ENSEEIHT

Programmation Fonctionnelle
Rapport

Projet Newtonoid

Élèves :
Romain Haessler
Benjamin Pascal
Leïlie Canillac
Vianney Hervy

Enseignant :
Guilaume Dupont

23 janvier 2025

Table des matières

1 Introduction

2 Architecture

Les sources sont réparties en deux répertoires : `bin` et `lib`.

— `bin` contient le programme principal `newtonoid.ml` qui peut utiliser les modules de la bibliothèque `Libnewtonoid`.

— `lib` contient les modules et interfaces impémentant les divers éléments du jeu. Le fichier `lib/dune` contient une directive pour créer une bibliothèque appelée `Libnewtonoid` contenant tous les modules de `lib`.

Les répertoires `configs` et `levels` contiennent respectivement des configuration et des niveaux d'exemple.

2.1 Modules

La plupart des modules de `Libnewtonoid` sont dotés d'une interface documentée en `.mli`. Certains modules sont relatifs à des objets : `Ball`, `Box`, `Brick`, `Level`, `Paddle`, `State`. Tandis que d'autres sont un groupement de fonctions nécessaires à une fonctionnalité : `Collision`, `Input`, `Iterator`, `LoadLevel`, `Utils`, `ParamValidator`, `Iterator`.

La plupart des modules contiennent un sous-module foncteur (au sens d'OCaml) paramétré par un module `P`. On a donc une structure récurrente de la forme : `module Make(P: Params)` qui permet d'instancier le foncteur avec les paramètres fournis par l'utilisateur.

3 Objets

Chaque objet du jeu est modélisé par un type enregistrement afin de permettre un accès efficace aux différents champs (par opposition aux tuples qui nécessitent du filtrage par motif).

Chaque type objet est appelé `t` et est défini dans son propre module qui regroupe les fonctions associées. Parmis celles-ci, on note surtout `make`, `draw` et `draw_shadow` et qui permettent respectivement de créer un objet, de l'afficher à l'écran et de dessiner son ombre.

4 Niveaux de jeu

Initialement, un niveau de jeu était défini en OCaml et chaque brique ajoutée séparément. C'était à la fois verbeux et peu pratique. Nous avons donc "développé" une syntaxe très simple pour définir un niveau textuellement. Chaque type de brique est représenté par un caractère différent et l'air est représenté par un espace. Les bordures sont fixées avec les caractères tiret et barre verticale. Deux exemples de niveaux sont donnés à la figure 1.

Des exemples de niveaux sont disponibles dans le répertoires `levels`.

```
|----------------|                    |----------------|
| ############## |                    | @+ + + + + + @ |
| ============== |                    | @ + + + + + +@ |
| ++++++++++++++ |                    | @+ + + + + + @ |
|                |                    | @ + + + + + +@ |
|                |                    | @@@@@@@@@@@@@@ |
|                |                    |                |
|----------------|                    |----------------|
```

FIGURE 1 – Exemples de niveau

Cette solution a donné une liberté de création qui pose un autre problème. Désormais, les niveaux peuvent avoir différents formats. Afin d'éviter les niveaux débordant de la fenêtre, les dimensiosn de cette dernière sont fonction directe des dimensions du niveau et de la taille des briques.

5 Paramètres

Nous avons vite compris que la possibilité de changer efficacement les paramètres était cruciale pour le développement et pratique pour le joueur. C'est pourquoi nous avons créé un module `Params` qui contient tous les paramètres du jeu. L'idée était de paramétrer les autres modules en fonction de `Params`. Ainsi, chaque module est en partie défini par les paramètres qui lui sont donnés.

Cette paramétrisation des modules permet de diminuer le nombre d'arguments des fonctions ainsi que la présene de valeurs "magiques" dans le code.

Pour faciliter la gestion des paramètres, nous avons développé un "parsing" primitif de fichier ".conf" qui récupère les paramètres du jeu et crée un module `Params` correspondant. Cette fonctionnalité est implémentée via le foncteur (au sens d'OCaml) `Params.Make` qui prend en argument un simple module contenant le nom du fichier de configuration et renvoie un module `Params`.

Des exemples de fichiers de configuration sont disponibles dans le répertoire `configs`.

6 Tests

Les tests unitaires de ce projet sont réalisés avec la bibliothèque `ppx_inline_test`.

Afin d'éviter des paramétrages insensés, à chaque lancement de jeu, `paramValidator.ml` valide (ou non) les valeurs données par l'utilisateur dans le fichier de configuration. On vérifie par exemple que `ball_r > 0`, `ball_init_vy <= ball_max_vy` etc.

7 Contrôles

7.1 Démarrer le jeu

Initialement, le jeu est en pause, la balle collée sur la raquette. Lors du clic du joueur, la balle est propulsée vers le haut et les mises-à-jour commencent à s'effectuer. Le clic change l'état de jeu de `Init` à `Playing`.

7.2 Mettre le jeu en pause

Le joueur a la possibilité de mettre le jeu en pause en cliquant sur le bouton de sa souris. La fonction `State.update` change le `game_status` de `Playing` à `Paused` (et inversement) lorqu'un clic est détecté. Étant donné que la boucle de mise-à-jour continue de tourner mais sans modifier l'état de jeu, il est impératif de rajouter un `sleep` lors d'un clic, pour éviter de le comptabiliser sur plusieurs frames.

8 Fin du jeu

La fin du jeu intervient dans deux cas : le joueur n'a plus de point de vie (état`GameOver`), ou le niveau ne contient plus que des briques incassables (état `Victory`). Dans les deux cas, la mise-à-jour s'arrête et un message de fin s'affiche à l'écran.

9 Améliorations

— **Taille du texte :** Les messages de fin de jeu affichés à l'écran sont écrit en petit. Le module `Graphics` d'OCaml ne possède pas d'implémentation de la fonction `set_text_size` pourtant spécifiée dans l'interface. Il est donc difficile de changer la taille d'écriture.