Projet Programmation Fonctionnelle
Arkanoid

BOUCHAMA Ayoub, ELGUERRAOUI Oussama, DIOURI Ayoub

Contents

List of Figures

1 Introduction

Ce projet vise à recréer une version du jeu Arkanoid. Dans ce rapport, on va présenter en détail le processus de reproduction de ce jeu en OCaml. Voici les principes fondamentaux du jeu. Le principe du jeu consiste à donner au joueur le contrôle d'une raquette horizontale en déplacement de droite à gauche. La balle doit etre réflichi sur la palette et aller briser les briques au dessus pour accumuler des points. Chaque fois que la balle passe en dessous de la raquette et sort du bas de l'écran, le joueur perd

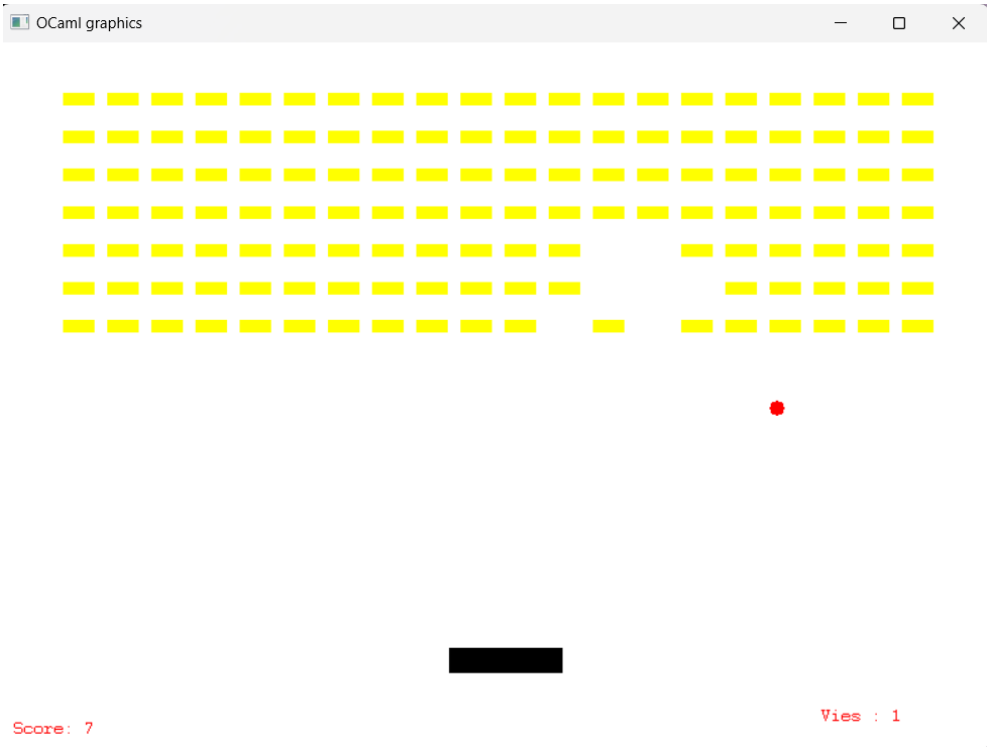

Figure 1: Interface graphique du jeu

Pour implémenter ce jeu, nous avons utilisé des modules et des flux, ainsi que la librairie Graphics. Chaque élément du jeu (balle, paddle, brick) a été défini dans un module distinct dans un fichier distinct, de même qu'un autre module pour le jeu lui-même (game). Pour les collisions entre les différents éléments, nous avons adapter ce qu'on a fait dans le TP7 à nos besoins.

2 Les modules

Dans tous les modules implémentés, on a travaillé avec le type (`t : float`) seulement pour définir les des coordonnées d'une position et les dimensions des solides, et un entier pour la puissance de chaque brique.

2.1 Module des briques

Nous avons modéliser une brique comme étant une combinaison d'une **position**, d'une **taille**, d'une **puissance** et d'une **couleur**.

```
(t * t) -> (t * t) -> int -> color -> brick
```

Figure 2: Le type d'une brique

Nous avons ainsi implémenté des méthodes pour pouvoir manipuler les briques, principalement la génération des lignes de briques, remetrre la puissance de certains briques à 0, le nombre de brique sans puissance pour calculer le score.

En effet, au lieu de mettre un nouveau type option qui permet de modéliser les états d'une brique, nous avons décidé de se contenter de la puissance et de la couleur pour gérer ceci :
Une brique détruite se transforme en une brique de puissance nulle et de couleur blanche pour disparaitre avec la couleur de l'arrière plan du jeu.

Le nombre de briques d'une ligne est déterminé à aprtir de la largeur de la fenetre du jeu, de la largeur des briques, et de l'espace entre chaques briques 2 à 2.

Une méthode pour dessiner chaque brique sous forme d'un rectangle à partir de sa longueur, sa largeur et sa couleur a été implémentté également à l'aide du module Graphics.

2.2 Module de la palette

La palette ressemble parfaitement à une brique mais sans puissance, donc le type d'une palette est une combinaison d'une **position**, d'une **taille**, et d'une **couleur**.

```
(t * t) -> (t * t) -> color -> paddle
```

Figure 3: Le type d'une palette

La méthode principale de ce module est celle permettant le mouvement de la palette en accord avec le mouvement de la souris.

```
(* Mise à jour de la position du paddle *)
(t * 'a) * (t * 'b) * 'c -> t * bool -> (t * 'a) * (t * 'b) * 'c
let updatePadle ((x, y),(width, height), color) (xs,buttonDown)  =
  let new_x =
    if buttonDown then
      max 0. (min (xs -. (width /. 2.)) (float_of_int (size_x ()) -. width))
    else x
  in
  ((new_x, y),(width, height), color)
```

Figure 4: Méthode pour le mouvement de la palette

cette methode prend en paramètre une ancienne palette, l'abscisse de la souris de l'utilisateur et un boolean qui indique que la palette est en mouvemment.
On calcule ensuite l'abscisse du centre de la palette tout en vérifiant qu'il ne sort pas de l'ecran du jeu, puis on initialise la nouvelle palette.

2.3 Module de la balle

Nous avons implémenter le module de la balle en la représentant comme combinaison d'une **position**, d'un **rayon**, d'une **couleur**, d'un **etat** (Normal ou perdu) et d'une **vitesse** sous la forme de deux composante (`Vx` et `Vy`).

```
(t * t) -> t -> color -> state -> (t * t) -> ball
```

Figure 5: Le type de la balle

Le module balle possède plusieurs méthodes necessaires pour la manipulation de l'objet balle, notamment les getters et setters des attributs de la balle et les méthodes pour la gestion des collisions et la mise à jour de la balle.

Pour les collisions, on a implémenter des méthodes pour gérer les trois types de collision existante :

2.3.1 Collision entre la balle et la palette

```
(t * t) * 'a * 'b * 'c * (t * t) -> t * t -> t * t -> (t * t) * 'a * 'b * 'c * (t * t)
let reflectBallPaddl (position, radius, color, state, speed) (xp,yp) (width, height) =
  let (x,y) = position in
  let (vx,vy) = speed in
  let nx =   if (x >= float_of_int (size_x ())) || x <= 0.  then
  -.vx
else
  vx in
  let ny = if (y <= (yp +. height) && x <= (xp +. width) && x >= xp)  || y >= float_of_int (size_y () ) then -. vy else vy in
  let newSpeed = (nx,ny) in
  (position, radius, color, state, newSpeed)
```

Figure 6: Gestion de collision entre la palette et la balle

Cette methode permet de modifier les composantes vitesses de la balle pour changer la direction du déplacement de la balle puisqu'on sait que :

$$\alpha = arctan(\frac{V_y}{V_x})$$

Lorsque la balle touche la palette, on inverse le signe de la composante vitesse verticale tout en gérant les dépassements des coordonnées de la balle ceux de l'ecran pour éviter les bugs et les erreurs.

2.3.2 Collision entre la balle et les briques

```
(t * t) * 'a * 'b * 'c * ('d * t) -> Brick.Brick.brick list -> (t * t) * 'a * 'b * 'c * ('d * t)
let rec reflectBallBrick (position, radius, color, state, speed) brickLine =
  let (x,y) = position in
  let (vx,vy) = speed in
  match brickLine with
  | [] -> (position, radius, color, state, speed)
  | t :: q -> if (Brick.is_position_inside_brick (x,y) t) && (Brick.brickWithPower t) then
      let newSpeed = (vx, -.vy) in
      (position, radius, color, state, newSpeed)
    else
      reflectBallBrick (position, radius, color, state, speed) q
```

Figure 7: Gestion de collision entre une brick et la balle

Cette methode permet notamment de gérer la collision entre la blle et une brique de la meme manière que la collision de la balle avec la palette en inversant le signe de la composante vitesse verticale.
On a également vérifier que la brique en collision a une puissance puisqu'on a considérer une brique sans puissance tel qu'elle n'existe pas, donc il n'y aurait pas de collision entre la balle et celle ci.
La methode qui permet de vérifier si la balle a toucher une brique ou pas est la suivant :

```
t * t -> (t * t) * (t * t) * 'a * 'b -> bool
let is_position_inside_brick (xb, yb) ((brick_x, brick_y), (brick_width, brick_height), _, _) =
  xb >= brick_x && xb <= brick_x +. brick_width &&
  yb <= brick_y && yb >= brick_y -. brick_height
```

Figure 8: Vérifier collision entre la balle et une brique

Cette methode se contente de vérifier si les coordonnées d'une brique avec celles de la balles coincident. Si c'est le cas, il y a forcement une collision.

2.3.3 Collision entre la balle et les extrimités de l'ecran

On devait également forcer la balle à ne pas dépasser les limites de la fenetre du jeu et d'avoir des reflexions lors de du contact entre la balle et les deux extrimités gauche et droite et celle d'en haut.

```
(t * t) * 'a * 'b * 'c * (t * t) -> (t * t) * 'a * 'b * 'c * (t * t)
let reflectSide (position, radius, color, state, speed) =
  let (x,y) = position in
  let (vx,vy) = speed in
  let nx =   if (x >= float_of_int (size_x ())) || x <= 0.  then
      -.vx
    else
      vx in                                                    t
  let ny = if y >= float_of_int (size_y () ) then -. vy else vy in
  let newSpeed = (nx,ny) in
  (position, radius, color, state, newSpeed)
```

Figure 9: Vérifier collision entre la balle et les extrimités de la fenetre

Pour l'extrimité inférieure de l'ecran, l'utilisateur doit perdre la partie courante donc soit perdre une vie ou terminer le jeu.

```
('a * t) * 'b * 'c * 'd * 'e -> t -> t -> bool
let isFinished ((_,y), _, _, _, _) yp height =
  if (y <= yp -. height -. 10.) then true else false
```

Figure 10: Vérifier collision entre la balle et l'extrimité inférieure de l'ecran

Cette methode nous permet de vérifier si la balle a atteint le plus bas de la fenetre en prenant en considération la marge.

2.4 Type Jeu

On a représenté ce type pour modéliser l'etat d'une partie qui doit contenir une palette, des briques, une balle, un score et des vies.

```
(* Définition du type game *)
type game_state = {
  bricks : Brick.brick list list;
  paddle : Paddle.paddle;
  ball : Ball.ball;
  score : int;
  vie : int;
}
```

Figure 11: Type d'un jeu

Celui ci va nous permettre de créer plusieurs instance d'un jeu notamment pour rafraichir les données de la partie à une frequence de 60Hz.
La séparation de ce type du fichier principale de l'execution nous a offert plus de lisibilité puisqu'on pourrait ajouté d'autres choses à l'état du jeu sans devoir modifier toute un fichier.

3 Lancer le jeu

Pour lancer le jeu, il suffit de se placer dans le dossier principale et lancer la commande suivant :

$$dune\ exec\ \_build/default/bin/newtonoid.exe$$

4 Conclusion

La réalisation de ce projet de reproduction du jeu Arkanoid en OCaml a été une expérience stimulante. En mettant en pratique les principes de la programmation fonctionnelle, nous avons développé une version du jeu axée sur la modularité et la clarté du code. Ce projet nous a permis d'approfondir nos compétences techniques, de renforcer notre compréhension de la programmation fonctionnelle et de travailler efficacement en équipe.