ENSEEIHT

Programmation Fonctionnelle
Rapport

Projet Newtonoid

Élèves :
Romain Haessler
Benjamin Pascal
Leïlie Canillac
Vianney Hervy

Enseignant :
Guilaume Dupont

26 janvier 2025

Table des matières

1 Usage

— **Compilation :** `dune build`
— **Tests :** `dune runtest`
— **Exécution (par défaut)** [1] **:** `dune exec bin/newtonoid.exe`
Les contrôles sont indiqués dans le jeu directement

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` et de tests. 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 de configuration `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.
Les objets définis juste après sont mieux documentés dans leur interface `.mli`.
— **Ball :** la balle du jeu, rebondit avec un facteur d'accélération soumise à la gravité.
— **Box :** Le rectangle de jeu, propre à chaque niveau.
— **Brick :** Les briques du jeu, avec un type de brique (`Standard`, `Unbreakable` etc) et un nombre de pv, une couleur, etc associés.
— **Paddle :** La raquette du joueur, se déplace horizontalement. Sa vitesse horizontale est calculée à chaque frame en fonction de sa position précédente.

3.1 États de jeu

Nous avons décidé de n'utiliser qu'un seul flux d'état de premier élément l'état initial et de fonction de "génération" `STATE.update`. C'est une sorte d'`unfold` [2] qui génère ce flux. Ainsi, à

1. Il est possible de choisir une configuration de jeu ainsi que des niveaux :
`dune exec bin/newtonoid.exe configs/default-dark.conf levels/level-1.txt levels/level-2.txt`
2. voir `Utils.unfold2`

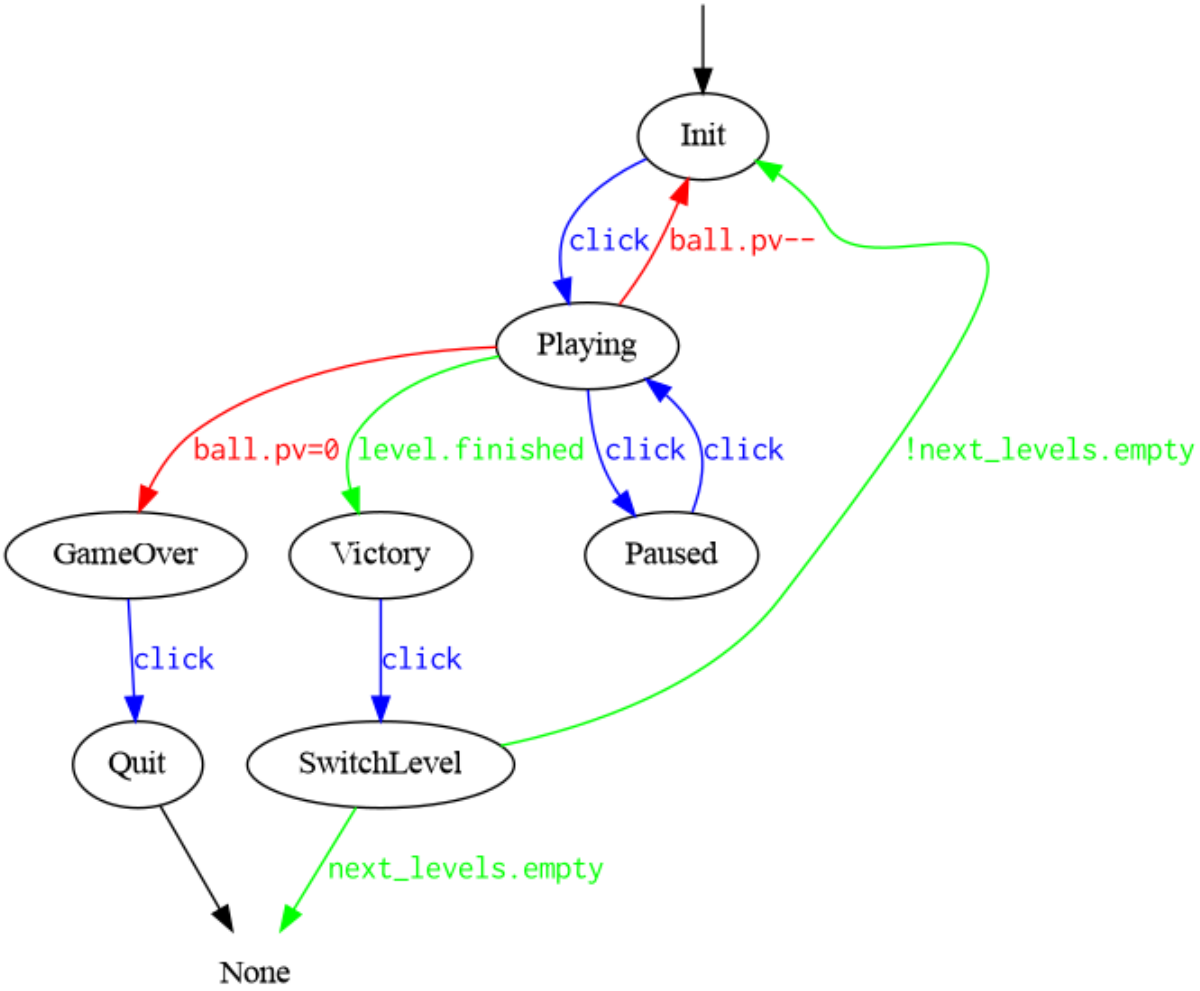

FIGURE 1 – Graphe de transition des états de jeu

partir de l'état initial et du flux de la souris, on obtient le flux des états de jeu. Un `STATE.t` est un enregistrement contenant les différents objets du jeu (le score, état de la balle, de la raquette, du niveau, etc). Son `game_status` est un type somme qui contient le status de l'état actuel (en pause, en jeu, fini etc). La figure 1 montre le graphe de transition des états de jeu.

3.2 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. la figure 2 montre deux exemples de définition de niveaux. La figure 3 montre un exemple de niveau tel que vu dans le jeu.

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

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

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

FIGURE 2 – Exemples de niveau

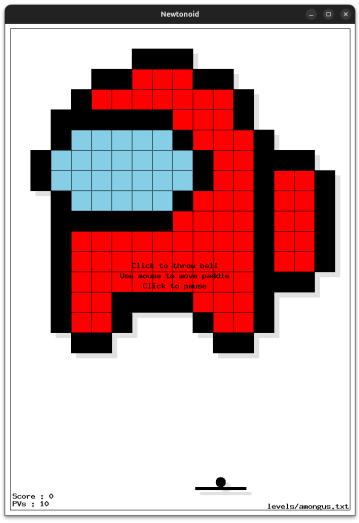

FIGURE 3 – Représentation du niveau issu du fichier `levels/amongus.txt`

dimensions de cette dernière sont fonction directe des dimensions du niveau et de la taille des briques.

4 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 en `.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`.

5 Tests

Les tests unitaires de ce projet sont réalisés avec la bibliothèque `ppx_inline_test`. Leur code se trouve dans le répertoire `test`. Les tests sont lancés avec la commande `dune runtest`.

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.

6 Améliorations et manquements

- **Gestion de l'espace de jeu :** À chaque frame, on vérifie les collision balle/brique pour toutes les briques. Le développement d'un quadtree (voir `lib/quadtree.{ml,mli}`) n'a pas abouti pour cause de blocage lors de la gestion des briques chevauchant plusieurs cadrans. Toutefois, le jeu tourne bien sans cette optimisation même avec un grand nombre de briques.
- **Sauvegarde :** Il serait intéressant de pouvoir sauvegarder et charger une partie en cours.
- **Niveaux :** La création de niveaux plus complexes et la gestion de niveaux aléatoires.
- **Effets :** Ajouter des effets spéciaux (briques qui explosent, balle qui se divise, etc).
- **Son :** Ajouter des effets sonores.
- **Lore :** Ajouter une histoire au jeu.