



Acting Shooting Star

Projet réalisé par

Boullit Mohamed Fayçal
Izekki Jalal
Moussa Guimba Mamadou
Pierre Hugo

Projet encadré par

Calandra Joséphine

Equipe 9151

Année 2019/2020

Table des matières

1	Introduction	3
2	Structure du Jeu	3
2.1	Répresentation des acteurs	3
2.2	Représentation et traitement des messages	4
2.2.1	Move	4
2.2.2	Collide	4
2.2.3	Shoot	5
3	Gestion du jeu	5
3.1	Le monde des acteurs	5
3.2	Avancement du jeu	6
3.3	L'interface graphique	6
4	Tests et documentation du projet	7
4.1	Les tests	7
4.2	Documentation	8
5	Conclusion	9
5.1	Perspective et suite du projet	9
5.2	Les apports du projet	9

1 Introduction

Le sujet de programmation informatique traité ce semestre est le modèle basé sur les acteurs (**Actor Model**). C'est un modèle mathématique qui considère un monde remplie d'acteurs qui interagissent entre eux. Dans ce modèle un acteur est une unité fondamentale de calcul ; c'est un objet qui reçoit un message et fait une sorte de calcul sur lui même. Tout acteur possède alors un état propre, envoie mais aussi reçoit des messages d'autres acteurs et met à jour son propre état.

Le jeu *Terminal Phase* codé par Christopher Lemmer Webber en langage Racket, est un exemple concret de ce type de programmation. A l'instar de ce dernier, le but de notre projet est de construire et développer une bibliothèque d'acteurs en Racket menant à l'élaboration d'un jeu qui approche un shoot'em up.

2 Structure du Jeu

Dans l'environnement du jeu se trouvent différents types d'acteurs qui interagissent entre eux en s'envoyant différents types de messages. Pour une bonne hiérarchisation une horloge de synchronisation découpe le temps en intervalle. A chaque pas de l'horloge, les acteurs reçoivent leurs messages et se mettent à jour en fonction de ce qu'ils ont reçu en attendant le prochain pas de l'horloge.

2.1 Représentation des acteurs

Un acteur est représenté par une structure de données nommée **Actor** et il est caractérisé par son nom, sa position (x,y) dans l'espace du jeu et sa durée de vie. Il faut savoir que les messages sont envoyés de manière asynchrone aux acteurs et ces derniers les traitent séquentiellement ; d'où la nécessité d'avoir une boîte de messages **mailbox** où ils peuvent stocker les différents messages. En outre, les acteurs sont de deux sortes : les vaisseaux et les missiles. Cependant on distingue quatre types d'acteurs à savoir :

- les vaisseaux alliés **ally-ship** : il existe généralement un seul vaisseau allié, qui est le vaisseau principal. Son objectif est d'attaquer et de détruire les autres vaisseaux **enemy-ship**
- les vaisseaux ennemis : c'est une masse de vaisseaux qui sort de manière aléatoire du côté ennemis et envahit le vaisseau allié.
- les missiles alliés **ally-missile** : Ce sont les acteurs créées par le vaisseau allié, qui lorsqu'ils entrent en collision avec les acteurs ennemis les détruisent et meurt également.
- les missiles ennemis **enemy-missile** : Ceux là sont créées par les vaisseaux ennemis et ont le même rôle que ceux cités précédemment ; celui de tuer ou de réduire la durée de vie du vaisseau principal.

A ceux là s'ajoutent deux autres types d'acteurs :

- les acteurs **obstacle** qui lorsqu'ils rencontrent le vaisseau allié, diminuent sa durée de vie.
- les acteurs **bonus** qui ont pour rôle d'augmenter la durée de vie du vaisseau allié.

Ainsi un acteur est représenté de la manière suivante :

```
(struct concrete-actor (name location mailbox type health))
```

Pour en créer un dans l'espace du jeu, une fonction nommée **make-actor** qui prend comme paramètre la position et le type de l'acteur, est implémentée. Elle est utilisée par (**generate-actor tick**) qui crée des acteurs ennemis, obstacles et bonus. A un pas **tick** donné, elle génère une liste des acteurs avec des ordonnées différentes aléatoires mais avec une même abscisse qui est celle de l'extrémité droite du cadre du jeu; sachant qu'ils se dirigent tous vers l'acteur principal.

2.2 Représentation et traitement des messages

Il existe différentes formes de messages en fonction de l'information que l'on souhaite transmettre aux acteurs :

- un message (**'move x y**) permet à l'acteur visé de se déplacer de (x, y) dans l'espace du jeu.
- un message (**'Collide**) qui peut prendre plusieurs formes. Il dépend des types d'acteurs qui se rencontrent.
- un message (**'shoot "-"**) qui est envoyé soit au vaisseau principal soit aux vaisseaux ennemis pour pouvoir créer des missiles.

Ces messages sont envoyés à un **actor** par la fonction **actor-send** qui retourne alors un nouvel **actor** contenant le message dans sa **mailbox**.

Vient alors la fonction **actor-update** qui met à jour les **actor** en vidant la **mailbox**. Pour ce faire, la fonction s'appelle récursivement en modifiant l'**actor** à l'aide d'autres fonctions appelées en fonction du message.

```
(actor-send actor message)
(actor-update actor)
```

2.2.1 Move

Le message **move** est géré par deux fonctions, la première :

```
(actor-move actor mvnt)
```

prend en paramètre un **actor** et le mouvement (**mvnt**) à lui faire réaliser. Ce mouvement est représenté par une liste de deux coordonnées représentant le vecteur du déplacement.

Vient alors l'autre fonction :

```
(sum-pair-list p l)
```

Cette fonction permet de sommer la liste représentant le mouvement et la position de l'**actor** qui est une **pair**.

2.2.2 Collide

Il y a plusieurs messages pour les collisions chacun correspond à un type de collision entre le vaisseau allié et un autre **actor**.

Chaque fonction permet une réaction différente en fonction du type de l'**actor** rencontré et ainsi soit détruire le vaisseau, soit lui réduire sa durée de vie.

2.2.3 Shoot

Le message de tire est géré par la fonction **actor-shoot** qui en fonction du type de l'**actor** donné en paramètre, crée alors un missile allié ou ennemie à l'avant de l'actor.

3 Gestion du jeu

3.1 Le monde des acteurs

Pour manipuler le monde des acteurs, une structure **world** est créée qui regroupe l'ensemble des acteurs dans une liste.

```
(struct world (actors))
```

Cela permet de gérer de manière globale l'ajout des acteurs dans le jeu, les déplacements des acteurs, l'envoi des différents messages mais également la gestion des différentes collisions.

- L'ajout d'une liste d'acteurs dans le monde se fait à l'aide d'une fonction

```
(world-add-actor world actors)
```

qui est de complexité en temps linéaire.

- En ce qui concerne les collisions rappelons qu'un acteur dispose d'une certaine durée de vie qui est constante et négative pour les acteurs ennemis du vaisseau allié mais positive et constante pour ce dernier. Lors d'une collision du vaisseau principal et d'un autre acteur, un message de type **collide** qui est fonction du type du second acteur, est envoyé aux deux acteurs. Par la suite, à la mis à jour des acteurs le second est détruit et l'acteur principal gagne ou perd en durée de vie. Prenons l'exemple d'une collision avec un acteur **bonus** ; dans notre cas nous considérons arbitrairement que ce dernier rapporte 3 unités de plus dans la durée de vie du vaisseau principal, contrairement à une collision avec un missile ennemi qui réduit de 2 unités sa durée de vie. Ainsi donc, afin de vérifier qu'il y a des collisions dans le monde des acteurs, une fonction

```
(world-collision world)
```

est implémentée. Elle prend récursivement un acteur du world et vérifie d'éventuelles collisions avec le reste des acteurs. S'il y a collision, elle envoie le message **collide** aux deux acteurs et passe à un autre acteur, sinon refait la même démarche sur le reste des acteurs. Ce qui donne à cette fonction une complexité quadratique $\theta(n^2)$ où n est le nombre d'acteurs dans l'espace de jeu.

- L'envoi des différents messages se fait à l'aide d'une fonction :

```
(world-send world msg type)
```

qui recherche de manière récursif dans world, les acteurs dont le type correspond à celui passé en paramètre et lui envoie le message msg à l'aide de la fonction **actor-send**. Cette fonction est de complexité en temps linéaire.

- La mise à jour du monde se fait à l'aide d'une fonction :

```
(world-update world)
```

qui utilise la fonction **actor-update** récursivement sur le world mais aussi la fonction **world-collision** ce qui lui donne d'ailleurs une complexité en temps quadratique.

Notons par ailleurs qu'il arrive un moment où certains acteurs sortent du cadre du jeu. Arrivés à l'abscisse 0, ces derniers doivent être détruits vu qu'ils sont arrivés à l'autre extrémité du cadre et qu'ils ne sont plus utiles dans la suite du jeu. La fonction (**destroy-actor actors**) le fait récursivement sur les différents acteurs. Elle vérifie également que le vaisseau principal se trouve à tout moment dans l'espace du jeu. S'il arrive qu'il sorte du cadre, ce dernier est détruit et le jeu prend fin.

3.2 Avancement du jeu

La structure world citée précédemment est placée dans une autre nommée **runtime** qui contient des messages et l'horloge globale **tick**. Elle a pour fonction d'envoyer les différents messages aux acteurs du monde **world** à chaque pas de l'horloge, de les mettre à jour et de renvoyer un nouvel état du monde.

```
(struct runtime (world messages tick))
```

Au début du jeu, le seul acteur présent est le vaisseau principal attendant l'invasion des différents vaisseaux ennemis. A chaque pas de l'horloge **tick** et avant le prochain pas :

- Le runtime ajoute dans son **world** des nouveaux acteurs, fournis par (**generate-actor tick**), à l'aide de la fonction

```
(runtime-add-actor runtime actors)
```

- le runtime par la suite reçoit une liste de messages par la fonction

```
(runtime-receive runtime messages)
```

avec **messages** de la forme (**list (msg type)**) où chaque message **msg** est de type **move** à envoyer à l'acteur de type **type**. Cela a pour but de faire avancer d'un pas les différents acteurs.

- Après avoir reçu ses messages, le runtime vide son champs messages et envoie ces derniers aux différents acteurs à l'aide de la fonction

```
(runtime-send runtime)
```

- Enfin le runtime, à l'aide de la fonction

```
(runtime-update runtime)
```

, met à jour tout les acteurs et renvoie un nouvel état du monde.

Ces étapes sont effectuées successivement à chaque pas de l'horloge globale du jeu. Toutes les fonctions citées plus haut sont linéaires à l'exception de la fonction de mise à jour qui est quadratique et de la fonction runtime-receive qui est constante.

3.3 L'interface graphique

L'affichage des acteurs ainsi que l'interface graphique est faite à l'aide de la bibliothèque **raart** qui fournit plusieurs fonctions permettant d'afficher du texte sur un terminal. Pour l'animation du jeu c'est la bibliothèque **Lux** qui est utilisée. Elle fournit un moyen efficace de créer des programmes interactifs en temps réel. Pour le






fonctionnement de la bibliothèque sur notre projet, il s'agit de créer une structure qui représente l'état globale de l'application.

```
(struct game (runtime)
 #:methods lux:gen:word)
```

La structure **game** comprend le runtime ainsi que différentes méthodes qui permettent la gestion de l'état du jeu :

- * la méthode (word-fps w) renvoie le taux de mise à jour souhaité pour le **game** en images par seconde.
- * la méthode (word-label w) renvoie le titre de la fenêtre contenant le jeu.
- * la méthode (word-event w e) qui à partir d'un état du monde et d'un événement **e** renvoie un nouvel état du monde. C'est cette méthode qui permet l'interaction entre le jeu et l'utilisateur. C'est ainsi que pour faire déplacer le vaisseau principal, l'utilisateur tape sur les touches de direction de son clavier et cette fonction envoie un message move ou shoot à l'acteur principal en fonction de la touche enfoncé sur le clavier. Rappelons par contre que seul le vaisseau principal est manipulable par l'utilisateur ; le reste des acteurs est géré par le runtime.
- * la méthode (word-output w) qui permet d'afficher n'importe quel objet graphique.
- * la méthode (word-tick w) qui permet la mise à jour des différents acteurs et renvoie un nouveau monde.
- * la méthode (word-return w) qui affiche un message en cas d'éventuelles erreurs.

A partir de la figure 1 nous pouvons visualiser la manière dont sont représentés nos différents acteurs :

-  représente l'acteur principal,
-  représente un vaisseau ennemi et fait perdre 2 unités de vie au vaisseau principal,
-  représente un missile et fait perdre également 2 unités de vie au vaisseau principal,
-  représente un acteur obstacle. Il fait perdre 5 unité de vie au vaisseau principal,
-  représente un bonus et fait gagner 3 unités de vie au vaisseau principal

4 Tests et documentation du projet

4.1 Les tests

Les tests pour ce projet ont été réalisés à l'aide de la bibliothèque **rackunit**. Il sont décomposés en 3 fichiers. Chaque fichier contient une **test-suite** :

- Un fichier **test-actor.rkt** : teste les fonctions du fichier **actor.rkt**. Les fonctions testées sont **actor-location**, **actor-send** et **actor-update**.

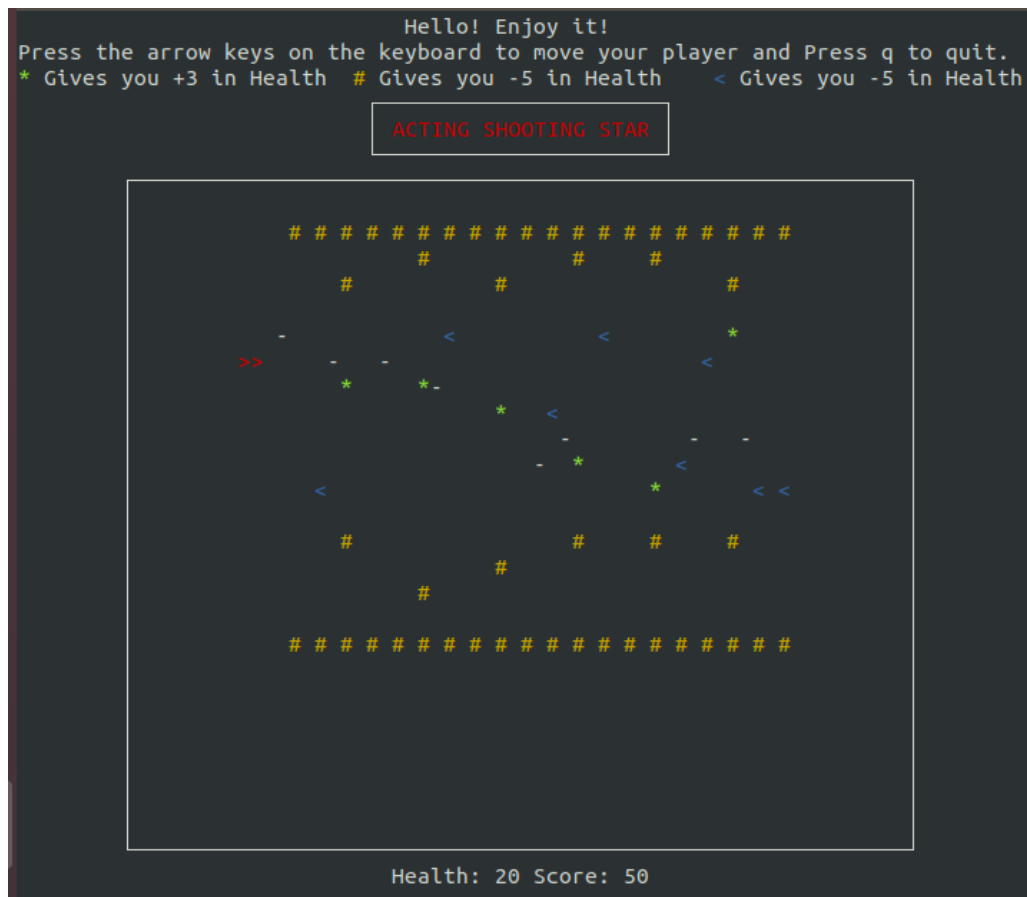


FIGURE 1 – L’interface graphique du jeu

- Un fichier `test-world.rkt` pour tester les fonctions `world-send`, `actors-collision` et `world-update` du fichier `world.rkt`.
- Un dernier fichier `test-runtime.rkt` pour tester les deux fonctions `runtime-send` et `runtime-update`.

La commande `make test` permet d’exécuter l’ensemble de ces fichiers et afficher les résultats des tests.

4.2 Documentation

La documentation de ce projet est réalisée avec **Scribble** qui est un outil de description ressemblant au langage \LaTeX . Les fichiers écrits avec cet outil en plus du fichier principal `doc.scrbl` sont les suivants :

- Un fichier `actor.scrbl` pour décrire le code du fichier `actor.rkt`.
- Un fichier `world.scrbl` contenant la documentation des fonctions et structures de `world.rkt`.
- Un fichier `runtime.scrbl` qui montre le fonctionnement du code de `runtime.rkt`.

La commande pour générer la documentation est `make doc`. Cette commande permet de générer plusieurs fichiers et les mettre dans le répertoire `doc`. Pour visualiser la documentation, il suffit d’ouvrir le fichier `doc.html`.

5 Conclusion

5.1 Perspective et suite du projet

Ce projet nous a permis de développer un jeu du style shoot'em up, dans lequel le joueur joue un vaisseau qui se défend d'une invasion ennemie, toute fois nous n'avons implémenté qu'un seul niveau de difficulté. Dans la continuité, nous pourrions donc implémenter différents niveaux de difficultés comme par exemple des vaisseaux ennemis qui tire des missiles, en augmenter le nombre ou bien des vaisseaux ennemis qui lorsqu'ils sont détruits en libèrent d'autres.

5.2 Les apports du projet

Ce projet nous a permis de réaliser un projet en Racket qui est un langage fonctionnel or les autres projets que nous avons réalisé étaient dans des langages impératifs, or la programmation fonctionnelle est une autre façon de voir la programmation.

De plus, ce projet nous à également permis de découvrir les modules racket **raart** et **lux**.