PROG2 : Projet PONG

Tom Bachard Guillaume Barbier Victor Careil

25 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. Un exemple de partie est donné en figure 1.

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 qu'elle reçoit, et leur réactions. Ce qu'on définit comme « événement » est très large, en effet, sont considérées 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.

De plus, cette notion d'évenement change la façon dont sont gérés les acteurs. Si l'affichage a besoin de la position de la balle, ce n'est pas à la balle de donner sa position à l'affichage, mais c'est plutôt à l'affichage de réagir à l'évènement « la balle a bougé ». Ceci change la logique et l'organisation du code pour le mieux.

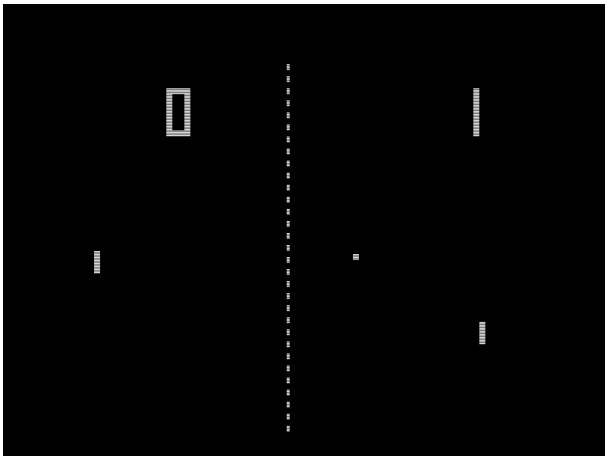

FIGURE 1 – Une partie de Pong.

Enfin, il est à noter 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 par des *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 selon son vecteur vitesse jusqu'à ce qu'elle arrive au point d'intersection calculé. En prévision du prochain déplacement, on calcule aussi la prochaine direction de déplacement de la balle après le rebond.

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 de la balle à son affichage.

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, des trajectoires, ou encore des rebonds, et celui du rendu des figures à l'écran, de la balle ou encore des 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.

Nous avons également deux systèmes de coordonnées, un système local où sont effectué les calculs et la logique interne du jeu. Ce système de coordonnées peut ensuite être converti dans un système

de coordonnées graphique qui lui est uniquement dédié à l'affichage. Ainsi, la logique interne du jeu PONG n'a pas à dépendre de la bibliothèque JavaFX qui gère l'affichage.

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 package *core* des énumérations et des interfaces, tant pour la lisibilité du code que pour sa modularité.

Classe *Vector2D*

Cette 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, de calcul d'un vecteur normal, ou d'ajout d'une petite perturbation angulaire aléatoire, utile pour rendre chaque partie unique.

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*. C'est aussi cette classe qui gère la scène qui contiendra les objets que l'on générera et déplacera lors du jeu.

De plus nous considérons que certaines valeurs, telles que la position des raquettes, font partie intégrante du jeu. Plutôt que de les reléguer à des classes qui gèrent les raquettes, ces valeurs sont des « Property »qui appartiennent à la classe *Engine*. Ce sera ensuite aux classes filles de *Racket* de modifier ces valeurs selont les actions des joueurs.

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, sous la forme d'instances de classes sous l'interface *StaticCollision*. Pour effectuer le calcul des collisions, il suffit d'indiquer à la classe « World »la position, la vitesse et le rayon de la balle. Cette classe se charge ensuite d'effectuer tout les calcules nécessaires de manière transparente. La balle n'a ainsi aucune connaissance des objets dans le jeux, mais peut tout de même les prendre en compte dans ses déplacements.

Interface *StaticCollision*

L'unique but de cette interface est de calculer les prochaines intersections de la balle avec un certain objet, grâce à quoi la balle peut calculer les *timelines* nécessaires à son déplacement. Les objets qui sont gérés pour l'instant sont les bordures du jeu, avec la classe *Borders*, et les segments, avec la classe

Segment. Une méthode statique de *LevelGenerator* permet de créer des rectangles, mais elle crée en réalité 4 segments.

Interface *Level*

Cette interface permet de décrire ce qu'est un niveau : c'est une classe qui crée une instance de *Worl* et la rempli d'instances de *StaticCollision*. Elle décrit ensuite comment afficher ces objets, ce qui sera passé à l'*Engine* par le *LevelManager*.

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, avec la fonction *nextLevel*. La structure de sauvegarde des niveaux est invisible pour le reste du programme. En effet, la classe *Engine* n'a pas connaissance de la classe *LevelManager*, car le moteur de jeu doit seulement se soucier de la bonne exécution des parties, pas du niveau dans lequel ces parties sont exécutées.

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*. Cela confirme la séparation des deux univers.

3.3 Dialogue entre agents

Compte tenu de ces nombreuses classes et des *bindings*, de nombreux agents communiquent au cours de l'exécution du programme. On donne en figure 2 l'ensemble de ces communications, qui sont perçues au travers de nos différentes classes explicitées précédemment.

4 Extensions

Notre implémentation du jeu PONG propose comme extension la possibilité de jouer à plusieurs niveaux différents. En effet, lorsque l'un des joueurs atteint le score de 5, le niveau actuel change, et les scores sont remis à zéro. La classe *LevelManager* s'occupe de la gestion des niveaux.

De plus, s'ajoute à cette fonctionnalité la possibilité de créer des niveaux nous-même. Un créateur de niveaux, au travers de la classe *LevelGenerator*, propose une interface simple, mais « codée en dur », de créer des classes de niveaux. Nous nous sommes limités à deux niveaux, *Level0* et *Level1*, ce dernier étant donnée en figure 3. Actuellement, il est possible de créer des segments, des rectangles et des losanges. D'autres formes géométriques sont envisageable, il suffit de les implémenter dans *LevelGenerator* pour améliorer l'interface de création.

On remarque aussi que, bien que non réalisée, s'esquisse une implémentation d'un créateur de niveau avec une interface utilisateur.

FIGURE 2 – Diagramme de classe.

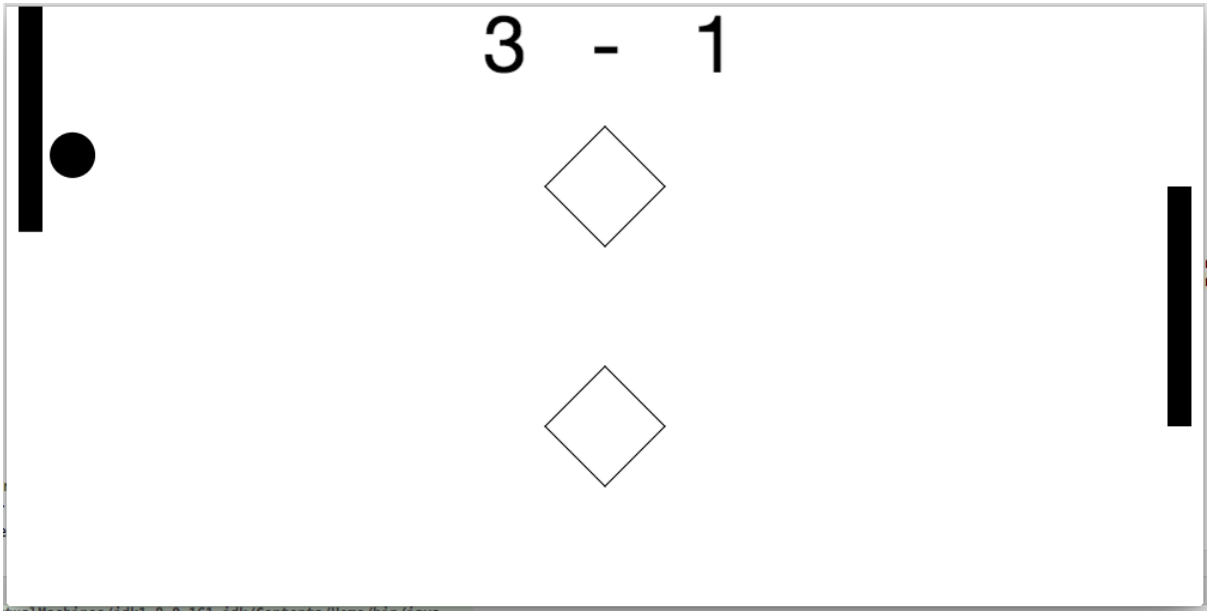

FIGURE 3 – Exemple de niveau PONG.

Auto-évalutation *SWOT*

Forces Un code modulaire est robuste, prenant bien compte de l'utilisation des *bindings*, des *timelines* et respectant le paradigme de programmation événementielle ;

Faiblesses Mauvaise répartition du code due à une mauvaise organisation ;

Opportunités Découverte et compréhension de Java, rajout de fonctionnalités aisé grâce à la forme du code, possibilité de porter le moteur à d'autres jeux ;

Menaces Prévoir un temps d'organisation est nécessaire en début de projet, il ne faut pas négliger ce moment, même pour un projet qui semble aussi simple.