Programmation fonctionnelle avancée

NEWTONOID

par

Thomas BOCANDÉ
Tom AUDARD
Margaux GARRIC
Youssef EL ALAMI

Département Science
du Numérique

TOULOUSE-INP ENSEEIHT

Date : 26 janvier 2024

Table des matières

Table des figures

1. INTRODUCTION

Durant ce projet, nous avons essayé d'implémenter le comportement du jeu Arkanoid mais avec une balle soumise à la pesanteur. Nous avons découpé le programme en différents fichiers afin de moduler le plus possible le jeu. Nous avons eu beaucoup de mal à rejoindre les bouts avec le fichier state.ml qui est censé faire la liaison entre tous en représentant l'état du jeu. Nous n'avons pas réussi à arriver à un jeu fonctionnel. Nous avons bloqué pendant longtemps sur la gestion des collisions. Dans ce rapport allons tout d'abord parler des différents objets implémentés. Ensuite nous parlerons de la gestion de l'espace de jeu et notamment de l'état du jeu. Enfin nous parlerons du programme principal.

2. LES OBJETS

Chacun des fichiers correspondant à un objet contient une fonction "create" qui permet de créer l'objet lors de l'initialisation par exemple et une fonction "draw" qui dessine l'objet lors de l'affichage.

2.1 ball.ml

Ce fichier contient ce qui correspond à la balle. Au début, nous étions partis sur une implémentation proche de celle du TP7. Par la suite, après l'implémentation du quadtree, nous avons modifié cette implémentation pour essayer de prendre en compte des collisions AABB.

2.2 brick.ml

Ce fichier contient ce qui représente les briques. Le type brick possède un objet et une couleur. On utilisera une liste de ce type brick pour représenter les briques.

2.3 paddle.ml

Ce fichier contient ce qui correspond à la raquette. Le type paddle contient un objet. En plus des deux fonctions de base ici on retrouve la fonction filter. Cette fonction filter va nous permettre de récupérer la position de la souris en fonction de si c'est cliqué ou non.

2.4 Pistes d'amélioration

- Intégration plus poussée de la détection des collisions AABB pour améliorer la gestion des interactions entre les objets.
- Ajoutez des animations ou des effets visuels lorsqu'une brique est touchée par la balle.
- Inclure des options de contrôle, telles que l'utilisation du clavier pour déplacer la raquette.
- Le jeu se termine lorsque l'utilisateur a utilisé un certain nombre de balles. Si l'utilisateur perd une balle et n'a pas encore utilisé toutes les balles, le jeu recommence. Une fois le jeu terminé, la fenêtre de jeu se ferme automatiquement.

3. GESTION DE L'ESPACE DE JEU ET DES COLLISIONS

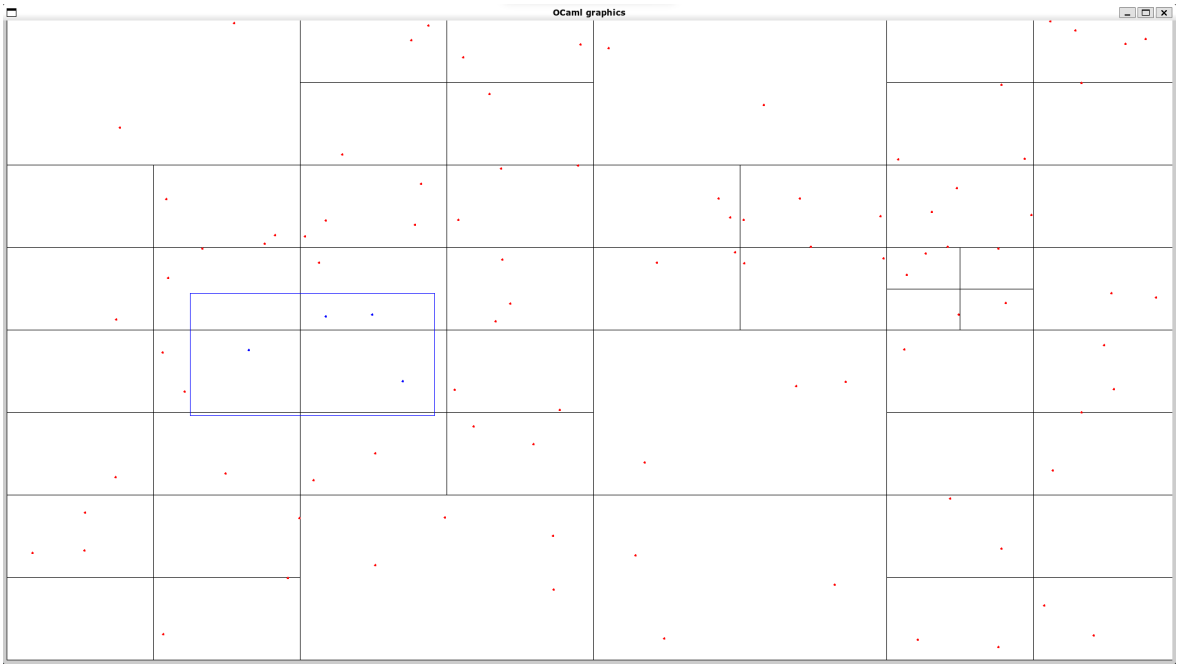

FIGURE 3.1 – Quadtree

3.1 quadtree.ml

Le fichier quadtree regroupe les fonctions permettant la manipulation et création de quadtree dans le jeu

3.1.1 Type aabb

le type `aabb` (*"Axis-Aligned Bounding Box"*) définit par une hauteur et une largeur et de son centre `(float*float)` (un `vector`) , elle nous permet de sevir de *"frontière"* comme par exemple comme hitbox pour les objets, mais aussi de cadre d'une feuille d'un quadtree ainsi d'une fonction de recherche `query` permettant de trouver des objets au sein d'une frontière.

3.1.2 Type objet

Générailastion des diférents objet physique du jeu (la balle, la raquette, les bricks...) composée d'une position (`vector`) une vitesse optionionelle (`vector option`) qui est `None` quand l'objet est immobile, ainsi qu'une boîte `aabb`.

3.1.3 Le type quadtree

Le quadtree est un arbre dont les noeuds sont composé de 4 branches et où les feuilles possède un nombre limité d'objet (ici égal à 4) qui ce subdivise en 4 dès que la capacité est atteinte. On a donc une fonction nous permettant d'insérer des objets dans un quadtree,

3.2 state.ml

Le fichier state va permettre de regrouper les différents objets dans un même type "state". En plus de la balle, la liste de briques et la raquette on a ajouté un int qui va représenter le score. Ce fichier va surtout permettre de réagir à une détection de collision. "contact" va utiliser par l'intermédiaire de "contact_objet" et "collisions_objet" le quadtree afin de repérer une collision. "bounce" lui va permettre de réagir et actualiser l'état de la balle suite à cette collision.

3.2.1 Le type state

Ce type permet de connaître l'état du jeu à un instant, donc composé de la balle, la raquette, la liste des briques et du score.

3.2.2 Gestion des collisions

La fonction collision_objet renvoie l'objet rentré en collision avec la balle en recherchant les objets dans la hitbox de la balle. La fonction contact_objet quand à elle réalise les différents tests de collision en créeant un quadtree et en y ajoutant tous les objets du jeu et appelle si nécessaire la fonction bounce qui permet de mettre à jour la balle après rebond.

3.3 Pistes d'amélioration

— Pour réduire la complexité temporelle des opérations de recherche, il est important de s'assurer que le quadrangle est équilibré. En outre, l'étude de méthodes d'insertion plus efficaces peut contribuer à réduire le temps nécessaire à la construction de l'arbre quadratique.
— Lorsque la balle entre en collision avec une surface, il est crucial de calculer correctement la normale à la surface. Ce calcul détermine la direction du rebond.
— Ajustez la réponse aux collisions en modifiant la vitesse de la balle en fonction de la normale calculée. Utilisez une approche basée sur la physique pour obtenir un rebond réaliste.
— Définir des règles spécifiques d'attribution de points en fonction du type d'impact, comme le rebond sur la raquette ou la destruction d'une brique. Intégrer un affichage dynamique des scores dans l'interface graphique pour permettre aux joueurs de suivre leur progression.

4. LE PROGRAMME PRINCIPAL

4.1 game.ml

Ce fichier permet de paramétrer le jeu comme on le souhaite en initialisant l'état initial du jeu :

— La position de la balle avec une vitesse initiale et son rayon ;
— La disposition des briques sur le terrain avec leur largeur, leur hauteur et leur couleur ;
— La position de la raquette ;
— Le score initial.

4.2 newtonoid.ml

Ce programme est ainsi le programme principal qui génère un flux d'état du jeu (type state) grâce aux fonctions inspirées du TP7 run_FF et run. Ces 2 fonctions appellent les fonction integre et unless produite lors du TP7, que nous avons mises dans le module Iterator car elles s'appliquent aux flux. Le flux d'état est ainsi affiché sur l'interface graphique via la fonction draw_etat qui appelle toutes les fonctions d'affichage des objets (balle, raquette et briques). En revanche, Nous n'avons pas trouvé comment récupérer l'objet en contact avec la balle à temps pour la faire rebondir. Par conséquent, la balle ne détecte pas la raquette, et elle se bloque dans les briques. Comme nous n'avons pas réussi à gérer la collision des objets, nous n'avons pas implémenté le système de score.

5. FONCTIONNALITÉS

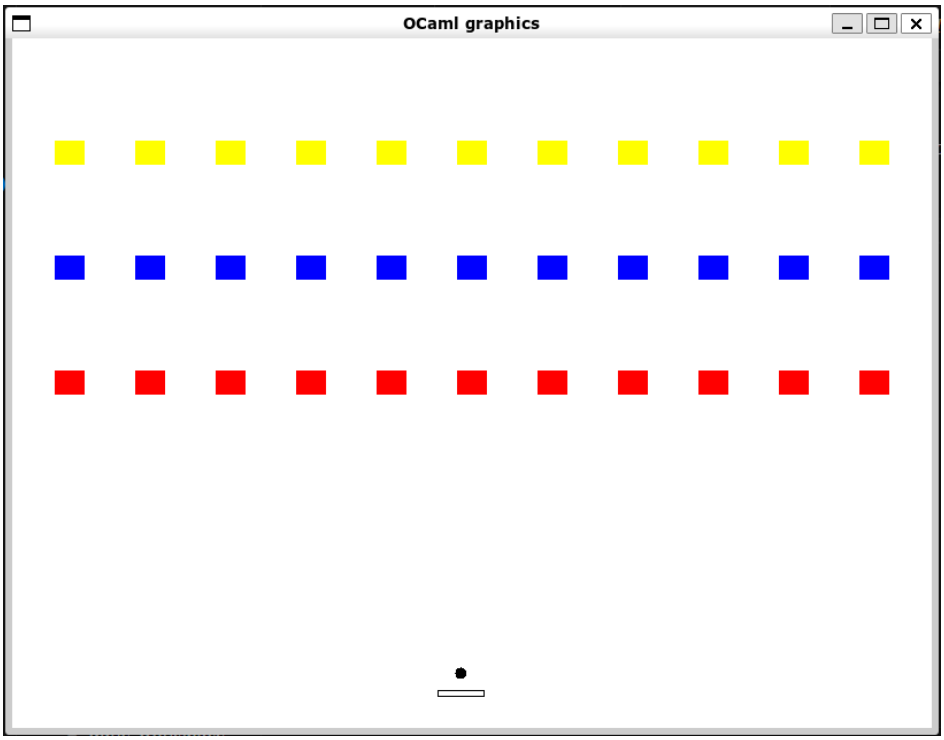

FIGURE 5.1 – Newtonoid

5.1 Fonctionnalités actuelles

Actuellement, Newtonoid possède plusieurs caractéristiques fondamentales :

5.1.1 Affichage des objets

— La balle est affichée à l'écran avec une position dynamique qui se met à jour au fil du temps. Cependant, le rebond sur les objets n'a pas encore été correctement implémenté.
— La raquette est bien affichée et peut réagir à l'interaction de la souris.
— Les briques sont disposées sur le terrain de jeu et affichées en différentes couleurs.

5.1.2 Détection des collisions

— Le système de détection des collisions est partiellement implémenté, mais pour l'instant la balle ne rebondit pas de manière réaliste sur la raquette ou les briques.

5.2 Pistes d'amélioration

5.2.1 Rebond de la balle

— Améliorez la gestion des collisions en calculant correctement la normale de la surface lorsque la balle entre en collision avec un objet.
— Affiner la réponse aux collisions en ajustant la vitesse de la balle en fonction de la normale calculée.
— L'ajout d'inertie à la raquette donnera une sensation plus réaliste lors de l'interaction avec la balle.

5.2.2 Interactions avancées

— Ajouter des accélérations temporaires à la balle pour augmenter la difficulté.
— Réduction de la taille de la raquette, pour augmenter le défi.
— Mettre en place un système de niveaux avec une progression de la difficulté, en ajoutant des briques ou en augmentant la vitesse de la balle.
— Intégrer un affichage dynamique des scores dans l'interface graphique pour permettre aux joueurs de suivre leur progression.
— Définir des règles précises d'attribution de points en fonction du type de collision, comme le rebond d'une raquette ou la destruction d'une brique.

5.2.3 Amélioration dans le code

— Redéfinir la fonction *bounce* pour y inclure les calculs normaux et les ajustements de vitesse.
— Intégrer de nouvelles fonctions pour gérer les interactions avancées.
— Revoir le code existant pour l'optimiser et s'assurer qu'il fonctionne correctement avec les nouvelles fonctionnalités.

6. CONCLUSION

En conclusion, l'implémentation de la gestion des collisions entre la balle et les objets du jeu reste un défi non résolu dans le fichier state.ml. Bien que le quadtree ait été intégré dans la gestion de l'espace de jeu, les problèmes de collision non résolus ont entravé la réalisation d'un jeu pleinement fonctionnel. Malgré ce revers, nous avons acquis une expérience précieuse dans la conception de jeux en OCaml et la gestion d'espaces de jeu complexes. Les travaux futurs devraient se concentrer sur la résolution des problèmes de collision et sur l'amélioration des mécanismes de rebond de la balle.