Rapport

Projet Programmation Fonctionnelle

NEWTONOID

Membres

TENE FOGANG ZACHARIE IGOR
OUDARD VERON
MEKKAOUI OSSAMA MOUSSA

Département Sciences du Numérique - Deuxième année, parcours Logiciel

Table des matières

Introduction

Dans ce jeu, une balle soumise par la gravité doit rebondir sur une barre mobile contrôlée par le joueur. La barre, limitée à des déplacements horizontaux, permet de diriger la trajectoire de la balle. L'objectif est de viser des briques, qui disparaissent lorsqu'elles sont touchées par la balle. Chaque brique détruite contribue à augmenter le score du joueur. En revanche, si la balle n'est pas interceptée par la barre, elle chute, pouvant entraîner la fin de la partie.

I. Organisation du code

Le code est organisé sous forme de différents fichiers .ml et .mli qui remplissent chacun un rôle dans le fonctionnement du jeu. De plus, ils sont séparés dans deux dossiers, un dossier lib contient les fichiers qui sont utilisés par le fichier dans le dossier bin qui lance et affiche le jeu. Nous avons majoritairement travaillé dans le répertoire lib.

Notre implémentation est structurée en plusieurs modules interdépendants :

- ***game.ml*** : Gère la logique principale du jeu, les états, les interactions et les boucles de jeu. Il orchestre les différents composants en coordonnant les mouvements des balles, les collisions, les interactions avec la raquette et la gestion des power-ups(bonus). Ainsi, il met à jour l'état du jeu à chaque instant. C'est la logique principale du jeu.

- ***brick.ml*** : Définit les différents types de briques et leur comportement. Il est responsable de la gestion détaillée des briques, définissant leurs types, leurs caractéristiques comme les points de vie, les valeurs de score et les bonus potentiels. Il implémente les mécanismes de collision et de destruction des briques, offrant une abstraction complète de ces éléments de gameplay.

- **paddle.ml** : Gère la raquette et ses interactions.

- ***bonus.ml*** : Gère le système de bonus, définissant différents types de power-ups comme l'agrandissement de la raquette, l'accélération des balles ou la multiplication des balles. Il implémente des mécanismes pour créer, déplacer et gérer les interactions des bonus avec la raquette et l'environnement de jeu.

- ***gamestate.ml*** :Module permettant de définir la structure de l'état du jeu (en particulier de l'état initial).

- ***iterator.ml*** : Fichier qui nous a été fourni implémentant des flux.

- ***quadtree.ml*** : Module implémentant un quadtree permettant la gestion efficace de l'espace de jeu et de la collision.Il permet de repérer les briques proches de la position de la balle et de limiter la recherche de la collision de la balle à ses briques.

- ***type.ml*** et ***input.ml*** : Modules qui fournissent des définitions de types fondamentaux et la gestion des entrées utilisateur.
- **ball.ml** : Module permettant de gérer la logique de mouvement et collision de la balle.
- **config.ml** :Modules des paramètres de configuration du jeu.
- ***newtonoid.ml*** **:** Fichier principal gérant l'affichage du jeu et utilisé pour l'exécution du jeu (notamment le graphique.)

Cette modularisation permet une séparation claire des responsabilités et facilite la maintenance. Nous précisons que dans ces fichiers .ml (des interfaces ont été définies dans certains d'entre eux). Le répertoire principal de travail étant le dossier lib, nous présentons dans la figure ci-dessous son contenu :

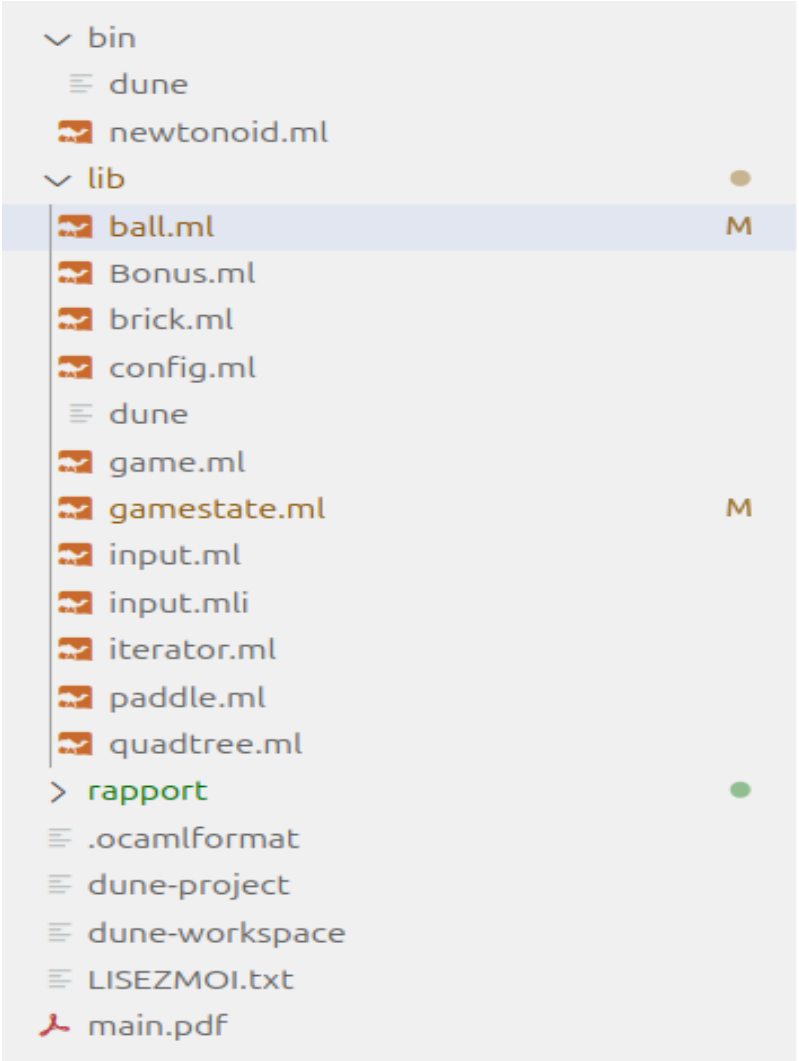

Figure 1 : Présentation de la structure du répertoire lib

II. Explication détaillées des parties clés et choix de conceptions

Présentation

Nous avons élaboré pour chaque éléments constitutif du jeu un module correspondant:

- brick.ml pour les briques
- paddle.ml pour la raquette
- bonus.ml pour les bonus
- ball.ml pour la ball

Dans newtonoid , nous avons implémentés les fonctions

1. draw_bricks pour tracer les briques
2. draw_state pour la gestion de l'état global
3. draw_effects pour tracer les bonus visuellement
4. draw_paddle pour tracer la requête
5. draw_ball pour tracer la balle

Nous précisions que l'état global du jeu est un flux comme il est défini dans create_game_state du module **game.ml.**

```
let create_brick_grid rows cols brick_width brick_height spacing level =
  let ((_, _), (_, y_max)) = Config.screen_bounds in
  let create_row y row_idx =
    let rec create_cols x col_idx acc =
      if col_idx >= cols then acc
      else
        let brick_type =
          let r = Random.float 1.0 in
          if r < 0.1 then
            Brick.Bonus (
              match Random.int 1 with
              (* | 0-> Brick.StretchPaddle
              | 1 -> Brick.ShrinkPaddle *)
              (* | 2 -> Brick.SpeedUpBall
              | 3 -> Brick.SlowDownBall *)
              | 4 -> Brick.ExtraLife
              | 5 -> Brick.RestoreLife
              (* | 6 -> Brick.MultiplyBall *)
              | _ -> Brick.ScoreBonus (Random.int 100)
            )
```

Figure 1 :illustration du type de brique qui apparaitra par défaut

Remarque: La modulation des couleurs des briques se gère dans newtonoid.ml et la configuration des types briques qui doivent apparaitre au démarrage du jeu se manipule dans gamestate.ml dans la fonction **create_brick_grid** .

II. 1 L'état global du jeu

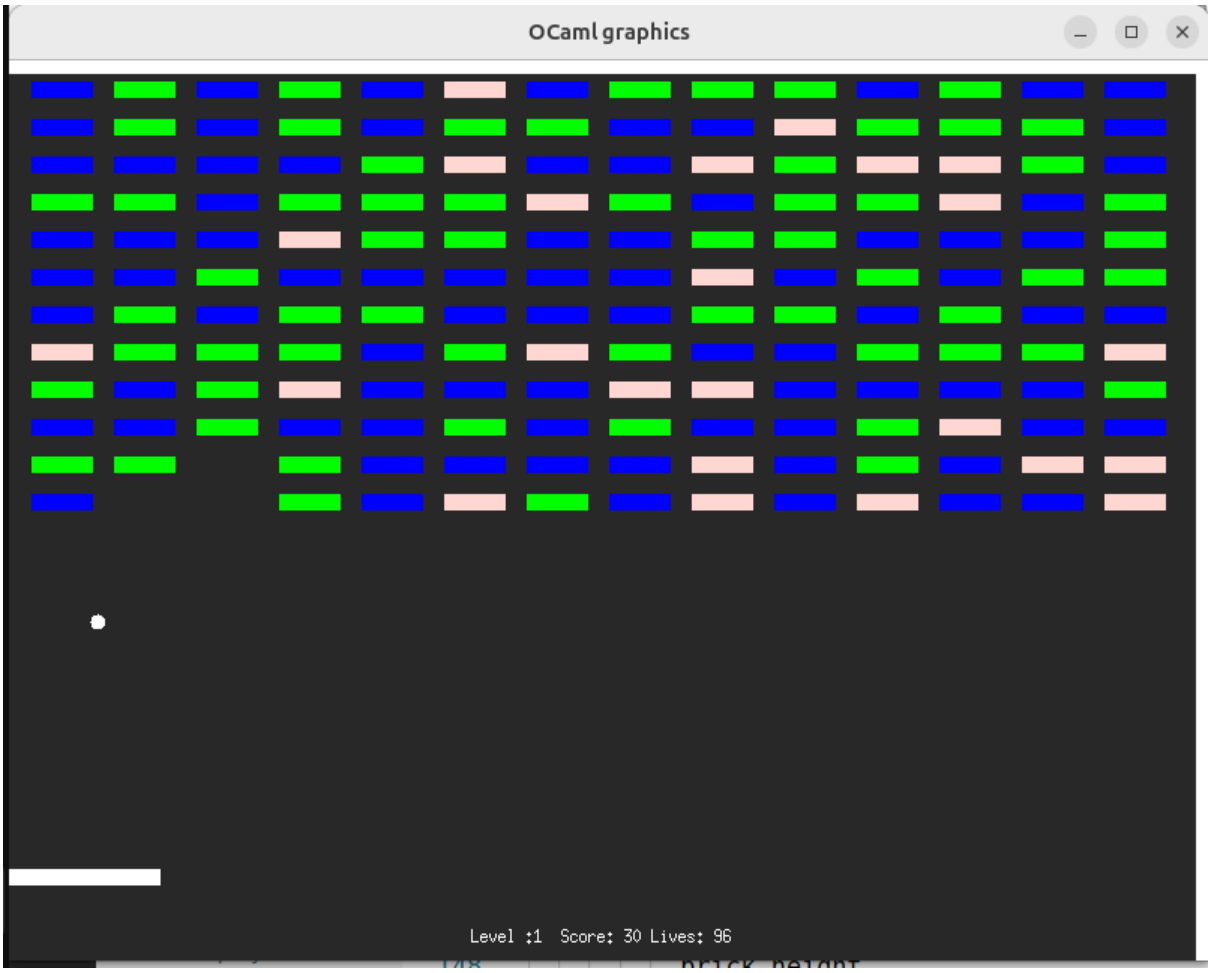

Figure 2 : illustration de l interface de jeu

1. Structure de l'état

L'état global est défini dans le module **Gamestate** avec le type suivant :

```
type t = {
  ball: Ball.t;               (* État de la balle *)
  paddle: Paddle.t;           (* État de la raquette *)
  bricks: Brick.t list;       (* Liste des briques restantes *)
  score: int;                 (* Score actuel *)
  lives: int;                 (* Vies restantes *)
  level: int;                 (* Niveau actuel *)
  status: game_status;        (* État du jeu (Playing, Paused, etc.) *)
  last_update: float;         (* Timestamp dernière mise à jour *)
  quadtree: Brick.t Quadtree.node; (* Structure spatiale pour collisions *)
  bonus_items: Bonus.t list;  (* Bonus actifs en jeu *)
  active_effects: (Brick.bonus_effect * float) list;  (* Effets sur la raquette *)
  active_ball_effects: (Brick.bonus_effect * float) list; (* Effets sur la balle *)
}
```

Figure 3 : représentation de l'état global du jeu

2. Configuration initiale

La configuration initiale est centralisée dans le module **Config** pour faciliter les ajustements :

```
module Config = struct
  let initial_lives = 100     (* Vies de départ *)
  let initial_score = 0       (* Score initial *)
  let initial_level = 1       (* Niveau de départ *)
  let paddle_position = (350., 50.)  (* Position initiale raquette *)
  let ball_radius = 5.        (* Rayon de la balle *)
  let brick_width = 40.       (* Largeur des briques *)
  let brick_height = 10.      (* Hauteur des briques *)
```

Figure 4 : Illustration du contenu module config

Cependant, l'ajustement de certains paramètres le nombre de lignes (rows) et colonnes(cols) peut ne pas fonctionner ,car nous avons implémenté dans Gamestate, un fonction **calculate_brick_layout** permettant de les déterminer automatiquement.

3. Gestion des états du jeu

Les différents états possible de jeu sont gérés via :

```
type game_status =
  | Playing        (* Jeu en cours *)
  | Paused         (* Jeu en pause *)
  | GameOver       (* Partie terminée *)
  | LevelComplete (* Niveau terminé *)
  | NextLevel      (* Transition vers niveau suivant *)
  | Starting       (* Initialisation *)
```

Figure 5 :états possibles de jeu

En effet, la structure aurait dû nous permettre de gérer différentes situations cependant, seul l'état playing a été pleinement utilisé.

4. **Mise à jour de l'état**

La mise à jour de l'état global est gérée par la fonction **update_state** dans le module *Game* qui :

- 1. Met à jour la position de la balle
- 2. Gère les collisions (balle-murs, balle-raquette, balle-briques)
- 3. Met à jour les bonus et leurs effets
- 4. Vérifie les conditions de fin de niveau/partie
- 5. Met à jour le score et les vies

```
let update_state dt mouse_state state =
  match state.bricks with
  | [] -> (* Niveau terminé -> création du niveau suivant *)
  | _ -> game_loop dt mouse_state state
```

Figure 6 Fonction de mise à jour de l'état

5. **Système bonus extensible**

Le système de bonus utilise un type somme polymorphe permettant d'ajouter facilement de nouveaux effets. Nous l'avons défini dans le module brick.ml car c'est la destruction d'une brique qui permet déclencher un effet de gratification.

```
type bonus_effect =
  | Normal
  | StretchPaddle
  | ShrinkPaddle
  | SpeedUpBall
  | SlowDownBall
  | ExtraLife
  | RestoreLife
  | MultiplyBall
  | ScoreBonus of int
```

Figure 7 : Présentation du type bonus effects (défini dans brick.ml)

Tout au long du jeu, les bonus sont gérés en 03 phases :

1. Création : Lors de la destruction d'une brique

```
let bonus_items =
  List.filter_map
    (fun brick ->
      match Brick.get_type brick with
      | Brick.Bonus effect -> Some (Bonus.create (Brick.get_position brick) effect)
      | _ -> None)
    destroyed_bricks
```

Figure 5 ; illustration création bonus

2. Application : Lors de la collision avec la raquette

```
let handle_active_effects dt state new_paddle collected_bonuses =
  let new_effects = List.map Bonus.get_effect collected_bonuses in
  (* Application des effets avec durée limitée *)
```

Figure 7 : illustration application

3. Gestion de la durée des effets

II.2 Système de Collisions avec quadtree

Le quadtree est une structure hiérarchique utilisée pour partitionner l'espace des briques. Cela permet de réduire le coût des vérifications de collisions, en n'évoluant que les briques proches de la balle.L'un des piliers de notre travail réside dans la gestion des

collisions. Nous avons implémenté une structure de *quadtree*, divisant intelligemment l'espace de jeu pour réduire la complexité computationnelle des détections de collision.

Au lieu d'une vérification naïve $O(n^2)$ qui testerait chaque objet contre tous les autres, notre quadtree permet de réduire cette complexité. L'espace de jeu est récursivement divisé en quatre quadrants, permettant des recherches de collision bien plus rapides et efficaces.

```
let handle_brick_collisions ball quadtree =
    let (candidate_bricks, others_brick) = Quadtree.query quadtree ball in
    let (colliding_bricks, candidate_bricks_without) =
      List.partition (fun brick -> Brick.check_collision brick
        (Ball.get_position ball) (Ball.get_radius ball))
        candidate_bricks in
    (colliding_bricks, candidate_bricks_without, others_brick)
```

Figure 8: Utilisation du quadtree

1. Collision avec les murs

La balle peut rebondir sur les bords de l'écran (murs latéraux et supérieurs). L'algorithme (**handle_wall_collision**) vérifie si la position de la balle dépasse les limites définies par les coordonnées de l'écran.

Les caractéristiques clés sont:

- Conservation de l'énergie : La vitesse totale est conservée en utilisant **current_speed**
- Angle de rebond aléatoire : Pour éviter les boucles infinies, un angle aléatoire est ajouté
- Gestion des bordures: Empêche la balle de sortir de l'écran en ajustant sa position

2. Collision avec la raquette

La raquette est contrôlée par le joueur et peut influencer la trajectoire de la balle. La vitesse de la ball apres collision est gérée dans le module ball par la fonction **handle_paddle_collision** . Lors d'une collision, l'angle de rebond dépend de la position relative du point d'impact sur la raquette.

Les caractéristiques clés sont les suivantes :

- Angle de rebond variable : L'angle dépend du point d'impact sur la raquette
- Accélération progressive : Léger boost de vitesse à chaque rebond
- Conservation de l'énergie: Maintien de la vitesse totale

- Prévention des blocages: Ajustement automatique de la position

3. Collision avec les briques

Lorsqu'une balle entre en contact avec une brique, celle-ci disparaît et la balle change de direction en fonction de l'angle d'impact.

Les caractéristiques clés sont:

- Calcul de l'angle de réflexion : Basé sur le point d'impact
- Conservation de l'énergie : Vitesse totale maintenue
- Prévention du blocage : Force la composante verticale vers le haut
- Détection : Utilisation de l'algorithme AABB

4. Physique de la Balle

Les propriétés fondamentales de la balle définis dans le module ball sont :

```
type t = {
    position: float * float;
    velocity: float * float;
    radius: float;
    mass: float;
}
```

Figure 9 : propriétés de la balle

La trajectoire de la balle résulte d'un calcul vectoriel précis, prenant en compte :

- La vélocité initiale
- L'angle de rebond
- L'influence de la gravité
- Les interactions avec les différents objets du jeu

Chaque rebond est calculé de manière déterministe, assurant un comportement physique réaliste et prévisible.

III . Points d'amélioration et difficultés rencontrées

1. Points d'amélioration

La gestion des rebonds de la balle intègre plusieurs aspects sophistiqués :

- Angles rebond variables sur la raquette selon le point d'impact
- Conservation de l'énergie cinétique
- Ajout d'aléatoire pour éviter les boucles infinies

En plus de cela, nous avons mis en place un système de bonification (les bonus qui permettent de restaurer le nombre de vie, d'ajouter le nombre de vie d'une vie , de ralentir la balle, d'accélérer la balle, d'agrandir et de diminuer la taille de la requête).

2. Difficultés rencontrées

- La gestion des effets temporaires (bonus) a nécessité une réflexion approfondie sur la persistance des états.
- L'implémentation du Quadtree a demandé plusieurs itérations pour gérer correctement tous les cas.
- Les rebonds de la balle ont nécessité des ajustements pour éviter les comportements cycliques , en particulier des déplacements horizontaux et/ou verticaux intermittent qui pouvaient se produire dans certaines situations.
- L'annulation des effets (bonus) sur la raquette et la balle.

3. Choix rétrospectif et limites

1. Points positifs

- L'utilisation du Quadtree qui améliore significativement les performances
- La séparation claire des responsabilités entre modules :
- L'utilisation de types algébriques pour les bonus et les briques

2. Limites

- Gestion plus efficace et sophistiqués des collisions multiples
- La gestion du rendu des briques non frappées par la balle laisse à désirer car peu efficace.
- La gestion des effets (notamment leur annulation) n'est pas complète.
- Le rendu de la balle et de la raquette
- Le mécanisme de gestion des états n'est pas très évident.

Conclusion

A travers l’utilisation des flux, ce projet nous a montré la puissance de la programmation fonctionnelle en OCaml. Contrairement à l'approche impérative, notre conception permet une manipulation plus intéressante des états du jeu, où chaque transformation génère un nouvel état plutôt que de modifier directement des variables.