Rapport Projet PFA

Hugo BARREIRO LDD3 Magistère Info
Adrien CHALLE LDD3 Magistère Info

19 avril 2024

Table des matières

1 Description du jeu

1.1 But du jeu

Nous sommes un ovni (texturé comme une fusée pour des raisons de jouabilités) dans l'espace. Néanmoins, nous nous sommes égaré et nous trouvons à présent au milieu d'une horde d'astéroïdes.

L'objectif du jeu est de survivre le plus longtemps possible afin d'accumuler les points. Ainsi, le jeu n'a pas de fin et est donc un infinit runner.

Rentrer en collision avec un astéroïde ou se faire rattraper par le décor fait perdre une vie (sur un total de 5). On doit donc esquiver ou casser (à l'aide de lasers) les astéroïdes pour survivre.

1.2 Vagues et Bonus

Les astéroïdes arrivent par vagues. Leur rapidité augmente au fur et à mesure que le nombre de vagues auxquelles nous avons survécu augmente. Ainsi, la difficulté du jeu augmente au fil du temps.

C'est pour cela, qu'au bout d'un moment, nous avons besoin de bonus pour survivre. Ceux sont simplement des astéroïdes spéciaux qu'il faut casser pour obtenir un bonus. Il en existe 5 types avec des raretés différentes :

— commun, de couleur marron
— atypique, de couleur verte
— rare, de couleur bleue
— épique, de couleur violette
— légendaire, de couleur jaune

Certains ont des effets permanents, d'autres éphémères. De plus, les bonus sont cumulables lorsqu'ils sont différents de ceux actifs sinon ils rallongent la durée de ces bonus.

Liste des bonus :

— 2 communs, augmente le score et augmente la vitesse de l'ovni (temporaire)
— 2 atypique, double les points gagnés (temporaire) et augmente la vitesse de l'ovni (temporaire)
— 2 rares, permet de tirer avec plusieurs lasers de manière répartie (temporaire) et augmente la vitesse de l'ovni (permanent)
— 2 épiques, gagne une vie et déclenche une bombe nucléaire
— 3 légendaires, augmente le nombre de laser (permanent), augmente la vitesse de tir (permanent) et déclenche l'étoile d'invincibilité (temporaire)

1.3 Comment jouer

1.3.1 Lancer le jeu

Pour construire le projet, il suffit d'effectuer un `make` à la racine du projet.

Nous recommandons de lancer le jeu avec sa version Javascript. En effet, SDL ne fonctionne pas correctement sur nos équipements, ainsi nous n'avons pas pu approfondir les tests sur la version SDL. Ainsi, toutes les features présentent dans la version JS sont également présentent dans la version SDL ; néanmoins il peut avoir des ralentissements ou des "sauts" d'images avec SDL.

Pour jouer avec la version Javascript, il faut lancer un serveur avec la commande :

```
python3 -m http.server
```

Puis aller à l'adresse `http://localhost:8000` sur un navigateur de recherche. Pour la version SDL, il suffit d'utiliser la commande `make exec` à la racine du projet.

1.3.2 Les contrôles

Pour se déplacer il faut utiliser les touches `z`, `s`, `q` et `d`, afin de se diriger respectivement vers le haut, le bas, la gauche et la droite.

Pour tirer un laser, il faut simplement appuyer sur `espace`.

On peut mettre le jeu sur pause avec `m + échap`, puis reprendre la partie avec `p + échap`. On peut aussi quitter la partie avec `m + &`.

Pour tester et présenter le jeu, des "cheatcodes" ont été implémentés. Ainsi, `m + é` active le "god mode", i.e. on devient invincible (`p + é` pour le désactiver). On peut également ne jouer qu'avec des bonus grâce à la combinaison `m + "` et revenir à la normale avec `p + "`.

On peut remarquer que `m` sert à activer un mode et `p` à le désactiver.

2 Organisation du code

Notre dépôt *GitHub*.

Sur le GitHub il manque les commits du début de projet car nous sommes passés de GitLab à GitHub entre temps.

2.1 Groupement des fichiers

Le projet est organisé en 4 grands répertoires :
— `lib`, les bibliothèques sur lesquelles s'appuie la réalisation du projet
— `prog`, les fichiers qui vont produire les exécutables (1 pour JS, 1 pour SDL)

— `resources`, les ressources externes au code (images, animations, etc...)
— `src`, le code source du projet qui manipule le contenu des 3 autres répertoires pour produire le jeu final

Détaillons le contenu du répertoire `src` puisqu'il est celui le plus fourni :
— `components`, les composants du modèle ECS (avec de plus, ce qui est en rapport, par exemple le fichier d'initialisation et celui qui déclare les entités connus globalement)
— `systems`, les systèmes du modèle ECS
— `core`, les autres parties du code (affichages, textures, score, commandes, bonus, timers, variables globales, etc...)
— `game.ml`, les fonctions principales du jeu
— `components_defs.ml` et `systems_defs.ml`, déclarations et définitions des composants et des systèmes

2.2 Composants, Systèmes et Bibliothèques

2.2.1 Composants

Nous avons beaucoup de composants dans notre code. Nous allons évoquer que les principaux.

Les `movable` composés d'une position, de frottements, d'une vitesse et d'un booléen afin de savoir si ils sont soumis à la gravité ou non. Ceci définit les composants en mouvements.

Les `collidable` qui inclus les `movable` en plus d'une masse, de la somme des forces, une force de rebond, un rectangle et un type d'objet. Ainsi, nous avons un type de composant pour les objets qui sont en mouvements avec des collisions.

Les `drawable` composés d'une position, d'un rectangle et d'une texture. Ainsi, nous pouvons dessiner nos composants avec une texture (couleur uni, image ou animation).

Les `box` qui combine les `movable` et les `collidable` avec un id afin de les identifier. Nous avons donc notre composant de base afin d'afficher nos objets en mouvement avec des collisions.

Les `offscreen` pour gérer nos composants qui sont voués à sortir de l'écran, que l'on va vouloir supprimer (lasers et astéroïdes) qui sont composés d'un booléen `is_offscreen`, d'un booléen `is_dead` (par exemple, ces composants peuvent être des astéroïdes et on doit savoir s'ils sont morts lors de leur suppression, notamment pour les astéroïdes bonus) et d'une fonction `remove` qui définit l'action à effectuer lors de la mort ou de la suppression de ce composant.

Les `offscreenable_box` qui combine `box` et `offscreen` en plus d'un nombre de points de vie et d'un niveau (tout deux pour les astéroïdes).

L'`ovni` composé d'une `box` avec des points de vie, et de deux booléens pour gérer son invincibilité (par exemple lorsqu'il est rentré en collision avec un astéroïde) et son droit à tirer.

Et enfin, les `box_collection` qui contiennent une **`Hashtbl`** d'`offscreenable_box` ainsi qu'un booléen pour différencier les astéroïdes des lasers et des fonctions pour gérer leur traitement et leur suppression en particulier.

2.2.2 Système

Nous avons 5 systèmes :

- `move.ml`, met à jour les mouvements des composants (par rapport à leur vitesse)
- `forces.ml`, met à jour les mouvements des composants (par rapport à la gravité et les frottements)
- `collisions.ml`, effectue correctement les collisions entre les composants (en collisions)
- `draw.ml`, dessine les composants à l'écran
- `cancellable.ml`, supprime les composants qui doivent l'être (ceux en dehors de l'écran et ceux morts)

Remarquons que nous avons ajouté un système non fourni car il était essentiel à notre jeu. De plus, nous avons fait une modification dans `collisions.ml` ; nous transformons le `seq` en **`array`** avant d'itérer sur les `collidable` car cela nous permet d'effectuer une vraie boucle *for* et donc de ne considérer que $n^2 \div 2$ tours à la place de n^2 avec n le nombre de `collidable`.

2.2.3 Bibliothèques

Nous utilisons les deux bibliothèques fournis **`Gfx`** et **`Ecs`** pour respectivement afficher à l'écran et gérer la collaboration entre les composants, systèmes et entités.

Néanmoins, nous utilisons une 3ème bibliothèque (faite maison) pour compléter **`Gfx`** que nous avons nommé **`Audio`** qui, comme son nom l'indique, nous permet de lancer du son. Nous détaillons cette bibliothèque plus tard.

3 Répartition du travail

Le travail a été réparti équitablement. Globalement, nous avons, tous les deux, travaillés sur toutes les composantes du projet.

Néanmoins, Adrien s'est plus concentré sur les textures et les sons, et Hugo sur les vagues d'astéroïdes et les bonus.

Cela ne nous a pas empêché se s'aider l'un l'autre sur "ses" parties du code, notamment à travers la recherche d'idées, l'élaboration de solutions et des séquences de débogages.

4 Descriptions de certaines fonctionnalités

4.1 Bonus et Timers (par Hugo)

4.1.1 Première approche

Au départ notre jeu était dépourvu de bonus et donc n'avait que deux timers : un pour l'invincibilité après avoir été en collision avec un astéroïde, et un pour le délai avant de pouvoir tirer de nouveau après un premier tir.

Pour ce faire, on maintenait deux références globales que l'on décrémentait à chaque *frame*. Or, lorsque l'on a décidé d'ajouter des bonus à notre jeu, cette solution n'était plus viable.

On a du donc changer de stratégie afin de passer à l'échelle.

4.1.2 Passer l'échelle

Pour passer à l'échelle, une des solutions est de créer de nouveaux composants et un nouveau système dédiés aux timers. Mais ce n'est pas ce que j'ai retenu pour plusieurs raisons.

Tout d'abord, jusqu'ici tous nos composants se complètent et sont reliés les uns les autres. Or, notre système de timers, n'aurait rien à voir avec les composants déjà existants, ce qui n'est pas pratique puisqu'il faudrait repartir de zéro. De plus, cela rendrait également le code des composants moins compréhensible car on mélangerait deux *features* totalement indépendantes.

Ensuite, par définition comme un timer expire, l'implémentation des systèmes n'est pas vraiment adéquat. En effet, on a vu précédemment tous les composants et le système que l'on a du créer pour supprimer les lasers et les astéroïdes.

Enfin, on voudrait que si un timer pour un bonus donné est déjà lancé, alors on rallonge sa durée au lieu d'en créer un nouveau. Comme les systèmes sont des `seq`, la recherche et la mise à jour sont en temps linéaire, ce qui n'est pas optimal (par exemple pour une `Hashtbl` cela serait en temps constant).

4.1.3 Solution finale

J'ai opté pour un petit module `Timer`. Celui-ci contient une `Hashtbl` de timers (non accessible depuis l'extérieur du module), une fonction permettant de lancer un nouveau timer, une fonction qui met à jour les timers et d'une fonction permettant de récupérer les bonus actifs (pour l'affichage).

Pour ce faire, il nous faut un type pour représenter les timers. Le type que j'ai choisi est une structure `typ_timer` à deux champs :

```
type typ_timer = { mutable time : int ; f : unit -> unit }
```

Le 1er est un entier mutable que l'on va décrémenter à chaque *frame* et le 2ème une fonction de type `unit -> unit` qui sera appelée à la fin du timer afin d'annuler un effet (pour les bonus par exemple) ou de changer un état du jeu (l'invincibilité de l'ovni par exemple).

Ensuite, comme un timer doit être clairement identifié (car si un timer est déjà lancé pour une certaine action, on ne doit pas ajouté un nouveau timer mais le rallonger), j'ai créé un type algébrique `kind_timer` qui répertorie tous les différents timers sans aucun paramètre :

```
type kind_timer =
  | OvniDelayShoot | OvniInvicible | SplitShoot | Nuke | ...
```

Donc, notre `Hashtbl` qui stocke l'ensemble des timers est de type :

```
(kind_timer, typ_timer) Hashtbl.t
```

Ainsi, pour ajouter un timer il suffit de regarder si un timer de la sorte est déjà lancé, si oui on rallonge son temps, sinon on l'ajoute à la `Hashtbl`. Pour mettre à jour les timers, il suffit de parcourir la `Hashtabl` et de décrémenter de 1 chaque timer, si il tombe à zéro alors on le supprime (de la `Hashtbl` et donc du jeu) et on exécute sa fonction associée.

4.1.4 Implémentation des bonus

Il existe deux types de bonus : ceux à effet permanent et ceux à effet temporaire. Pour les permanents il suffit simplement d'effectuer l'effet souhaité. Pour les temporaires, il suffit d'effectuer l'effet souhaité puis de créer un timer avec la durée du bonus en plus d'une fonction qui annule son effet. Ainsi, avec notre petit module `Timer`, implémenter les bonus devient assez simple.

De plus, certains bonus permanents pourraient ralentir ou déséquilibrer le jeu (par exemple, si on autorise un nombre de lasers trop grand), on veut donc pouvoir enlever ces bonus de ceux disponibles (à partir d'un certain moment). C'est pour cela qu'on les stockent dans une `Hashtbl`.

Comme pour les timers, on a un type algébrique pour identifier les différents bonus :

```
type kind_bonus =
  | IncreaseNbLasers | IncreaseShootSpeed | SplitShoot | ...
```

Ainsi, notre `Hashtbl` est de type :

```
(kind_bonus, unit -> unit) Hashtbl.t
```

Lorsque l'on veut enlever un bonus de ceux disponibles, il suffit de le retirer de la `Hashtbl`.

On a en réalité une `Hashtbl` par rareté de bonus afin que le tirage au sort des bonus soit plus simple ; c'est-à-dire qu'on tire un entier entre 0 et 99, et selon sa valeur on tire au sort dans la `Hashtbl` des bonus d'une certaine rareté. Ainsi, nous pouvons aisément contrôler la rareté des bonus de façon à ce que les bonus ayant les plus forts effets soeint les plus rares.

4.2 La bibliothèque Audio (par Adrien)

4.2.1 Première approche

Étant donné qu'aucune des bilbiothèques fournies dans le squelette du projet (que ce soit **`Gfx`** ou **`Ecs`**) ne proposait de méthodes pour jouer des sons, j'ai dû regarder quelle bibliothèque tierce pourrait m'aider. Au premier abord je pensais partir sur la bibliothèque **`Audio`**, qui, comme son nom l'indique, permet de jouer des sons. Je me suis vite rendu compte qu'elle ne correspondait absolument pas aux attentes du projet car ce dernier devait par la suite est compilé en `JS` et `SDL`. J'ai donc décidé de créer une librairie permettant de gérer les sons au même titre que **`Gfx`** gère les graphismes.

Avec cette idée en tête, j'ai longuement analysé la manière dont était écrite la librairie **`Gfx`**, que ce soit un fichier `.mli` pour deux fichier `.ml` ou les trois fichiers `dune`. Je suis donc parti sur cette structure qui semblait bien fonctionner.

4.2.2 La partie JavaScript

Bien que ce fût la partie qui semblait être la plus simple, il y avait tout de même quelques pièges venant des navigateurs à déjouer.

Pour ce qui est de l'implémentation, le fichier `audio.ml` agit simplement comme une passerelle entre le code `OCaml` et le fichier `resources/scripts/audio.js` en passant les chemins d'accès des fichiers audio au script `JS` lors de la phase d'initialisation, puis en transformant les valeurs du type algébrique associé à chaque son en entier lors de la lecture d'un son, permettant ainsi de sélectionner le son à jouer dans le fichier `JS`. Les fichiers audios sont ainsi chargés dans des objets de classe `Audio` dans le fichier `JS`.

Le problème auquel j'ai été confronté est relativement récent et est dû à la nouvelle politique anti-autoplay des navigateurs, interdisant la lecture d'un son sans interaction préalable de l'utilisateur. Pour contourner cette dernière, un bouton (invisible) par effet sonore a été ajouté au fichier `HTML`. Lors de l'initialisation on ajoute un callback à ce bouton qui est ensuite appelé lorsqu'on simule un click sur ce même bouton via `btn.click()`, permettant ainsi de passer outre les restrictions concernant l'autoplay.

4.2.3 La partie SDL

Pour la partie `SDL`, j'ai fait le choix d'utiliser la librairie *tsdl-mixer* qui est faite pour jouer des sons en harmonie avec la librairie *tsdl* utilisée initialement dans le projet.

Le code est plutôt expressif. Lors de l'initialisation on initialise **`Sdl`** et sa partie audio, après quoi on ouvre le module **`Audio`**. Si ces deux étapes se sont bien déroulées, alors on poursuit avec l'allocation de channels audios sur lesquels seront joués les différents sons de notre programme. On termine cette phase d'initialisation en chargeant les fichiers audios pour récupérer des `chunk`, qu'on stockera dans une **`Hashtbl`** avec pour clefs les valeurs du type abstrait `audio_kind`.

La lecture, quant à elle, est bien plus concise car elle consiste simplement à récupérer le `chunk` correspondant à la clef passée en argument dans la **`Hashtbl`** et à la jouer sur le premier canal de disponible (l'argument `-1` de l'appel).

4.2.4 Utilisation

Cette librairie rend la lecture de son extrêmement simple. En effet, il suffit de passer les chemins d'accès au fichiers `.wav` en argument de la fonction `init` lors du lancement du jeu et ensuite chaque son peut être joué à l'aide de la seule instruction **`Audio`**`.play *audio_kind*` et ce depuis n'importe quel endroit du code. Il faut néanmoins veiller à bien passer en argument des fichiers `.wav` car la fonction **`Mixer`**`.load_wav` utilisée en `SDL` ne permet que de charger des fichiers `.wav`, d'où le nom.

5 Tests

Tout d'abord, nous avons essayé au maximum de lancer le jeu sur ces deux versions (malgré les difficultés de lancer SDL avec nos distributions...) car elles ne réagissent pas de la même manière à certaines parties du code (par exemple, affichages non actualisés). En effet, parfois un bug pouvait être présent dans une version et pas dans l'autre. Nous avons corrigé certains bugs avec cette méthode.

Ensuite, nous nous sommes beaucoup servis de l'affichage offert par **`Gfx`**. Il permettait se s'assurer qu'une action soit bien déclanchée par une autre. Qu'une action précise exécute bien une partie du code voulu. Par exemple, l'affichage du nombre de *components* nous a permis de vérifier que ceux en dehors de l'écran étaient bel et bien supprimés.

De plus, nous avons essayé grâce aux "cheatcode" de tester le jeu dans toutes les situations "limites" où des bugs pouvaient être présents. Plusieurs bugs ont été détectés ainsi. Cela a été le cas lorsque plusieurs bonus étaient actifs en même temps. Par exemple, si on rentrait en collision juste avant de prendre une étoile d'invincibilité, la période d'invincibilité était désactivé durant l'effet de l'étoile, ce qui n'était pas voulu.

Enfin, nous avons régulièrement lu les codes de chacun afin d'avoir une vision nouvelle sur ceux-ci dans l'objectif de détecter de potentiels bugs mais aussi pour les optimiser si cela était possible.