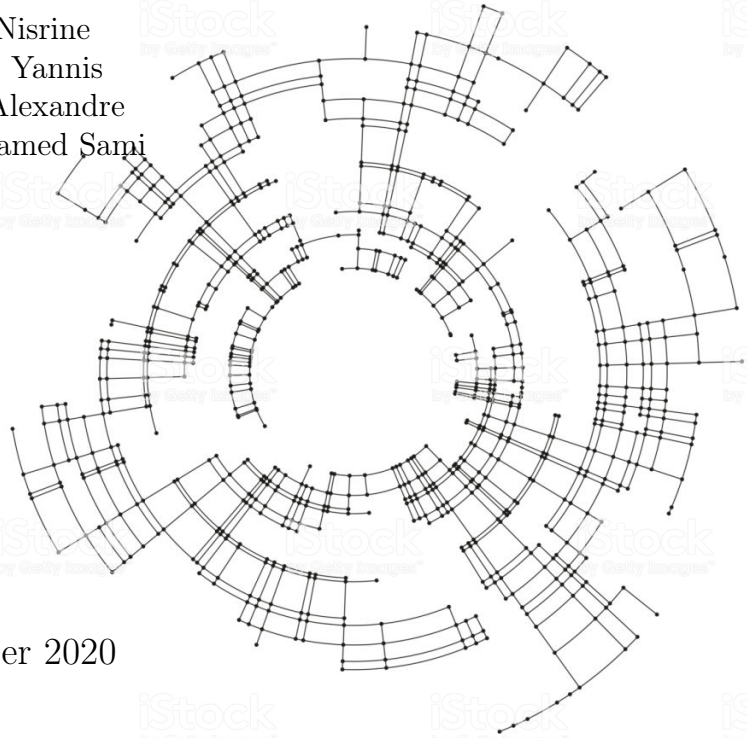
ENSEEIHT

Programmation Fonctionnelle

Rapport du projet Arkanoid

Amrir Nisrine
Benabbi Yannis
Chatain Alexandre
Touili Mohamed Sami



24 janvier 2020

Table des matières

Table des figures

1 Introduction

L'objectif de ce projet est l'implantation d'une version du jeu *Arkanoid* commercialisé en 1986. Ce rapport détaille donc la reproduction de ce type de jeu *casse-briques*. Le principe de ce jeu est le suivant :

- Le joueur déplace de droite à gauche une barre horizontale représentée par notre **raquette**.
- Il perd une **vie** à chaque fois qu'il laisse filer la balle au bas de l'écran (le nombre de vie est initialisé à 3). Lorsqu'une vie est perdue, les effets sont réinitialisés.
- Le joueur passe à un **niveau supérieur** lorsque toutes les briques du niveau courant sont détruites.
- Les **briques** peuvent être détruites en plusieurs coups.
- Les briques relâchent parfois des **bonus** (ou malus) choisis aléatoirement. Parmi ces bonus/malus, nous avons :
  - *L'augmentation (ou diminution) de la taille de la raquette*
  - *L'ajout d'une balle supplémentaire*
  - *L'ajout d'une vie supplémentaire*
  - *L'inversion de contrôles*
  - *La possibilité de traverser les blocs pendant une durée limitée*
  - *Le gain de points*

Nous avons implémenté ce jeu à l'aide de modules et de flux, ainsi qu'avec la librairie Graphics. Nous avons défini un module pour chaque élément de jeu (Balle, Raquette, Bloc et Bonus), un autre pour le Jeu en lui même, et repris les éléments du *tp7* en tant que base pour l'utilisation de Graphics et une version basique des collisions.

2 Les modules de préparation

2.1 Le module solide

Ce module sert avant tout à définir des méthodes pour simplifier l'écriture des calculs de positions. Il sera inclus dans tous les modules physiquement représentés, et dont il est nécessaire de détecter une collision : *Balle*, *Raquette*, **Bonus** et *Bloc*.
Un solide est défini comme ayant une position (celle du coin inférieur gauche), sa hauteur, sa longueur, sa couleur, et ses paramètres spéciaux. Son type est donc :

```
type ('a, 'b) t = ('b * 'b) * 'b * 'b * color * 'a
```

Les paramètres spéciaux correspondent aux paramètres qui ne sont pas communs aux 4 modules cités précédemment. On peut par exemple citer la vitesse de la balle, l'effet d'un bonus, le sens de contrôle de la raquette, ...
D'autre part, nous avons fait le choix d'abstraire le type de nombre utilisé pour en étendre son utilisation. Dans notre cas, nous utiliserons le module Nombre définissant le type nb.

2.2 Le module Nombre

Nous avons défini un module nombre dans lequel se trouve un type *nb* défini tel que :

```
  type nb =
    | Int of int
    | Float of float
```

Cela permet d'établir des opérations sans se soucier du type du nombre fourni en argument. En effet, entre affichage graphique et traitement des déplacements, un grand nombre de fonctions type *int_of_float* ou *float_of_int* auraient été requis.
Ce choix a un coût significatif dans le nombre d'opérations à implémenter dans la méthode, puisque (presque) toutes les opérations binaires ont été redéfinies :

```
module type NombreItf =
sig
    (* Operations mathematiques *)
    val ( /- ) : nb -> nb -> nb (* Entre deux nb *)
    val ( /~ ) : nb -> int -> nb (* Entre un nb et un entier *)
    val ( /~. ) : nb -> float -> nb (* Entre un nb et un flottant *)
    (* +~ et +~. sont aussi definies, et ainsi de suite... *)
    val ( +- ) : nb -> nb -> nb
    val ( -- ) : nb -> nb -> nb
    val ( *- ) : nb -> nb -> nb

    (* Comparaisons *)
    val ( >> ) : nb -> nb -> bool
    val ( << ) : nb -> nb -> bool
    val ( >>= ) : nb -> nb -> bool
    val ( <<= ) : nb -> nb -> bool

    (* Transformation de int/float a nb *)
    val toInt : int -> nb
    val toFloat : float -> nb
```

```
21
22      (* Transformation de nb a int/float *)
23      val intv : nb -> int
24      val floatv : nb -> float
25
26  end
```

3 Les modules des objets

3.1 Généralités

Tous les modules de cette section ont quelques points communs :

- Tous sont représentables graphiquement (donc ont une méthode *draw*)
- Tous sont des solides (donc de type t)
- Tous ont des paramètres spéciaux (et donc un type consacré à ces paramètres)
- Tous manipulent le même type de nombre *nb* (donc travaillent avec *Nombre.nb*, et non pas *XXX.nb*)
- Tous peuvent être construits à l'aide d'un cons, variant d'un objet à l'autre (*override* de celui inclus via le module *Solide*)
- Certains ont un état de départ, accessible via *init*
- Tous ont des getters et des setters sur leur paramètres spéciaux.

L'avantage de cette structure est la facilité avec laquelle nous pouvons étendre le jeu. Il suffit de modifier le type des paramètres spéciaux, le constructeur, et les méthodes de type getter/setter pour aboutir à un code stable, ne restant plus qu'à traiter l'extension.

3.2 Le module Bloc

3.2.1 Type des paramètres spéciaux

Les blocs ont un paramètre de type *tb*, représentant les caractéristiques spéciales d'un bloc. Pour un bloc, il est nécessaire de connaître le score propre au bloc, ainsi que le nombre de coups nécessaires pour être détruit (appelé puissance dans ce document, et *power* dans le code). Donc :

```
1 type tb = int * int
```

3.2.2 Génération de couleur

Le but de la méthode de génération de couleur est de retourner la couleur du bloc en fonction de sa puissance.
La génération de couleur est réfléchie de la façon suivante : On considère un repère en 3 dimensions, aux axes (r, g, b). On considère qu'une couleur est représentée par un triplet (rx, gy, bz), où chaque composante représente l'intensité d'une couleur comprise entre 0 et 1. Le cube de côté 1, et dont 3 des faces sont sur chacun des plans (r,g) (r,b) (g,b) contient donc toutes toutes les couleurs représentables. On nommera ce cube C.
On dira qu'une couleur v est agréable lorsque les propriétés suivantes sont vérifiées :

1. $\|v\|_2 = 1$
2. $\exists i \in \{1, 2, 3\}, v_i = 0$

La première condition correspond à la vivacité de la couleur, la seconde à son contraste. Nous avons donc construit un chemin cyclique dans C ne possédant que des couleurs agréables, dont voici un schéma :

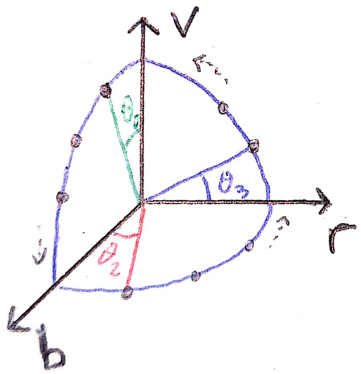


FIGURE 1 – Représentation du chemin cyclique des couleurs agréables choisi

3.2.3 Génération de blocs

Lors de la génération d'un niveau, il faut initialiser la quantité de blocs, leur puissance et leur disposition. Ces paramètres sont manipulables via les méthodes internes suivantes :

```
let fun_evo_lignes_nb = fun niveau hauteur -> (7 + 2 * niveau)
let fun_evo_lignes_pw = fun niveau hauteur -> (niveau + ((niveau *
    hauteur / 3 )) )
let fun_nb_lignes = fun niveau -> 8
let propInvincibles = 0.1
let espacement = 5
```

La première méthode permet de déterminer le nombre de blocs en fonction du niveau et de la hauteur de la ligne à former. La seconde détermine la puissance des blocs d'une ligne en fonction du niveau et de la hauteur de la ligne. Le troisième paramètre détermine le nombre de lignes en fonction du niveau (même si, dans cet exemple, le niveau n'est pas pris en compte). Le quatrième paramètre représente la probabilité qu'un bloc dessiné soit un bloc invincible. Enfin, le dernier paramètre représente l'espacement entre les blocs (espacement x et espacement y).

3.3 Le module Raquette

3.3.1 Type du paramètre spécial

Il existe un bonus inversant le sens de la raquette par rapport à la position de la souris. Si cet effet est actif, il est nécessaire de mémoriser l'information dans ce paramètre spécial. Ainsi :

```
type tb = bool
```

3.3.2 Déplacement à la souris

Pour que le déplacement à la souris soit effectif, il est nécessaire de mettre à jour les méthodes de position `xg`, `xc` et `xd`, en calculant la position à partir de `Graphics.mouse_pos`.

3.4 Le module Bonus

3.4.1 Type du paramètre spécial

Un bonus (comprendre ici, bonus et/ou malus) est principalement une fonction qui fait évoluer un élément de jeu. Il s'agit donc de fonctions prenant en paramètre un élément de type `'a` à valeurs de type `'a`, représentant l'élément dans l'état évolué. Un type `'a evo` est créé pour cet effet.
Il faut aussi pouvoir contenir tous les bonus dans une liste, et savoir si le bonus agit sur une raquette, une balle, etc... Pour cela, nous avons créé un type *bonusType* qui est un type somme, donc chaque définition de type est de la forme `'a evo`, permettant d'accéder aux méthodes via un `match ... with`.
Enfin, un bonus a aussi une vitesse et une probabilité relative aux autres bonus. En effet, il est possible de créer des bonus ayant par exemple 3 fois plus de chances d'apparaître qu'un autre bonus. Ainsi, nous avons :

```
type 'a evo = ('a -> 'a)

type bonusType =
  | BonusRaq of (Raquette.tr, nb) Raquette.t evo
  | BonusBal of (Balle.tba, nb) Balle.t list evo
  | BonusVie of int evo
  | BonusScore of int evo

(* Parametre special *)
type tbo = bonusType * (nb * nb) * float
```

3.4.2 Exemples de bonus

```
let tailleRaquetteInv = BonusRaq(Raquette.changeSens)
let vieSupp = BonusVie(fun vies -> vies + 1)
let multiballes = BonusBal(fun listeBalles -> Balle.dupliquer
    listeBalles)
let acceleration = BonusBal(fun listeBalles -> List.map( fun balle ->
    Balle.(setVit balle (dx balle *~. 1.2, dy balle *~. 1.2) ))
    listeBalles)
```

3.5 Le module Balle

La gestion des rebonds est détaillée dans une partie dédiée.

3.5.1 Type du paramètre spécial

Une balle possède une vitesse. Aussi, un bonus permet de lui faire traverser les blocs pendant un certain temps, quantifié par un nombre de rebonds (entier). De plus, lorsque la balle possède ce bonus, il est nécessaire de mémoriser les blocs qui ont déjà été pénétrés pour éviter qu'à chaque pas de temps `dt`, la balle enlève une couche de bloc, ce qui donne l'impression que la balle détruit instantanément tout bloc sur son passage. Le type de la balle est donc :

```
type tba = (nb * nb) * int * (Bloc.tb, nb) Bloc.t list
```

Remarque : Le fait que la balle mémorise la liste des blocs touchés à l'itération précédente est aussi utile en dehors du traitement de ce bonus. En effet, cela permet de déboguer des cas très particuliers, notamment celui où la vitesse est extrêmement grande, qui peut faire planter le programme lorsque la balle se retrouve dans une mauvaise position (ce qui peut empirer si les blocs en question sont invulnérables).

3.5.2 Contact avec la raquette

Lorsque la balle entre en contact avec la raquette, elle pivote d'un certain angle dépendant de la distance entre le centre de la balle et le centre de la raquette. C'est pourquoi une méthode `pivote` est rendue disponible dans le module.
Pour éviter des situations de balles avec une direction horizontale (ou presque, et donc impossible à atteindre), nous avons imposé à la vitesse d'avoir un angle compris entre $\frac{\pi}{8}$ et $\frac{7\pi}{8}$.

3.6 Le module Jeu

3.6.1 Type d'un jeu

Ce module contient toutes les informations du jeu à un instant t. Ainsi, il est nécessaire d'avoir l'information des balles, de la raquette, des blocs, et des bonus qui sont en train de chuter.
En plus de ces paramètres il existe des paramètres propres à la partie en cours tels que le score, le niveau actuel et les vies restantes. Pour plus de clarté, ces paramètres, ont été créés dans un second type. Nous avons alors :

```
type ('a, 'b) t =
    (Balle.tba, nb) Balle.t list *
    (Bloc.tb, nb) Bloc.t list *
    (Bonus.tbo, nb) Bonus.t  list *
    (Raquette.tr, nb) Raquette.t *
    'a
type g = int * int * int
```

3.6.2 Gestion d'application des bonus

C'est dans ce module qu'a lieu l'application de bonus. Chacun des bonus étant une fonction à appliquer, suffit alors de distinguer les différents cas :

```
let appliquer j bonus =
    let func = Bonus.func bonus in
    match func with
    | BonusRaq f -> remplRaquette j (f (raquette jeu))
    | BonusVie f -> setVies j (f (vies jeu))
    | BonusBal f -> remplBalles j (f (balles jeu))
    | BonusScore f -> setScore j (f (score jeu))
```

4 Gestion des rebonds de la balle

Comme indiqué dans les parties précédentes, la balle peut rebondir que ce soit sur la raquette, sur les briques, ou sur les bords de l'écran. Pour la raquette, il suffisait de vérifier que le bas de la balle pénétrait la raquette pour vérifier s'il y avait collision. Pour les bords, même principe, mais de tous les côtés.
En revanche, nous avons eu une approche différente en ce qui concerne les blocs.

4.1 Définitions

— **Collision** : Évènement qui surgit lorsque deux objets (Bloc - Balle, Balle - Raquette, mais pas Balle - Balle) sont en contact (Si un bloc est en contact pendant plusieurs instants avec une balle, alors il y aura de multiples collisions, mais une seule (ou zéro) sera effective.

— **Collision effective** : Collision nécessitant une modification de comportement de la balle (changement de vitesse, de direction ...) en cas de détection. Toutes les collisions ne sont pas effectives. En fait, si la balle est entré en collision avec le bloc B à l'instant t, et qu'à l'instant $t + dt$, cette collision est toujours d'actualité, alors ce bloc B ne générera pas de collision effective pour cette balle jusqu'à ce que la balle ne soit plus en collision avec B.

4.2 Première approche : Bloc par bloc

Au départ, dans notre première version, les blocs étaient suffisamment espacés pour que les collisions soient considérées comme dépendantes d'un seul bloc. Lorsque nous sommes passés à cette version, et que nous avons généré des niveaux avec des blocs avec un espacement suffisamment faible, il pouvait arriver que la balle reste bloquée entre deux blocs invulnérables, ou que cette dernière détruise trop de blocs en trop peu de temps car intercalée entre deux blocs vulnérables.
Nous avions utilisé pour cette approche les mêmes techniques que les collisions avec la raquette et les bordures de fenêtre.

4.3 Approche actuelle : Découpage de la balle

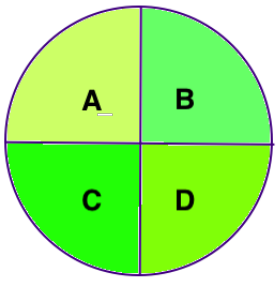


FIGURE 2 – Découpage de la balle en quatre quarts de disque

L'approche précédente avait un autre problème : La collision dans les angles considérait la balle comme "carrée", car seules les informations `xg xd yh yb` ont été utilisées (et

jamais le rayon). Cette approche corrige donc non seulement les problèmes liés au double contact, mais aussi aux collisions dans les angles.
Pour cela, il est nécessaire d'énumérer les cas. Prenons un exemple qui sera le fil directeur de cette approche : Supposons qu'une collision est détectée en a et en b avec une vitesse telle que $dx > 0$ et $dy > 0$. Cet évènement peut arriver deux fois :

— La balle arrive entre deux blocs par le haut.

— La balle arrive sur un bloc par le haut, mais n'arrive pas sur l'un de ses coins.

Dans ces deux situations, la balle a le même comportement : Ce découpage permet donc de généraliser la collision avec plusieurs blocs.
Après avoir analysé la situation, il faut déterminer son comportement. Nous avons utilisé la méthode des *Tableaux de Karnaugh* pour déterminer la condition de changement de sens pour la coordonnée x et y, en associant dans chaque case 1 à un changement nécessaire, et 0 au fait qu'aucun changement ne doit être effectué sur l'axe concerné.
Dans le tableau suivant, a, b, c, et d sont les booléens valant *true* si une collision est détectée respectivement en a, b, c ou d, et e est le booléen qui vaut *true* lorsque la vitesse selon l'axe concerné est positive. Ce tableau a été obtenu manuellement, en distinguant chaque cas.

$\overline{c}^c$	$\overline{d}\overline{e}$	$\overline{d}e$	de	$d\overline{e}$
$\overline{a}\overline{b}$	0^1	0^0	1^0	0^0
$\overline{a}b$	0^1	1^1	1^1	0^0
ab	0^1	0^0	1^0	0^0
$a\overline{b}$	1^1	0^0	1^0	1^1

De plus, pour y, on peut remarquer que le comportement de (ay, by, cy, dy, ey) est similaire à celui de (cx, dx, bx, ax, ex). Nous obtenons donc les équations logiques suivantes :

$$Rebond_x \iff c\overline{d}\overline{e} + \overline{c}de + a\overline{b}\overline{e} + \overline{a}be$$

$$Rebond_y \iff d\overline{b}\overline{e} + \overline{d}be + c\overline{a}\overline{e} + \overline{c}ae$$

En reprenant notre exemple, nous avons une collision telle que $coll = ab\overline{c}\overline{d}e_xe_y$. On en déduit :

$$dx_{t+dt} = dx_t \qquad et \qquad dy_{t+dt} = -dy_t$$

Quelques cas particuliers : Dans le tableau, nous avons fait le choix de traiter *abcd* de sorte à ne pas modifier la direction. En effet, si la vitesse de la balle est trop grande, il est possible que la balle pénètre complètement dans un bloc, auquel cas un comportement doit être prévu. Ce comportement a pour effet d'être d'une part non aléatoire, et d'autre part de ne pas faire planter le programme, qui n'arriverait pas à se décider (avec la version précédente) quoi choisir.
Il en est de même pour les cas à triple quarts en collision dont la vitesse est orientée vers "le bloc manquant".

4.4 Complexité

Pour cette partie n désigne le nombre de blocs sur le plateau de jeu, et m le nombre de balles. La recherche d'informations sur la collision nécessite de parcourir la liste des blocs, en mémorisant uniquement les zones de la balle ayant subi une collision. La vérification

d'une information de collision sur un quart de cercle de la balle utilise sept tests logiques unitaires dans le pire des cas (comparaisons de valeurs définies (coordonnées) ou calculées (distance)).
On se place dans le cas où une seule balle est en jeu. Pour ce qui est de l'algorithme dans sa globalité, il est nécessaire de :

1. Séparer les blocs morts (pour le calcul du score) des blocs restants (pour mettre à jour la liste des blocs), et de savoir quels sont les blocs qui ont été touchés par la balle. Cette opération est réalisée par `blocsBounce`, et fait appel à `Balle.collision_bloc` sur chaque bloc du plateau pour vérifier ensuite si l'un des quarts de cercle subit une collision (de complexité $O(n)$). Un test (en cas de réussite) est effectué pour savoir si la collision est effective pour le bloc donné (complexité presque unitaire).
2. On exécute ensuite la collision via `balleBounce`, utilisant `Jeu.collision_xy` qui retourne un couple déterminant si oui ou non le sens de x et/ou y doit être changé. Cette méthode renvoie à `Jeu.collision_autre` (complexité presque unitaire) et à `Balle.collision_xy_blocs` (complexité de même envergure que la méthode `Balle.emplacementCollisions`, répétant via un `fold_right` sur les blocs du plateau, la méthode `collision_bloc` (de complexité $O(n)$) .

En somme, la complexité de la gestion des collisions est de l'ordre de $O(mn^2)$

5 Contenu du jeu

5.1 Bonus et malus

Afin d'améliorer notre version finale du jeu *Arkanoid*, nous avons rajouté plusieurs bonus et malus. Ces bonus peuvent tomber (avec une probabilité de 0.3) dès lors qu'un bloc est détruit. Les bonus/malus implémentés sont les suivants :

1. **Modification de la taille de la raquette :**
   La taille de la raquette augmente (ou diminue), ce qui offre plus (ou moins) de surface pour faire rebondir la balle lorsque le joueur intercepte un bonus de couleur *verte* (ou *rouge*).
2. **Modification du nombre de balles :**
   La première balle de la liste des balles du jeu est dupliquée lorsque le joueur intercepte un bonus de couleur *blanche*.
3. **Modification du nombre de vies :**
   Le joueur obtient une vie supplémentaire lorsqu'il intercepte un bonus de couleur *rouge clair*.
4. **Pouvoir de traversée :**
   Les balles du plateau peuvent traverser les blocs, en enlevant une couche à chaque bloc traversé. Ce bonus a une durée limitée, et se désactive après 10 rebonds (Les rebonds considérés étant donc les rebonds aux murs et à la raquette). Ce bonus est de couleur *cyan*.
5. **Modification de la vitesse de la balle :**
   Augmente ou diminue la vitesse de la balle. Les deux bonus/malus sont aussi de couleur *blanche*.
6. **Bonus de score :**
   Augmente le score total de 10%. Ce bonus est de couleur *jaune*.
7. **Inversion des contrôles :**
   Les contrôles sont inversés : la raquette se déplace vers la gauche lorsque le joueur déplace la souris vers la droite, et inversement. Ce malus est de couleur *magenta*magenta.

Les bonus étant représentés par des méthodes, il est relativement aisé d'en ajouter/-supprimer, ou de modifier leurs effets.

```
1  (* Bonus disponibles *)
2  let tailleRaquetteUp =
3      BonusRaq(fun raq -> Raquette.ajouterTaille raq 20)
4  let tailleRaquetteDown =
5      BonusRaq(fun raq -> Raquette.ajouterTaille raq (-20))
6  let tailleRaquetteInv =
7      BonusRaq(Raquette.changeSens)
8  let vieSupp =
9      BonusVie(fun vies -> vies + 1)
10 let multiballes =
11     BonusBal(fun listeBalles -> Balle.dupliquer listeBalles)
12 let acceleration =
13     BonusBal(fun listeBalles -> List.map( fun balle -> Balle.(setVit
       balle (dx balle *~. 1.2, dy balle *~. 1.2) )) listeBalles)
14 let deceleration =
```

```
15      BonusBal(fun listeBalles -> List.map( fun balle -> Balle.(setVit
        balle (dx balle *~. 0.95, dy balle *~. 0.95) )) listeBalles)
16  let ballesTraversantes =
17      BonusBal(fun listeBalles -> List.map (fun balle -> Balle.setTraverse
        balle 10) listeBalles)
18  let gainPoints =
19      BonusScore(fun score -> int_of_float (float_of_int score *. 1.1))
20
21  (* Liste des bonus, avec couleurs et probabilite relative *)
22  let listeBonus = [
23        (tailleRaquetteUp, green, 2.);
24        (tailleRaquetteDown, rgb 150 0 0, 2.);
25        (tailleRaquetteInv, magenta, 3.);
26        (vieSupp, red, 1.);
27        (gainPoints, yellow, 2.);
28        (multiballes, white, 2.);
29        (acceleration, white, 2.);
30        (deceleration, white, 2.);
31        (ballesTraversantes, cyan, 1.);
32      ]
```

5.2 Les règles du jeu

— Le joueur dispose en début de partie de 3 vies.

— Son score est initialisé à 0. Chaque bloc rapporte 20 points, multiplié par le nombre de coups nécessaires à la destruction, et par le niveau. Ainsi, un bloc destructible en 10 coups au niveau 3 rapporte 600 points.

— Lorsqu'aucune balle n'est détectée sur le plateau, le joueur perd une vie. S'il n'a plus de vie, il perd définitivement.

— Lorsqu'aucun bloc destructible n'est sur le plateau, le joueur passe directement au niveau suivant.

— Les balles détruisent une seule couche de bloc par collision.

5.3 Évolutions et partie graphique

5.3.1 Première version

Cette capture représente la première version du jeu. La balle rebondit sur la raquette, puis sur une première brique ce qui active un bonus (qui par la suite a augmenté la taille de la raquette).

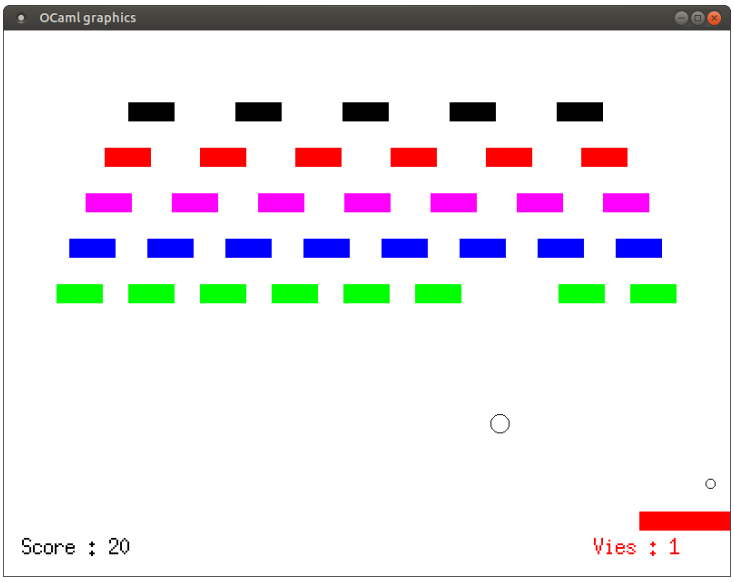


FIGURE 3 – Interface graphique représentant une partie du jeu

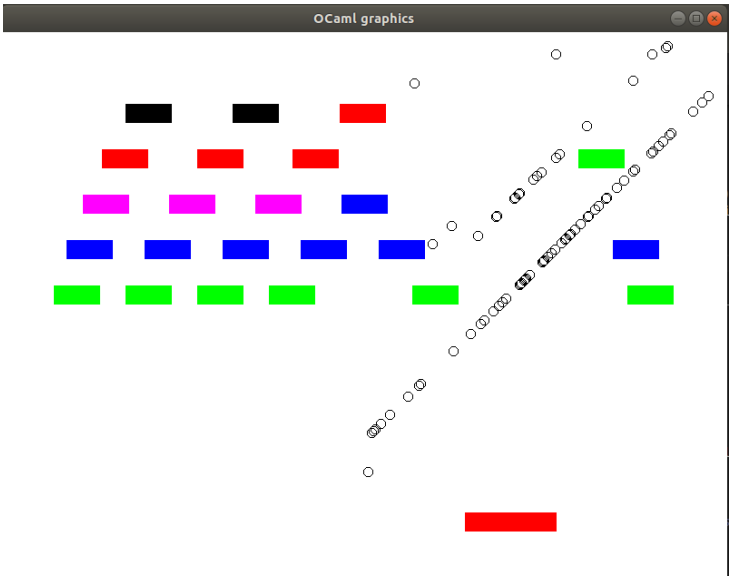


FIGURE 4 – Les premiers essais du bonus Multiballe, résultats parfois étonnants...

5.3.2 Version finale

— Les blocs blancs sont les blocs indestructibles.

— Lorsque la balle possède le pouvoir de traversée (il peut y en avoir plusieurs), elle devient rouge avec contour jaune. Ce contour n'est pas physique (il n'y a donc pas une augmentation du rayon de la balle).

— En cas d'échec, la fenêtre d'affichage affiche **Game Over**, avec le score final du joueur et le nombre de vies restantes (qui doit normalement être égal à 0).

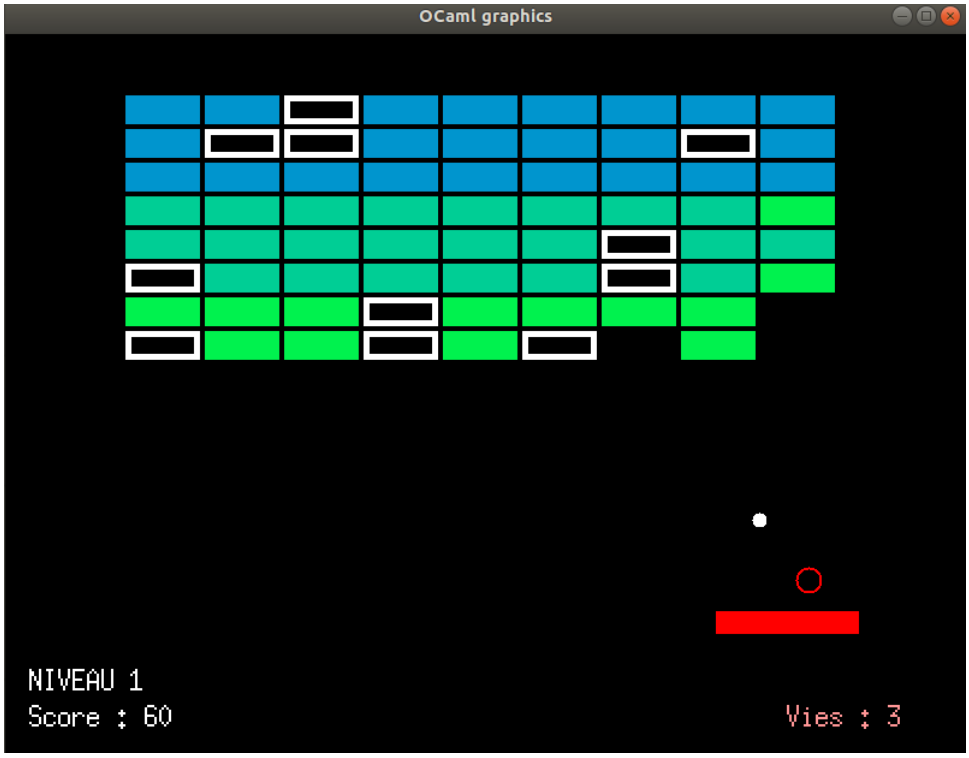


FIGURE 5 – Niveau 1 - Début de partie

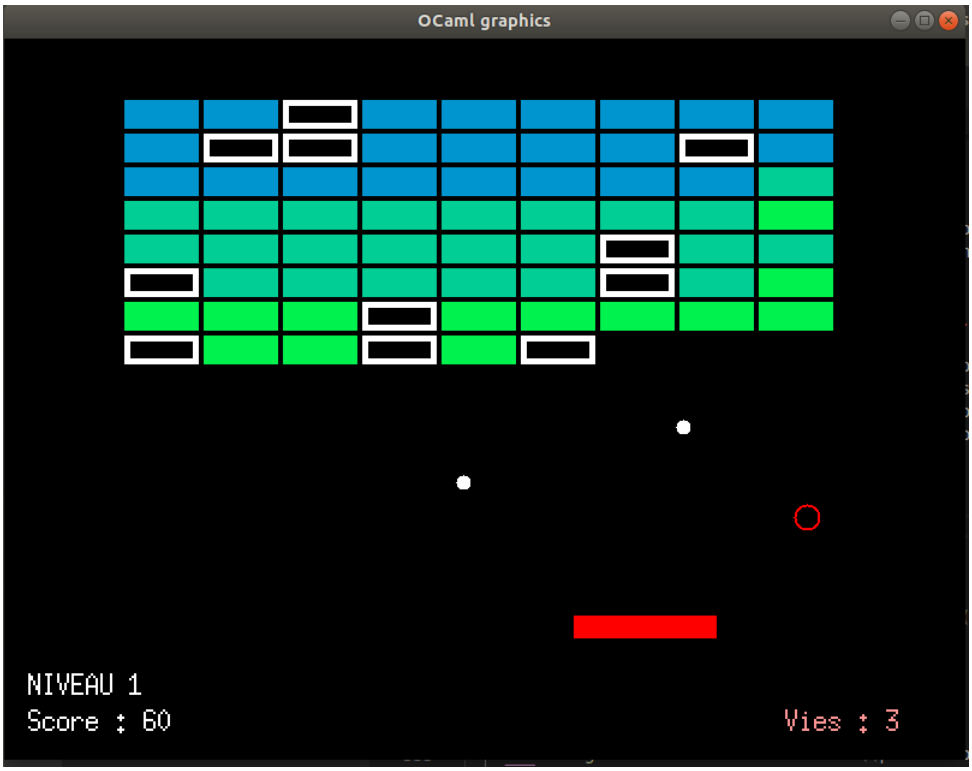


FIGURE 6 – Bonus multiballes

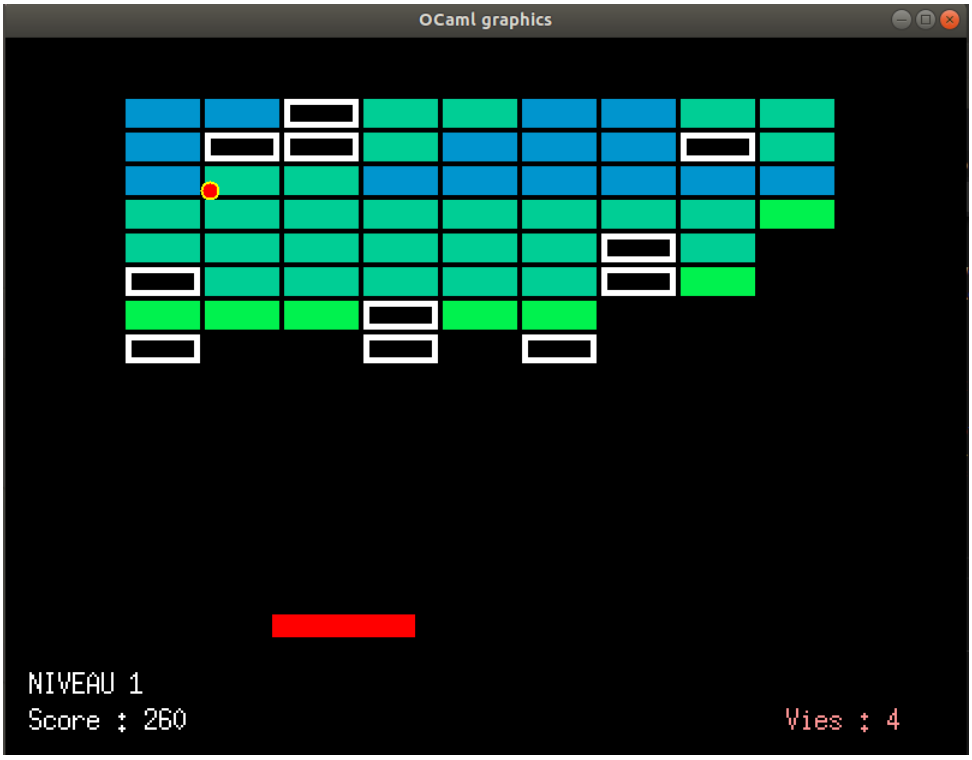


FIGURE 7 – Bonus de balle traversante.

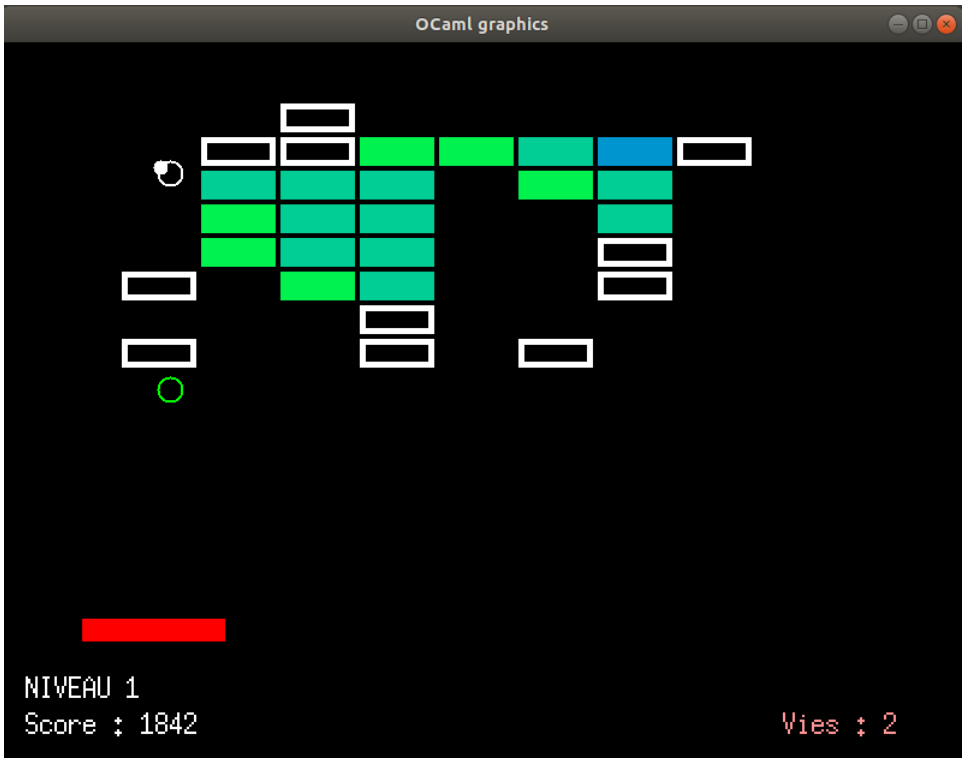


FIGURE 8 – Niveau 1 - Partie en cours

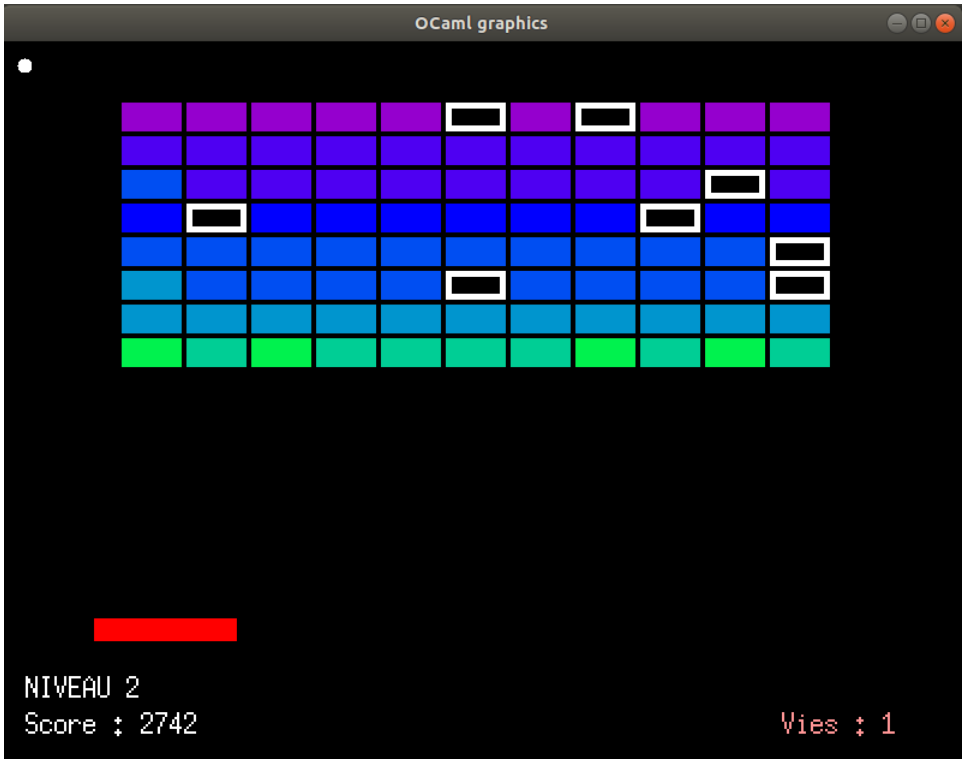


FIGURE 9 – Niveau 2 - Début de partie

FIGURE 10 – Affichage de **Game Over** en fin de partie

6 Conclusion

Pour ce projet, il a été nécessaire de découper le code en plusieurs modules afin de le rendre extensible. *CamL* étant très strict en ce qui concerne les types de variables, ne pas créer de type revenait à se compliquer la tâche : Chaque modification / implémentation de fonctionnalité nous prenait beaucoup de temps afin de remplacer des morceaux de codes dispersés un peu partout à la fois (le tuple décrivant le jeu contenu dans le flux était immense, et difficile à manipuler simplement).
Par ce passage aux modules et à la définition de types, créer une extension est bien plus simple, et structure le code de façon plus propre. Il nous a permis de nous familiariser avec l'utilisation des interfaces et des types en *CamL*, et avec la programmation fonctionnelle en général.