Projet
S6

Acting Shooting Star

Boudroua Imad **Zghari Hicham**

Nguyen Olivier **Gauchet Augustin**

Equipe 9145

Encadrants : M.Renault et M.Rollet

Table des matières

1 Introduction et analyse du projet **3**
1.1 Contexte . . . 3
1.2 Problématique . . . 3
1.3 Cadre de travail . . . 3
1.4 Bibliothèques utilisées . . . 3

2 Fonctionnalités développées **3**
2.1 Actor . . . 3
2.1.1 Structure et mise à jour des acteurs . . . 3
2.1.2 Gestion des collisions . . . 4
2.2 World . . . 4
2.2.1 Structures et fonctions . . . 4
2.2.2 Retour dans le temps . . . 4
2.3 Runtime . . . 5
2.4 Documentation . . . 5

3 Tests employés et contrats **5**

4 Conclusion **6**

1 Introduction et analyse du projet

1.1 Contexte

Dans le cadre du second projet de programmation en première année d'informatique à l'ENSEIRB-MATMECA, nous avons eu à prendre pour exemple le jeu **Acting Shooting Star**, en langage **Racket**. Ce rapport va permettre d'expliquer les démarches que notre groupe a effectuées pour mener ce projet.

1.2 Problématique

Le but de ce projet est d'implémenter le modèle de l'acteur : un monde rempli d'acteurs intervenant par le biais de messages. L'enjeu est alors d'assurer l'interaction entre les acteurs, c'est à dire implémenter leur comportement selon la réception de messages, leur permettre d'en envoyer aux autres et de pouvoir en créer de nouveaux.

1.3 Cadre de travail

À cause de la pandémie de Coronavirus, nous avons dû travailler chez nous, certains en ssh et d'autres en machines virtuelles. Les différences de configuration et le manque de maîtrise du ssh ont notamment été un frein au développement du projet. Le code a dû également subir de nombreuses restructurations pour plus de clarté.

1.4 Bibliothèques utilisées

1. Raart : fournit un modèle algébrique d'ASCII qui peut être utilisé pour l'art, l'interface utilisateur et les diagrammes. Dans notre projet, il a été utilisé essentiellement pour mettre le jeu dans un cadre déterminé par quatre variables (x_1,x_2,y_1,y_2) et dans l'affichage des joueurs avec les différents symboles et couleurs.
2. Lux : fournit un moyen efficace pour la création des programmes qui interagissent en temps réel avec l'utilisateur.
3. Rackunit : est une bibliothèque facilitant la mise en place de tests afin de s'assurer du bon fonctionnement du code.

2 Fonctionnalités développées

Afin d'assurer les interactions entre les acteurs, nous avons non seulement implémenté la structure de données **actor**, mais aussi d'autres structures qui régissent leur fonctionnement. Les acteurs sont regroupés dans une structure **world**, et la structure **runtime** permet de les manipuler en leur envoyant des messages à chaque **tick** de jeu.

2.1 Actor

2.1.1 Structure et mise à jour des acteurs

La structure **actor** est au coeur du projet, elle permet la création du joueur, d'ennemis ou encore de projectiles. Elle est composée de quatre champs :

— **position** est une liste de deux éléments qui contient les coordonnées x et y
— **mailbox** est une liste de messages
— **name** contient la représentation visuelle en **raart** de l'acteur
— **category** est le type de l'acteur (**player**, **enemy**, **projectile**)

Pour faire fonctionner ce projet, il est nécessaire de pouvoir mettre à jour les coordonnées d'un acteur ou d'en créer de nouveaux en accord avec sa **mailbox**. Pour ce faire nous avons implémenté la fonction `actor-update`.

Cette fonction parcourt récursivement la **mailbox** en effectuant les instructions situées dans les messages au fur et à mesure de son exécution. Sa complexité en temps est donc linéaire en fonction de la longueur de sa **mailbox**. De plus, cette fonction est récursive terminale, ce qui permet à l'interpréteur de l'optimiser.

2.1.2 Gestion des collisions

Afin de détecter les collisions entre les acteurs, nous avons comparé positions relatives des acteurs à chaque tick de l'horloge. Pour cela, la fonction `colliding?` a été implémentée pour vérifier si deux acteurs sont entrés en collision. La fonction `collisions?` nous permet de savoir s'il y a eu des collisions entre les différents acteurs. Elle est de complexité linéaire en fonction du nombre d'acteurs.

2.2 World

2.2.1 Structures et fonctions

La structure **world** est constituée d'une liste regroupant tous les acteurs. Elle assure l'interdépendance entre les acteurs et permet leur actualisation et leur interaction par la fonction `update-word`. Le monde agit sur les acteurs mais les acteurs peuvent aussi interagir sur celui-ci par le biais de fonctions comme `send-to-world` qui envoie un message à tous les acteurs du **world**. Ainsi, le monde permet de se détacher d'une vision individualiste d'un acteur qui évolue et s'actualise seul et d'arriver à une interdépendance entre les différentes entités qui évoluent. Les fonctions les plus importantes sont celles ci-dessous :

— `world-update` est au coeur du projet puisqu'elle permet la mise à jour de tous les acteurs du **world**. Pour ce faire, elle appelle la fonction `actor-update` pour chaque acteur de la liste, ce qui signifie une complexité en temps linéaire en fonction du produit du nombre d'acteurs présents par la longueur de **mailbox**.
— `generate` est une fonction qui permet de générer aléatoirement de nouveaux acteurs pour le jeu et de les ajouter au **world**
— `game` est la fonction qui régit chaque **tick** du jeu. Elle sauvegarde le **world** (cf partie suivante), élimine les acteurs qui sont entrés en collision et génère les ennemis.

2.2.2 Retour dans le temps

Nous avons mis en place une fonction pour revenir en arrière dans notre jeu. Pour ce faire nous avons utilisé une liste **latest-worlds** déclarée en variable globale dans notre fichier qui contient comme son nom l'indique les dernières structures **world** actualisées. Elles sont classées de la plus récente à la plus ancienne.

À chaque tour de boucle du jeu, une copie du **world** actuel est ajoutée dans **latest-world** par la fonction `save-world`. Cette fonction ajoute en tête de liste le world à sauvegarder et dans le cas où la liste est de taille supérieure à 20, elle supprime le dernier élément de la liste quand elle ajoute un élément. Effacer le dernier élément de la liste consiste à inverser la liste, prendre son `cdr` et l'inverser encore une fois, `save-world` est alors de complexité quadratique par rapport à la taille de la liste dans le pire des cas. Néanmoins la liste est de taille 20 au maximum.

Pour revenir en arrière de n ticks il suffit de remplacer le **world** actuel par le n-ième élément de **latest-worlds**, c'est ce que fait la fonction `world-travel` qui est ainsi récursive terminale de complexité linéaire par rapport à la n (nombre de ticks à revenir en arrière). De manière pratique (principalement pour débugguer), nous avons permis au joueur de revenir de 15 ticks en arrière lorsqu'il appuie sur "p".

L'implémentation d'une liste en variable globale aurait pu être évitée en intégrant une liste de **world** dans la structure représentant la boucle de jeu. Nous y avons pensé trop tard et si nous avions eu plus de temps, nous aurions opéré le changement.

2.3 Runtime

runtime est une structure qui orchestre le jeu entre les acteurs. D'une part, elle contrôle l'écoulement du temps et la durée du jeu et d'autre part elle organise la transmission des messages et les traduit en événements. Son utilisation avec la bibliothèque Lux a permis d'interagir avec les entrées du clavier. Cela s'avère utile en ce qui concerne le contrôle du jeu par un utilisateur. L'implémentation de cette structure s'articule autour de 4 éléments principaux :

1. **world-fps** initialise le nombre d'images par seconde que le jeu est censé s'afficher.
2. **world-event** gère les événements qui se produisent et les interactions avec les périphériques extérieurs.
3. **world-output** permet d'afficher les éléments de la bibliothèque raart sur le terminal.
4. **world-tick** s'occupe de mettre à jour la nouvelle **runtime** après chaque unité de temps.

2.4 Documentation

Nous avons écrit notre documentation avec Scribble. Elle contient l'explication détaillée des structures et fonctions utilisées. L'utilisation des fonctions `defproc` et `defstruct` permet d'établir rapidement une documentation structurée.

3 Tests employés et contrats

Les tests consistent principalement à vérifier que les fonctions renvoient bien ce qu'elles sont sensées renvoyer, ils nous ont permis de nous donner de la confiance dans notre code. Par exemple, pour tester `actor-update`, nous avons appliqué la fonction à divers acteurs possédant une **mailbox** différente. Nous avons ensuite vérifié que les résultats obtenus coïncidaient avec ceux escomptés. Certains tests ont néanmoins nécessité un affichage pour s'assurer que leur implémentation n'était pas défectueuse, notamment `world-travel` et la fonction qui génère les projectiles.
Un exemple de test consiste à contrôler les renvois de la fonction `actor-update` en se mettant en condition de chaque cas du **cond**. On vérifie ensuite que les fonctions renvoient les bons résultats attendus pour chaque test portant sur ces conditions.

Dans l'optique de vérifier que nos fonctions se comportent comme prévu, nous avons également fait des contrats. Ces contrats nous assurent que les fonctions sont appelées avec les bons types de paramètres, et qu'elles renvoient le bon type de données. Ils permettent ainsi de rendre notre code plus sûr.

4 Conclusion

Le jeu codé est un shooter simple où le but est de survivre tout en éliminant le plus d'ennemis possibles. Le joueur possède une unique vie et meurt lorsqu'il heurte un mur ou un ennemi.

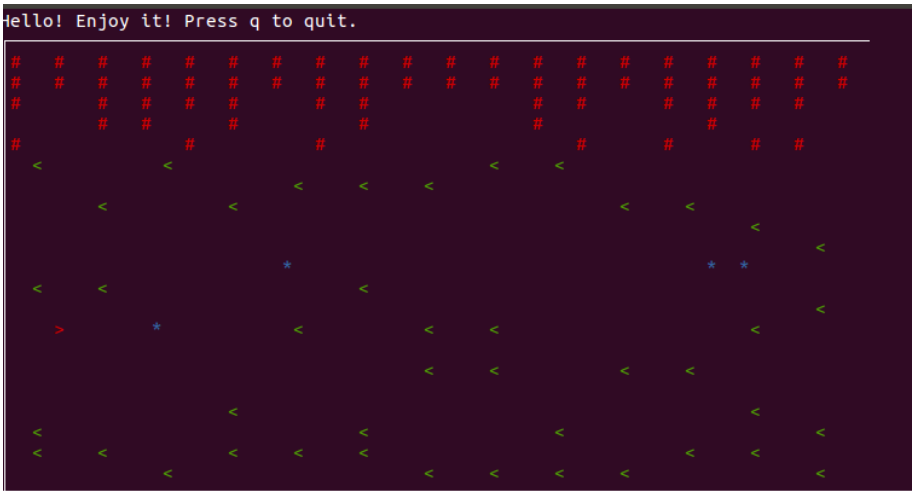

FIGURE 1 – Capture d'écran du jeu

Le projet de programmation fonctionnelle nous a ainsi permis de nous détacher de la programmation impérative. En effet, nous avons découvert un nouveau modèle de programmation qui consiste à n'utiliser que le résultat d'application des fonctions ce qui assure la transparence référentielle. Nous avons aussi intégré la notion du typage dynamique qui diffère du typage statique auquel nous étions habitués.