Acting Shooting Star

Rapport de projet

Projet de programmation - S6

– I1 - ENSEIRB-MATMECA –

Encadrants : M. David RENAULT - M. Sylvain Lombardy

Par Ismail ELOMARI ALAOUI - Nicolas LE QUEC - Vincent RIDACKER -

Alexandre GISSAUD

19 mai 2020

Table des matières

Introduction

Contexte

Ce rapport a pour objectif de décrire l'ensemble du processus de réalisation d'un projet de programmation basé sur le langage Scheme (*Drracket*). Ce projet est conçu par Mr. David RENAULT, le directeur des projets à l'ENSEIRB-MATMECA, pour aider les élèves en première année d'informatique à avoir des compétences basiques voire avancées sur l'utilisation du scheme. Le projet consiste en l'implémentation d'un jeu appelé ACTING SHOOTING STAR, en racket. Nous sommes guidés par les questions produites dans le projet, afin de premièrement bien structurer le projet, et ensuite implémenter des acteurs puis un monde. Enfin, nous devons animer le tout à l'aide de deux importantes bibliothèques : *Luxx* et *Raart*.

Description du sujet

Ce projet nous amène dans un monde virtuel dans lequel nous sommes un joueur (un acteur), qui essaie d'éviter certes l'embouteillage causé par d'autres acteurs mais aussi leurs missiles. Cet acteur n'est pas non plus sans espoir, il peut se protéger en tirant ses propres missiles, face aux ennemis malveillants. Ce projet est inspiré par le travail de C. L. Webber *Terminal Phase*, dans lequel un modèle basé sur les acteurs est utilisé.
L'implémentation de ce modèle provoque plusieurs problématiques :

- Comment un acteur peut-il être proprement structuré ?
- Comment ce monde virtuel que nous souhaitons codé, peut-il lui-même être structuré ?
- Comment les acteurs communiquent-ils entre eux ?
- Et, comment peut-on joliment animer ce monde saturé par des acteurs qui entre-échangent des messages ?

Ce sont de façon relativement succincte, les consignes principales à résoudre et à respecter pendant notre projet.

1 Méthode de travail

1.1 Répartition des tâches

Tout d'abord, notons que ce projet a été du début à la fin un travail d'équipe complet, dans lequel nous avons appris tout du long à beaucoup échanger et à travailler ensemble, tout en se répartissant les tâches quand cela était nécessaire. Nous pensons que tout ce qui peut fracasser une équipe de programmeurs est la mal-structuration du projet. Nous pensons avoir fait de notre mieux pour éviter cette catastrophe.
Pour s'organiser, nous nous rejoignions en vocal sur Discord généralement. Lorsque nous sommes d'accord sur les tâches à réaliser et leur répartition, nous commençons à coder. Dès qu'une tâche est finie ou que quelqu'un a besoin d'aide, on propose de nouvelles fonctionnalités à implémenter ou l'on aide quelqu'un qui aurait besoin de soutien. Nous nous connectons en vocal chaque séance pour coder ensemble et être disponible directement si une question se posait mais aussi pour combattre la solitude du programmeur.
Le fichier *StructureSpecs.txt* a été rédigé en raison du nombre de structures particulières et comme aide au développement. Il a pour objectif de rappeler de manière conçise les structures utilisées dans le projets(par exemple des messages) et de fournir des explications quant au rôle de chacune. Il a été également utilisé comme carnet de notes pour le futur développement du projet et comme calque pour la déclaration des structures en fournissant des exemples. Ce fichier est adressé particulièrement aux membres du projet.

1.2 Thor Project

Nous avons utilisé la Forge (le serveur $ThorProject$) de l'ENSEIRB-MATMECA pour gérer les différentes versions de notre projet. L'évolution du projet visible sur cette Forge peut mettre en évidence nos progrès en tant que débutants programmeurs sur Scheme.
Les codes sources, évidemment présents sur la forge, ont été codés à l'aide de l'application *Drracket* v7.6. Nous vous prions de vérifier votre version avant d'exécuter le jeu.
Nous avons utilisé comme il nous l'avait été demandé LaTeX pour rédiger ce rapport. Afin de simplifier le travail simultané sur l'écriture de ce rapport, nous avons utilisé $Overleaf$, un éditeur LaTeX en ligne.

1.3 *Makefile*

Nous avons utilisé l'outil d'aide à la compilation *make*, en réalisant un $Makefile$. L'exécution du jeu est très simple. Il suffit d'écrire la commande suivante : *make*. Une fenêtre de jeu apparaîtra dans le terminal sur laquelle vous pourrez jouer instinctivement (*ZQSD* pour se déplacer, *ESPACE* pour tirer, A et E pour se déplacer dans le temps, P pour mettre le jeu en pause et T pour quitter le jeu). Il est possible d'ajouter l'option *-f <nb>* pour choisir le nombre de fps du jeu.
Afin d'approuver la validité de nos implémentations et de nos fonctions, plusieurs tests ont été programmés. Pour les exécuter, écrivez la commande suivante : *make test*.
Enfin, la commande *make doc* permet d'afficher la documentation de la bibliothèque dans votre navigateur. Dans

celle-ci est présent les descriptions des fonctions utiles à l'utilisateur pour concevoir son jeu ainsi que des explications pour comprendre le fonctionnement de la bibliothèque.

1.4 Architecture du répertoire *GIT*

Notre dépôt contient plusieurs répertoires : "*src*" qui contient tous les codes sources utilisés pour passer les tests et créer ce magnifique monde virtuel. Et le répertoire "*doc*" qui contient la documentation du projet. Cette documentation est générée à l'aide du fichier *actors.scrbl*, présent -espérons- sur notre dépôt. La méthode de production des fichiers de documentation est expliquée dans la sous-section précédente. Enfin, le répertoire "*report*" contient ce rapport en version *pdf* et en LaTeX(code source).

2 Structuration du projet

2.1 *Actor*

Dans le cadre de ce projet, un message est une liste constituée d'un *tag* (un symbole), d'une fonction ainsi que d'un ensemble quelconque de paramètres (par ex. (list 'player move 1 0)). Un acteur est simplement une structure de données possédant un état propre, et capable de recevoir et d'émettre des messages. En fonction des messages qu'il reçoit, il peut choisir de mettre à jour son état, d'émettre de nouveaux messages, voire même de créer de nouveaux acteurs ou de mourir de sa plus belle mort. La structure principale d'un *actor* est donc très claire. Mais, d'autres champs ont été ajoutés pour mieux s'adapter à certaines situations. Nous avons pensé implémenter des événements particuliers qui rendront le jeu plus actif et plus aimable. Par exemple, un évènement où tous les acteurs ennemis tirent à la fois, ou un autre où la vitesse des murs fluctuent pendant un moment, mais également l'implémentation de quelques compétences pour le joueur. *(Petit Scénario : Après avoir tué 10 ennemis, le joueur obtient une compétence qui lui permet d'être invincible pour quelques instants. Le joueur peut manuellement l'utiliser en cliquant sur 'x' par exemple*). Ou pourquoi pas un combat avec un *boss*. Ces réflexions nous ont mené à ajouter un champ *tag* à l'acteur. Ce champ lui offre une identité (*wall*, *missible*, *player*, ...) . Sa position précédente *prev-pos* est nécessaire pour l'implémentation de certains messages (ex. *collide*). Le champ *msg-next-tick* sert à stocker les messages qui seront envoyés à l'acteur au prochain *tick*. Il est notamment utilisé lorsque les acteurs veulent s'envoyer des messages à eux-même (pour ne pas générer de boucle infinie). L'attribut *collide* est une fonction qui gère le résultat d'une collision de l'acteur avec un autre. Elle ne détecte pas les collisions mais uniquement leurs conséquences sur l'acteur. Le champ *attributes* est un attribut particulier de la structure. Il ne possède aucune spécification de format et est par conséquent, complètement libre en terme de structure. Il est libre d'utilisation par son acteur. Ce champ peut être utilisé par exemple pour stocker les points de vie du joueur ou les dégâts d'un missile. (cf. Figure 1)

```
(struct actor (pos prev-pos mailbox msg-next-tick tag collide sprite attributes))
```

FIGURE 1 – Structure *actor*

2.2 Messages

2.2.1 Description

Les messages concernés ici sont ceux destinés aux acteurs. Ils sont envoyés dans leur *mailbox* pour être lus et interprétés. Les messages sont une liste dont le premier paramètre est le *tag* du destinataire, le second une fonction d'un format spécifique (cf 2.2.2) et vient ensuite un nombre quelconque d'arguments pour la fonction. Ce format a été choisi pour donner à ces fonctions le plus large champ d'action possible. La fonction s'appliquera à l'acteur ayant reçu le message. De ce fait dans le message, le paramètre acteur ne sera pas spécifié pour la fonction.
Par exemple, imaginons une fonction *move* prenant en paramètre un acteur et deux nombres x et y. Le message envoyé à un acteur avec le *tag player* pour le déplacer sera : (list 'player move 1 2).

Plus généralement on obtient la spécification suivante pour les messages :

```
(list tag fonction-message arg1 arg2 ... argN)
```

FIGURE 2 – Structure d'un *message*

2.2.2 Fonctions dans les messages

Les fonctions contenues dans les messages ont une importance particulière dans ce projet. Elles seront appelées *fonction d'actions* dans la suite de ce rapport.
Ces fonctions prennent nécessairement un acteur en premier paramètre et renvoie en sortie une paire dont le premier élément est une liste d'acteurs et le second une liste de messages. Elles peuvent prendre autant de paramètres supplémentaires qu'elles le souhaitent tant que le premier est un acteur.
Ces fonctions déterminent une transformation appliquée à un acteur. La liste d'acteurs en sortie est la liste des acteurs toujours "en vie" et/ou créés après application de cette fonction. Et la liste des messages en sortie sera redistribuée aux acteurs concernés suivant leur *tag* (ce sont les mondes, structure *world*, qui se chargeront de cela).
Il faut noter que ces fonctions interviennent au niveau de l'acteur. Le résultat de cette fonction ne détermine pas l'ensemble de tous les acteurs toujours en vie dans le jeu mais uniquement l'ensemble des acteurs toujours en vie après l'application de cette fonction sur l'acteur concerné.

```
(define (kill act) (cons '() (list (list 'bullet kill))))
```

Ainsi la fonction ci-dessus appliquée à un acteur ne renverra pas cet acteur le tuant et renverra un seul message destiné aux acteurs avec le tag *bullet*.

2.2.3 Exemple d'application

Les fonctions d'actions sont très libres dans leurs applications grâce au format définie pour elles. Elles ne peuvent que créer des acteurs et des messages en plus d'accéder à l'acteur qu'elles prennent en paramètre. Cependant, cela suffit à obtenir des comportements plutôt intéressants. En exploitant la capacité des fonctions de messages à envoyer un message au prochain tick à l'acteur qui les appelle, on peut créer des fonctions qui se déroule sur plusieurs *ticks*. On peut notamment citer la fonction *tick-counter* qui permet d'appliquer une fonction de message avec ses paramètres après un certain nombre de *ticks* passés. Pour ce faire, le message garde un compteur de *tick* comme paramètre et se renvoie lui-même à l'acteur au prochain *tick* en incrémentant le paramètre comptant les *ticks* (un acteur peu s'envoyer un message à lui-même grâce à la fonction *actor-send* prenant un acteur et un message comme argument).
La fonction *tick-counter* utilise une structure *event* qui contient la fonction à appeler, à partir de combien de *ticks* et avec quels arguments.
En utilisant ce genre de fonction, on peut créer des actions continues sur le temps. Par exemple, le jeu contient un type d'ennemi "UFO" qui suit une succession de mouvement : il se déplace horizontalement, tire puis se déplace

verticalement.

La fonction *tick-counter* n'est pas la plus efficace pour compter les *ticks* (car on ajoute un compteur pour chaque message de contenu *tick-counter*) et fonctionne plutôt comme un compte à rebours. Elle permet de créer des niveaux scénarisés, cependant pour son bon fonctionnement, il faudrait déterminer les actions devant s'effectuer à chaque tick. Ce qui n'est pas le plus pratique.

En utilisant le principe de *tick-counter*, on peut alors créer des fonctions comme *repeat-call-delayed* qui permettent d'élaborer des messages qui rappellent des fonctions indéfiniment suivant une période définie. C'est en utilisant ce genre de fonction que le projet contient des générateurs automatiques d'ennemis.

2.3 *World* et *Runtime*

Rappelons que l'objectif de ce projet consiste à mettre en place une bibliothèque d'acteurs afin de construire un jeu ressemblant à un *shoot'em up*. Pour simplifier par rapport au modèle de programmation cité ci-dessus, nous allons considérer que les acteurs se mettent à jour de manière synchronisés. Une horloge globale découpe le temps en intervalles (*tick* en anglais). Au début d'un intervalle de temps, les acteurs n'ont aucun message en attente. Ils reçoivent un ensemble de messages pendant cet intervalle (*move*, *collide* ...) sans autre modification. Ces fonctions peuvent leur être envoyées par d'autres acteurs, par eux-même, ou même par le *runtime* (qui sera détaillé juste après). À la fin de l'intervalle de temps, ils vident leur boîte de message et se mettent à jour en fonction de ce qu'ils ont reçu. Tout ce processus d'échange et d'interprétation des messages est géré par les *worlds* mais la mécanique de l'horloge globale est réalisée à l'aide d'une structure de donnée globale, le *runtime*, organisant la mise à jour simultanée de tous les acteurs. Les messages sont alors considérés comme reçus simultanément pendant cet intervalle. L'implémentation d'un monde (*world*) et d'une horloge ou un contrôleur de temps (*runtime*) était donc indispensable.

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

FIGURE 3 – Structure *world*

Un monde contient 2 champs (cf. Figure 3) :

- *actors* est une liste de listes. En effet, les acteurs sont regroupés dans des listes selon leur *tag*. Le premier argument de chaque sous-liste commence par le *tag* des acteurs de celle-ci. Exemple de liste d'acteurs : (list (list player a0) (list enemy a1 a2)). Ceci permet d'envoyer plus facilement des messages à certains groupes d'acteurs.
- *mailbox* est liste contenant les messages qui seront envoyés aux acteurs au prochain *tick*.

```
(struct runtime (world-count max-worlds worlds functions))
```

FIGURE 4 – Structure *runtime*

Dans ce monde virtuel, des multi-dimensions sont tout à fait envisageables. Elles peuvent être modélisées comme des phases ou des stages du jeu. Le joueur passe d'un monde à un autre, s'il reste en vie à la fin d'une phase particulière.

De plus afin de permettre au joueur de revenir dans le temps pour pouvoir rejouer des séquences qu'il pense avoir échoué ou qu'il voudrait optimiser, les états des mondes sont sauvegardés à chaque *tick* dans le *runtime*. Remonter dans le temps revient alors à se balader dans cet liste de mondes. (cf. Figure 4) :

- *world-count* est le nombre de monde courant dans le *runtime*.
- *max-worlds* est le nombre maximal de mondes sauvegardés. Cela définit en quelque sorte la mémoire du *runtime*.
- *worlds* est une liste contenant tous les mondes sauvegardés. Le premier monde de la liste est le monde courant.
- *functions* est une liste contenant les fonctions primordiales au fonctionnement de la boucle de jeu. (ex. *world-update*, *world-empty-mailbox*, ...)

La figure 5 résume le déroulement d'un *tick*.

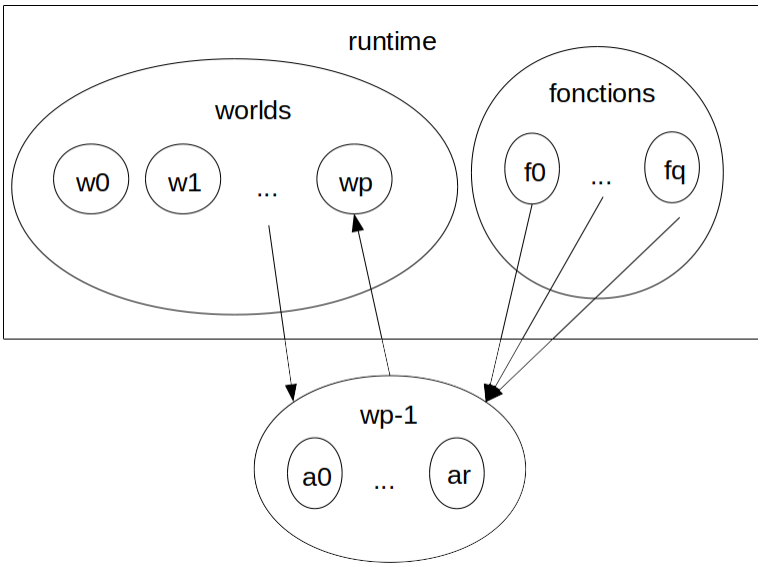

FIGURE 5 – Schéma de la boucle de jeu

Pour commencer, on récupère le monde courant et on lui applique toutes les fonctions du *runtime*. Dans celles-ci est notamment présent la fonction *world-empty-mailbox* vidant la messagerie du monde courant et envoyant les messages aux acteurs correspondant (en accordance avec le *tag* des messages). Chaque acteur va ensuite effectuer les actions qui lui ont été envoyées et renvoyer une liste d'acteurs (créés et/ou toujours en vie) ainsi qu'une liste de messages qui sera par la suite transférée dans la *mailbox* du monde pour le prochain *tick*. Une fois que tous les acteurs ont effectué leurs actions, l'état actuel du monde est affiché grâce aux bibliothèques *raart et lux*. Pour finir ce nouveau monde ainsi obtenu est ajouté à la liste des mondes sauvegardés. Lorsque l'on arrive en limite de mémoire, le monde le plus ancien est supprimé.

2.4 Optique de l'architecture

Ce projet consiste en l'implémentation d'un jeu en suivant un modèle basé sur des acteurs échangeant des messages. Pour ce faire, nous avons essayé de séparer les rôles et les responsabilités de chaque structure. Les structures agissent à différentes échelles de libertés :

- Les acteurs ne peuvent accéder qu'à eux-mêmes. Ils ne peuvent pas modifier le *world* ou le *runtime*. Ils peuvent échanger des messages entre acteurs par l'intermédiaire du *world*.
- Les mondes (structure *world*) englobent les acteurs. Ils sont chargés de délimiter un périmètre dans lequel des acteurs sont libres d'agir (comme dans un vrai monde). Ils ont aussi la charge de jouer le rôle de la poste et de retransmettre les messages selon les *tags* (les acteurs étant confinés, ils ne peuvent pas jouer le rôle du facteur).
- La structure *runtime* fait office de superviseur des mondes. Elle assure le bon déroulement des mondes en appliquant des routines de fonctions (notamment *world-update*) et fait l'interface avec l'utilisateur de l'application. Cette structure se charge également par conséquent des "retours dans le passé" à l'aide d'une liste de monde.

Pour faire tourner un monde, il faut des acteurs, mais il faut également que ces acteurs disposent d'actions. Nous avons ainsi décidé de maximiser la liberté de leurs actions comme dans un vrai monde. Ces actions sont modélisées à travers les messages et les fonctions d'actions. Les fonctions d'actions décrivent quelles conséquences a une action sur un acteur. Ce sont ces fonctions qui s'occupent de la plupart des interactions du jeu tandis que les structures *world* et *runtime* ne s'occupent que de maintenir les routines.

3 Fonctions particulières

3.1 actor-update-list

Cette fonction prend un acteur en entrée puis lit, applique et enlève tous ses messages pour enfin renvoyer le résultat dans le même format que les fonctions d'action.

Cette fonction a été particulièrement difficile à programmer en respectant le format des fonctions d'action et l'aspect fonctionnel du projet. En effet, puisque les fonctions de messages renvoient simplement une liste d'acteurs, il n'est pas possible d'identifier lequel est l'acteur initialement passé en paramètres sans appliquer de contraintes sur les fonctions de messages (comme par exemple ne pas changer le *tag* de l'acteur en entrée pour effectuer un test d'égalité à la sortie). On ne peut pas non plus identifier l'acteur par sa position dans la liste sans créer un acteur "mort" spécifique servant à l'identification.

De ce fait, la fonction *actor-update-list* maintient une liste d'acteurs en vie, et met à jour tous les acteurs de cette liste qui ont des messages sans distinction. Pour ce faire, elle ajoute la liste des acteurs résultant de l'application d'un message sur l'acteur propriétaire à la liste des acteurs en vie et répète le processus tant qu'il reste des messages. Par conséquent, toute fonction d'action qui créerait un acteur ayant un message dans sa *mailbox* verra cet acteur mis à jour dans le même *tick* que sa création. Pour pouvoir, envoyer des messages qui seront interprétés au prochain *tick*, il a donc été nécessaire de mettre en place l'attribut *msg-next-tick* dans *actor* (servant en quelque sorte de

mailbox tampon).

3.2 Animation et Affichage

Pour l'affichage, le projet se base sur la bibliothèque Raart. Tous les acteurs disposent d'un attribut *sprite* qui doit être affichable dans un terminal ; il doit être du type *raart*. C'est la structure *runtime* qui s'occupe de cela, les acteurs décrivent seulement comment ils apparaissent et le *runtime* les affiche. Pour cela, nous avons choisi de simplement afficher tous les acteurs du monde "courant" à l'aide de la fonction *place-at* fournie par *Raart*. C'est le rôle de la fonction *(define (display-world world))* qui produit l'objet Raart qui sera affiché dans le terminal en regroupant tous les acteurs du *world* passé en paramètre.

3.3 Collisions

3.3.1 Détection de collisions

À chaque tick du jeu, avant d'appliquer les messages à chaque acteurs, on applique la fonction `collisions` au *world*. Cette fonction parcours la liste des acteurs pour chaque acteur et vérifie si leur trajectoire se croisent. Pour cela la structure *actor* a un champ "position" qui contient sa position actuelle, et "position précédente" qui contient sa position au tick précédent. Si leur trajectoire se croisent, le message `collide` est envoyé aux deux acteurs.

3.3.2 Attribut collide dans la structure *actor*

Cet attribut est une fonction du format des fonctions de messages (cf 2.2.2). Ce choix a été fait pour permettre aux acteurs de gérer individuellement la conséquence d'une collision par eux-même. Lorsque le jeu détecte une collision entre deux acteurs, il envoie un message de collision aux deux acteurs concernés qui réagiront en conséquence. On peut ainsi moduler un comportement différent selon chaque acteur.

4 Boucle principale

Nous possédons maintenant tous les outils nécessaires à l'implémentation de la boucle de jeu dans le *src/main.rkt*. La boucle de jeu utilise les méthodes fournies par la bibliothèque Lux dans la structure *runtime*.
A chaque *tick*, la boucle appelle simplement la fonction *runtime-apply-func* qui applique un ensemble de fonction à la structure *world* courante. Cet ensemble de fonctions est stocké dans la structure *runtime* et peut être modifié selon les besoins, mais certaines fonctions comme *world-update* et *world-empty-mailbox* sont ajoutées de base (car elles sont indispensables au bon fonctionnement du jeu).
La gestion des contrôles utilise également Lux, lorsque l'on appuie sur une touche, la structure *runtime* envoie un message correspondant aux acteurs concernés grâce à leur *tag*.
La fonction *word-output* de Lux est utilisée pour l'affichage. Dans ce projet, nous affichons le monde "courant" grâce à la fonction *display-world*.
Le jeu s'arrête lorsque le *runtime* ne détecte plus l'acteur possédant le tag *player* dans le monde courant.

5 Tests effectués

Afin de s'assurer de la robustesse de nos structures ainsi que de la correction de nos algorithmes, nous avons tout au long du projet élaboré plusieurs tests à l'aide de la bibliothèque *RackUnit*. La commande *make test* exécute ces tests. Nous nous sommes concentrés sur la robustesse absolue de notre code, en conséquence plusieurs tests permettent de vérifier la validité de nos fonctions même dans les cas les plus extrêmes : de la belle mort d'un acteur, à la malheureuse création d'un autre, à l'extraordinaire voyage entre les mondes en survivant aux dangereuses collisions entre acteurs de différents *tags*, tout en passant par plusieurs positions pour enfin réaliser la *démo* dont le *gif* est présent sur la chaîne *SLACK* "projets-s-actors-gif". Comme prévu, tous les tests ont été programmés comme une application de plusieurs fonctions dans des scénarios où nous connaissons les conséquences avant l'exécution.

6 Conclusion

6.1 Programme et ses options

6.1.1 Code principal

En exécutant le programme à l'aide de la commande *racket src/main.rkt -f %* en remplaçant % par le nombre de *fps* (*frames per second*) voulu. Les *fps* sont maintenus à 10 par défaut, le jeu peut être lancé avec cette valeur en écrivant la commande *make*. Par exemple, l'utilisateur peut choisir l'option *f* en utilisant la notation suivante dans le terminal : **racket src/main.rkt -f 10**. Si cette syntaxe n'est pas respectée, un message d'aide apparaît et l'utilisateur doit à nouveau taper la commande. Si les valeurs rentrées ne sont pas compatibles (par exemple, *f<=0*), les valeurs par défaut sont utilisées.

6.1.2 Code de test

En lançant la commande *bash* **make test** dans le terminal, les codes de test sont directement compilés et exécutés. Il s'affiche alors à l'écran le nombre de tests réussis par catégorie de test, puis le nombre globale de tests réussis.

6.2 Difficultés rencontrées et retour sur expérience

D'une façon très générale, ce premier projet en langage *scheme racket* n'était pas exactement du même degré de facilité que les TDs de programmation fonctionnelle. Nous comprenons que ce cours était basé sur les bases de *racket*, mais la différence de difficulté entre les exercices résolus en TD et ce projet est énorme. Du coup, nous avons eu plusieurs problèmes, notamment sur la structuration du projet. De plus, le changement de façon de programmer entre un langage itératif et un langage fonctionnel a été difficile au début. Plutôt que de simplement affecter les instructions à la suite une par une, il a fallu programmer de sorte à ce que les changements s'effectuent et se cumulent à la suite. Nous pensons que le seul aspect itératif du projet est la boucle de jeu basé sur Lux.
De plus, il a été plus difficile de s'organiser à 4 plutôt qu'a 2 personnes, d'autant plus que nous n'étions pas familiarisé avec la création de jeu en racket (surtout la partie interface graphique et gestions des inputs). Il a fallu se concerter et comprendre ensemble le sujet pour définir des actions pouvant être faites en parallèle. Pour cela, nous avons dû définir l'architecture du projet en avance et notamment les formats des fonctions. Nous avons pu entrevoir à quel point il était embêtant de s'être trompé sur l'architecture lorsqu'il a fallu changer les formats des fonctions d'actions pour pouvoir renvoyer une liste de messages et une liste d'acteurs. Avec du recul, nous aurions peut être dû implémenter une structure *message* mais nous avons eu cette réflexion trop tard dans le projet.
Nous pensons que l'un des aspects les plus problématique de ce projet a été de debugger. Les erreurs d'exécutions sont particulièrement compliquées à gérer notamment à cause du typage dynamique du langage (et probablement notre manque d'expérience). La mise en place de contrats a été d'une grande aide pour combattre ce fléau.
Enfin, le côté autonome dans la recherche quant à l'utilisation de *raart* et de *Lux* a été assez intéressant. Nous nous sentons maintenant plus enclins à utiliser des bibliothèques dans notre code. Nous avons aussi pu réaliser l'importance de la documentation et de la clarté des fonctions(pour qu'elles soient faciles à identifier et à utiliser).
Pour résumer, nous laissions notre imagination couler(Implémentation de plusieurs mondes, combat contre des *boss* , des événements ...), et c'est exactement cela qui a rendu ce projet plus difficile à programmer. Enfin, créer ce jeu était une expérience très appréciable, nostalgique. Et, nous trouvons que cette nostalgie des jeux du passé, la curiosité d'explorer des bibliothèques d'animation et d'affichage graphique et la compétitivité entre les groupes pour réussir à proprement lancer son jeu nous a certainement aidé à nous évader pendant cette pandémie. Ce projet nous a aidé à renforcer nos compétences dans le langage *scheme*, à développer notre esprit d'équipe et de collaboration, ce qui est pour nous une préparation essentielle à ce qui nous attend dans les années à venir, et à réanimer la passion, de certains d'entre nous, à créer des jeux vidéos. C'était pour cela un excellent projet à programmer malgré les conditions particulières du confinement et nous souhaitons qu'il soit aussi merveilleux à corriger.

7 Remerciements

Nous remercions tout d'abord M. David RENAULT pour les remarques, aides, observations mais aussi la bonne humeur qui nous ont été données soit par *SLACK*, soit par courriel, ainsi que pour l'élaboration de ce projet original. De plus, nous remercions M. Sylvain LOMBARDY, qui nous a accompagné tout au long du projet en nous encadrant et en répondant à toutes nos questions. Ses réflexions sur l'importance de la rigueur dans la programmation nous ont été utile pour ce projet et nous seront probablement utiles pour le reste de notre vie de programmeur.

8 Références

8.1 Internes

— `https://www.labri.fr/perso/renault/working/teaching/projets/`
— `https://www.labri.fr/perso/renault/working/teaching/schemeprog/schemeprog.php`

8.2 Externes

— `https://docs.racket-lang.org/plait/Tutorial.html`
— `https://www.overleaf.com/`