N7EN13A : Programmation Fonctionnelle

2A Sciences du numérique - Parcours L

Programmation Fonctionnelle
Projet Newtonoid

Daphné Navratil, Pierre Saussereau, Alexandre Perrot

Table des matières

1 Introduction

Le projet Newtonoid est un jeu de casse-briques développé dans le cadre du cours de Programmation Fonctionnelle à l'ENSEEIHT. Ce jeu combine des éléments classiques du casse-briques type Arkanoid avec la gravité en plus. Le but du jeu est de détruire toutes les briques présentes à l'écran en utilisant une raquette contrôlée par la souris pour renvoyer la balle.

Ce rapport présente les différents aspects du développement de Newtonoid, en mettant l'accent sur les structures de données utilisées, les modules principaux, et les mécanismes de gestion des collisions. Nous expliquerons comment nous avons structuré le code pour gérer l'état du jeu, la balle, la raquette, et les briques, ainsi que l'utilisation d'un quadtree pour optimiser la détection des collisions.

2 Lancement du jeu

Afin de pouvoir lancer les tests unitaires ainsi que le jeu, vous pouvez exécuter les deux commandes suivantes dans le répertoire du projet

```
dune runtest                    % Pour lancer les tests unitaire
dune exec bin/newtonoid.exe     % Pour démarrer le jeu
```

Le point d'entrée de l'exécution du code se trouve dans le fichier *newtonoid.ml*, qui possède également les fonctions d'affichage graphique.

3 Modularité des paramètres des différents éléments du jeu

Nous avons créé différents modules réunis dans un fichier "parametresJeu.ml", permettant de définir les paramètres des éléments du jeu.

3.1 L'espace de jeu : module Box

Le module Box définit le format de l'espace de jeu :

- le coordonné en x du coin inférieur gauche (*infx*)
- le coordonné en y du coin inférieur gauche (*infy*)
- le coordonné en x du coin supérieur droit (*supx*)
- le coordonné en y du coin supérieur droit (*supy*)
- une marge (*marge*)

3.2 La balle : module ParametresBalle

Le module ParametresBalle définit le format de la balle et des paramètres associés à son mouvement :

- son rayon (*rayon*)

— sa vitesse initiale (*vitesse_initiale*)
— son accélération initiale (*acceleration_initiale*) qui est fixé à 0 en -98.1 en y pour que la balle respecte la gravité
— les gains de vitesse de la balle lorsque qu'elle rebondit sur un élément (brique, raquette, murs), qui permettent notamment conserve une vitesse suffisante malgré la gravité pour que le jeu continue (*gain_vitesse_touche_brique*, *gain_vitesse_touche_raquette*, *gain_vitesse_touche_mur*)

3.3 La raquette : module FormeRaquette

Le module FormeRaquette définit les variables suivantes :
— la hauteur de la raquette (*hauteur*)
— sa longueur (*longeur*)
— la borne supérieure du coefficient de changement de direction de la balle lorsqu'elle rebondit sur la raquette (*max_edge_shot*)

3.4 Les briques : module ParametreBrique

Le module paramètre brique définit la taille, le positionnement des briques :
— *espace_entre_briques* : est une variable booléenne qui définit si les briques sont espacées ou non.
— *nbColonnes* : définit le nombre de colonnes de briques. Si les briques sont espacées, les briques seront présentes une colonne sur deux.
— *nbLignes* : définit le nombre de lignes de briques
— *voir_bordures_quadtree* : est une variable booléenne qui permet d'afficher ou non les bordures du quadtree expliqué dans la partie 3.4.

4 Modules et types principaux pour décrire l'état du jeu et de ses éléments

4.1 Le module EtatJeu

Le module EtatJeu permet de décrire l'état global du jeu. Une instance de ce module est composée de :

- Deux alias de type entier permettant de décrire le nombre de vies et le score au cours d'une partie :
  — score
  — vies
- Et des instances de trois modules pour décrire l'état de la raquette, de la balle, et des briques :
  — EtatBalle
  — EtatEspaceBrique
  — EtatRaquette

Ce module possède des fonctions pour initialiser un état du jeu, et accéder à ses différentes composantes.

4.2 Le module EtatBalle

Le type EtatBalle permet de décrire l'état de la balle et contient :

— position (float*float) : sa position en x et y

— vitesse (float*float) : sa vitesse en x et y

— acceleration (float*float) : son accélération en x et y

Nous nous sommes aperçu plus tard qu'il n'était finalement pas nécessaire de stocker l'accélération de la balle, et la variable accélération pourrait donc être supprimée. Comme pour EtatJeu, le module EtatBalle possède des fonctions pour initialiser un état de balle, et accéder aux différents éléments qui composent un état de balle, comme la position de la balle avec la fonction *position* par exemple.

4.3 Le module EtatRaquette

Le type EtatRaquette permet de décrire l'état de la raquette et contient :

— position (float*float) : sa position en x et y qui correspond en réalité à a position de la souris

— clique (bool) : qui décrit si le joueur à cliquer avec la souris. Au début d'une partie, ou lorsque la balle doit être relancée après être tombée, la balle reste positionnée au centre de la raquette et n'est lancée qu'une fois que le joueur a cliqué.

Encore une fois, le module EtatRaquette possède des fonctions pour initialiser un état de raquette, et accéder facilement aux différents éléments qui composent un état de raquette.

4.4 Le module EtatEspaceBrique

Afin de pouvoir représenter les briques dans notre jeu, nous avons opté pour une structure de données nommée **QuadTree**, notamment parce qu'une brique contient sa position (X, Y), sa longueur et sa hauteur. Ce fait nous permet de stocker efficacement les briques

Un quadtree (arbre quaternaire) est un arbre dont chaque nœud contient exactement 4 fils. Chaque fils peut contenir un noeud, une feuille, ou être vide, pour représenter la terminaison. À l'ajout d'une nouvelle brique, on divise le nœud en 4 sous-portions égales, correspondant à des portions de l'écran (Nord-Ouest, Nord-Est, Sud-Ouest, Sud-Est), et on ajoute la brique à la portion correspondant à l'emplacement de la brique. L'idée de cette structure de données est de pouvoir rechercher efficacement, car on peut diviser un nœud en quatre sous-arbres en fonction des coordonnées de la balle. Si on a un nœud contenant quatre feuilles (qui sont des briques) et que la position de la balle est à l'intérieur d'un fils, alors on sait qu'on doit rechercher dans cette portion, divisant le temps de recherche par 4 au lieu de rechercher la brique la plus proche de la balle itérativement.

Pour la gestion des briques, nous avons trois types et quatre fonctions pour le bon fonctionnement :

— Le type *point* représente les coordonnées (X, Y)

— Le type *rect* représente un rectangle dans le quadtree (X, Y, Longueur, Hauteur) nous permettant de connaître les délimitations de cette portion.

— Le type *brique* représente une brique en stockant la position (X, Y) d'une brique (situé en bas à gauche), sa longueur, sa hauteur et sa couleur.

Le type *etatEspaceBrique* représente un quadtree, et plus précisément, il y a 3 types pour représenter un quadtree :

— Vide = L'arbre est vide

— Brique of brique = L'arbre est une brique

— Noeud of rect * etatEspaceBrique * etatEspaceBrique * etatEspaceBrique * etatEspaceBrique = L'arbre est un nœud contenant son aire ainsi que ses 4 fils

La fonction ajouter_brique permet d'ajouter une nouvelle brique à un quadtree de taille spécifiée (la taille du jeu par exemple), cette fonction vérifie le type du quadtree

— Si le quadtree est vide, alors on remplace le type Vide par Brique

— Si le quadtree est une Brique, alors on divise le quadtree par 4 fils de taille égale et on ajoute la brique existante et la nouvelle brique dans le quadtree avec 4 fils de façon récursive

— Si le quadtree est un Noeud, alors on vérifie les coordonnées de la brique qu'on souhaite ajouter et on regarde dans quel portion cette brique doit appartenir, et on ajoute cette brique à cette portion de façon récursive jusqu'à tomber sur un quadtree Vide

Le principe est le même pour la fonction retirer_brique, sauf que, si le quadtree est une Brique qui est égale à la brique que l'on souhaite retirer, alors on remplace ce quadtree par Vide et on s'assure que si le quadtree est un Noeud qui contient que des fils vides, alors on remplace ce noeud par Vide.

La fonction query nous permet de rechercher dans un quadtree la liste des briques qui touchent le rectangle passé en paramètre (la surface de la balle par exemple), ou bien de pouvoir récupérer l'ensemble des briques affichées dans le jeu. Afin d'effectuer la recherche, on vérifie le type du quadtree :

— Si le quadtree est vide, on retourne une liste vide

— Si le quadtree est une brique et qui est en collision avec le rectangle passé en paramètres, alors on ajoute cette brique à la liste

— Si le quadtree est un nœud et que le rectangle passé en paramètres est en collision avec ce noeud, alors on cherche récursivement dans ses quatre fils et on l'ajoute à la liste.

La fonction collision_avec_brique possède la même logique que la fonction query sauf qu'au lieu de retourner une liste de briques, cela renvoie un booléen si le rectangle passé en paramètres touche au moins une brique dans le quadtree, cette fonction permet de détecter si la balle touche une brique à n'importe quel moment du jeu, et si c'est le cas, d'effectuer l'appel à query qui renverra la brique qui a touché la balle.

Ci-dessous se trouve une illustration du quadtree avec les bordures en rouge. Chaque rectangle rouge représente un nœud.

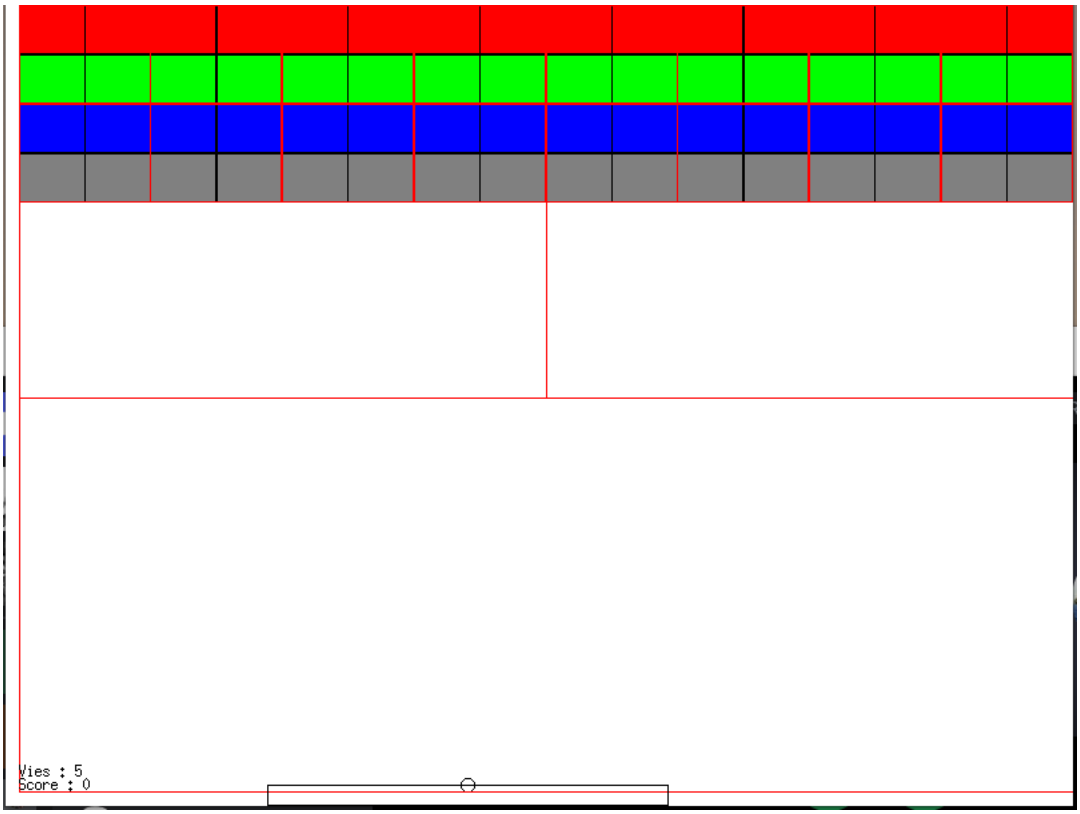

FIGURE 1 – Illustration du quadtree avec toutes ses briques

On remarque que la première bordure rouge autour de l'espace du jeu représente le premier nœud du quadtree et que les bordures plus petites en haut représentent la division du quadtree en quatre fils. On remarque également une délimitation au milieu en haut, mais pas en bas, car nous ne dessinons pas les quadtree qui sont vides. Mais s'il y avait une brique en bas à gauche, il y aurait également une division au milieu en bas.

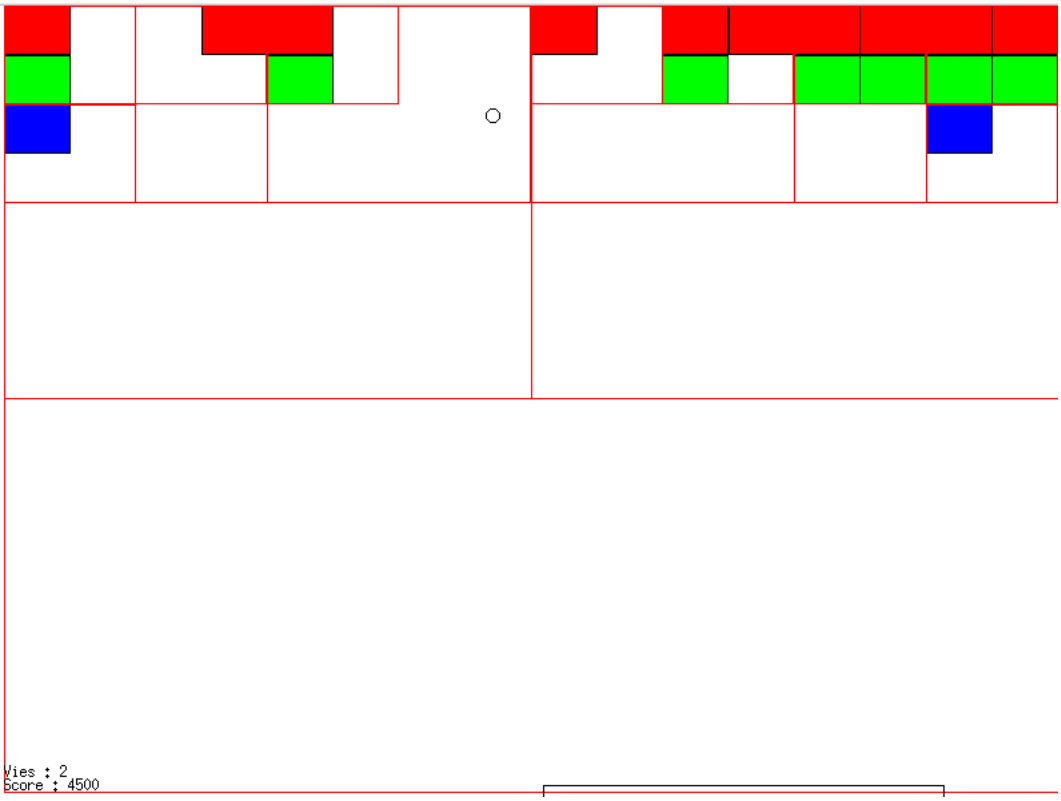

FIGURE 2 – Illustration du quadtree avec quelques briques cassées

5 Chute de la balle avec le module FreeFall

Le mouvement de la balle est définit par le module FreeFall (FreeFall.ml). Il contient une première fonction qui permet d'intégrer les valeurs successives d'un flux, et une seconde fonction (*run*, qui à partir d'un état de balle initial, permet de créer un flux d'état de balles décrivant le mouvement de la balle en chute libre. Cette seconde fonction, en utilisant la première, calcule les positions successives de la balle, sa vitesse, et son accélération au cours du mouvement.

6 Détection des collisions entre la balle et les autres éléments du jeu et gestion des rebonds

Le module GestionBalle dans le fichier gestionBalle.ml regroupe les différentes fonctions qui permettent de détecter les contacts entre la balle et les autres éléments du jeu, et de mettre à jour l'état de la balle, plus précisément sa vitesse, pour créer un rebond cohérent.

6.1 Détection des collisions

Dans ce module, une fonction *contact*, qui prend en paramètre un état de jeu, appelle les différentes fonctions décrites par la suite et donne la valeur "vrai" s'il y a contact entre la balle et l'un des éléments du jeu.

6.1.1 Détection de collision de la balle avec les murs

La fonction *contact_1d* est utilisée pour détecter un contact de la balle avec le plafond ou les côtés de l'espace de jeu. Lorsque la balle entre en contact avec le sol, elle tombe. Ce contact est détecté avec la fonction *contact_sol* qui prend en paramètre un état de balle et vérifie que sa position en y soit inférieure au sol et qu'elle soit en train de tomber, c'est-à-dire que sa vitesse en y soit négative. Cette fonction correspond à un cas particulier et non pas réellement à un contact. Elle permet de détecter dans quel cas la balle doit chuter et le joueur perdre une vie, elle n'est donc pas appelée par la fonction *contact* citée précédemment.

6.1.2 Détection de collision de la balle avec la raquette

La fonction *contact_raquette* qui prend en paramètres un EtatBalle et un EtatRaquette détecte les collisions entre la balle et la raquette en vérifiant si la position de la balle est comprise dans la raquette, élargie du rayon de la balle et d'une légère marge pour éviter que le joueur puisse voir la balle entrer à l'intérieur de la raquette. La fonction vérifie aussi que la balle se dirige vers la raquette (vitesse en y négative) et qu'elle ne soit pas en train de s'éloigner, et donc que le rebond aurait déjà eu lieu.

6.1.3 Détection de collision de la balle avec les briques

La fonction *gerer_contact_brique* qui prend en paramètre un EtatJeu et un EtatBrique permet de gérer les événements lorsqu'une balle touche une brique. Grâce au module *EtatEspaceBrique*, nous avons deux fonctions *collision_avec_brique* et *query* qui servent respectivement a détecter si la balle a touché une brique, et de récupérer la brique touchée. Lorsque une collision entre la balle et la brique a été détectée, on récupère la brique touchée, on la retire du jeu et on fait rebondir la balle en fonction de la direction dans laquelle elle a touchée la brique.

6.2 Gestion des rebonds

Différentes fonctions du module GestionBalle permettent de mettre à jour l'état de la balle selon l'élément avec lequel elle est entrée en collision.

6.2.1 Rebond de la balle sur les murs

Si la balle entre en contact avec les côtés, la direction de sa vitesse horizontale est inversée, tandis que si elle entre en contact avec la limite supérieure de l'espace de jeu, sa vitesse verticale est inversée et devient négative. Ces comportements de rebonds sont décrits par la fonction *rebond* qui prend en entrées un EtatBalle.

6.2.2 Rebond de la balle sur la raquette

La fonction qui permet de définir le rebond de la balle sur la raquette est *rebond_raquette*. Elle prend en entrée un EtatBalle et un EtatRaquette. Mais la direction de sa vitesse horizontale dépend de l'endroit où la balle entre en contact avec la raquette : au centre, elle repart avec une vitesse horizontale nulle, et plus elle tape sur les côtés de la raquette, plus sa vitesse horizontale est importante. De plus, si la balle tape sur le côté gauche, elle repart vers la gauche et si elle tape du côté droit, elle repart à droite. Pour définir ce comportement, cette fonction calcule un coefficient (*coeff_direction*) qui correspond à la valeur absolue de

la différence entre la position de la balle et de la raquette en x divisée par la demi-longueur de la raquette. Ainsi, ce coefficient vaudra 0 si la balle arrive au centre de la raquette, et 1 si elle tape à l'une des extrémités. Mais une valeur proche de 1 fait partir la balle fortement à l'horizontale. Pour éviter ce phénomène, nous avons ajouté le fait que si le coefficient puisse être borné par une valeur définie dans le module FormeRaquette : *max_edge_shot*. En suite, les vitesses horizontales et verticales de la balle sont mises à jour :

— la nouvelle vitesse en x vaut la sommes des valeurs absolues des vitesses initiales en x et y multipliées par le coefficient

— la nouvelle vitesse en y vaut la sommes des valeurs absolues des vitesses initiales en x et y multipliées par 1 moins le coefficient.

6.2.3 Rebond de la balle sur les briques

La fonction principale qui définit comment la balle rebondit sur une brique est *getVitesseRebondBrique*. Pour savoir comment mettre à jour les vitesses horizontales et verticales de la balle afin de créer un rebond opposé au côté touché de la brique, la fonction teste la position de la balle par rapport aux diagonales du rectangle représentant la brique. Si sa position en y est inférieure ou supérieure aux deux diagonales, alors la balle a touché la brique sur son côté haut ou bas. Pour créé le rebond, la vitesse verticale de la balle est donc inversée. Dans le cas contraire, la balle à touché le côté gauche ou droit de la brique, c'est donc sa vitesse horizontale qui est inversée.

7 Gestion du déroulement d'un jeu

Le module Jeu dans le fichier *bin/jeu.ml* possède l'ensemble des fonctions permettant le déroulement d'une partie de jeu.

7.1 Gestion du déroulement du jeu après une collision

Les fonctions décrites ici servent à altérer l'état du jeu en fonction du type de collision. La fonction *gerer_contact* s'occupe de vérifier le type de collision et d'appeler la fonction d'altération de l'état du jeu correspondant.

7.1.1 Etat du jeu après une collision avec le sol

La fonction *gerer_contact_sol* modifie l'état du jeu en détruisant en faisant perdre supprimant une vie au joueur, et en initialisant une nouvelle balle collée à la raquette.

7.1.2 Etat du jeu après une collision avec un mur

La fonction *gerer_contact_mur* modifie l'état du jeu en faisant rebondir la balle. Le score et le nombre de vie reste inchangé.

7.1.3 Etat du jeu après une collision avec la raquette

La fonction *gerer_contact_raquette* modifie l'état du jeu en faisant rebondir la balle (en gagnant de la vitesse), le score et le nombre de vie reste inchangé.

7.1.4 Etat du jeu après une collision avec une brique

La fonction *gerer_ contact_ brique* modifie l'état du jeu en faisant rebondir la balle (en gagnant de la vitesse), en supprimant la brique de l'espace de brique, en augmentant le score de 100 points et le nombre de vie reste inchangé.

7.2 Lancement du jeu

La fonction *run* s'occupe, à partir d'un état du jeu (et d'une condition qui indique si la balle est collée ou non à la raquette, donc pas encore lancée) produit un flux d'état du jeu correspondant au déroulé de la partie.

La fonction s'occupe de vérifier que l'utilisateur possède encore des vies. Ensuite, si la balle est collée à la raquette, le flux produit attend un clique souris, indiquant le début du lancement de la balle. Le reste du flux est ensuite le déroulé du jeu tant qu'aucune collision n'est pas détectée. Si une collision est détectée, le flux s'arrête, et on appelle la fonction d'altération de l'état du jeu (*gerer_ contact*). Puis, on rappelle récursivement la fonction *run* avec ce nouvel état modifié pour fournir le reste du flux.

8 Limites et propositions d'améliorations

Bien que nous ayons un jeu fonctionnel qui se rapproche du jeu Arkanoid tout en tenant compte de la constante de gravité, nous constatons également des limites à notre jeu, ainsi que des fonctionnalités que nous souhaitions implanter mais qui, par manque de temps, n'ont pas été concrétisées.

Tout d'abord, nous aurions voulu enrichir les fonctionnalités autour des briques. Par exemple, le fait qu' une brique nécessite d'être touchée plusieurs fois pour qu'elle se casse complètement. Pour se faire, nous pourrions modifier le type d'une brique en ajoutant un compteur du nombre restant de collisions nécessaires avant que la brique ne se détruise.

Ensuite, sous certaines configurations du jeu (vitesse initiale trop élevé), ou bien en visant des bords des briques, il arrive que la détection de collision ne fonctionne pas, ou bien qu'elle arrive tardivement, ce qui laisse la balle traverser les briques, ou bien ne la fait pas rebondir. Ce bug peut être atténué en choisissant judicieusement les paramètres du jeu.

Enfin, concernant le module Jeu, la structure du code permettant de gérer le flux du jeu est assez complexe, en faisant appel aux fonction *unless* et *unless_ modif* dès qu'un évènement se produit. Bien que la logique du code soit cohérente, nous constatons qu'il aurait été plus approprié d'utiliser des continuations pour gérer les états du jeu en fonction de l'état de la balle.

9 Conclusion

En conclusion, ce projet nous a permis de mettre en pratique des concepts avancés de programmation fonctionnelle et événementielle sous la forme d'un casse-brique. Nous avons implémenté un jeu de casse-briques traditionnelle avec de la gravité en plus.

Ce projet nous a permis de mettre en place des structures de données optimisés comme le quadtree pour gestion de la collision avec une brique. Ainsi que des fonctionnalités telle que la raquette, le rebond, un système de score et de vies. Cependant, il reste des pistes d'améliorations comme l'ajout de plusieurs vies à une brique.