Cette classe permet de créer la zone de jeu, dont les bordures de la fenêtre, et d'y ajouter les collisions avec les différents éléments présents dans la zone de jeu. On remarquera que les bordures extérieures ne sont pas les seuls objets capables de générer des collisions.

#### 3.1.4 Classe *Borders*

L'unique but de ce classe est de calculer les prochaines intersections de la balle avec les bordures. Ce faisant, on peut directement calculer les *timelines* pour la balle. À chaque nouvelle intersection, on recommence le calcul.

Les collisions sont gérées sur les vitesses selon l'axe des abscisses ou des ordonnées, selon le type de collision.

#### 3.1.5 Classe *LevelManager*

Notre implémentation de PONG propose, une fois un certain score atteint, de passer d'un niveau à un autre. C'est au travers de cette classe qu'est réalisée la transition des niveaux. Une interface simple en liste est proposée.

### 3.2 Classe de rendu graphique

Afin de respecter le paradigme de séparation moteur/graphique, nous avons implanté pour chaque objet à rendre une classe qui ne s'occupe que d'afficher la figure géométrique concernée. Chacun de nos *renderer* est en réalité une extension simple d'une classe Java déjà existante, permettant des modifications simples :

— *SimpleRacketRender*, qui affiche les raquettes, étend la classe *Rectangle* ;
— *SegmentRender*, qui affiche les segments, étend la classe *Line* ;
— *BallRender*, qui affiche la balle, étend la classe *Circle*.

Dans le code, on ne voit que de rares appels à des fonctions du moteur de jeu, la plupart étant des *bindings*.