université PARIS-SACLAY | FACULTÉ DES SCIENCES D'ORSAY

semaine du 15 janvier 2024

Projet PFA
L3 Informatique

TP n° 2

1 Introduction

Le but de ce TP est :
— de se familiariser avec la notion d'Objets en OCaml
— de se familiariser avec le patron de conception ECS
Ce TP est évidemment là pour servir de base à votre Projet.

2 Code

L'archive `zip` disponible sur la page du cours contient un squelette de code. L'architecture du code est la suivante :

`dune` : fichier de configuration de l'outil `dune` permettant de compiler le code

`dune-project` : autre fichier de configuration

`index.html` : fichier HTML simple permettant de lancer le jeu dans un navigateur.

`README.md` : fichier contenant quelques informations sur la façon de compiler le projet, ainsi que des pointeurs bibliographiques.

`src/` : répertoire contenant le code source.

`prog/` : le répertoire contenant le fichier `.ml` servant de driver, dont le seul but est d'appeler la fonction `Game.run` écrite dans `src/game.ml`

`lib/` : un répertoire (à ne pas modifier) contenant les bibliothèquges `Gfx` (TP1) et `Ecs` (TP2)

La version JavaScript du projet (plus portable) peut être construit simplement avec la commande `dune build`. Le code natif peut être construit avec `dune build @sdl`. Pour construire toutes les cibles (Javascript et Native), vous pouvez utiliser `dune build @all`.

La structure du répertoire `src` est plus complexe qu'au TP1 et reflète la structure qu'aura votre jeu. Ce répertoire contient :

`game.ml` : le point d'entrée du jeu, contenant la fonction `run`

`core/` : un répertoire contenant des fichiers auxiliaires base : définition d'un type de donnée pour les vecteurs, les rectangles. On pourra plus tard y ajouter un type pour les textures, les divers objets du jeu, ...

`component_defs.ml` : un fichier contenant toutes les définitions de composants

`components/` : un répertoire contenant les fichiers manipulant des composants

`system_defs.ml` : un fichier contenant l'enregistrement des systèmes

`systems/` : un répertoire avec un fichier OCaml par système

3 Types de base

1. Ouvrir le fichier `prog/core/vector.ml` et le lire. Compléter les fonctions `dot` (qui calcule le produit scalaire), `norm` (qui calcule la norme d'un vecteur) et `normalize` qui normalize un vecteur donné (le transforme en un vecteur de même direction et de norme 1.0).
2. Ouvrir le fichier `prog/core/rect.ml` et le lire. Les seuls points importants pour cette séance sont la définition du type et la présence d'une fonction `intersect` qui permet de tester si deux rectangles dont on donne les coins supérieurs gauches s'intersectent ou pas.
3. Ouvrir le fichier `global.ml` et le lire. Ce dernier regroupera toutes les variables globales modifiables. Il en contient pour l'instant que la fenêtre graphique.

4 Composants et premier système

Pour représenter des composants (au sens du patron de conception ECS) nous allons utiliser des objets OCaml. La bibliothèque `Ecs` fournie dans le répertoire (`lib/`) propose deux modules.

4.1 `Component`

Le type `'a Component.t` est une classe possédant deux méthodes :

```
1  method get : 'a
2  method set : 'a -> unit
```

Il ne s'agit ni plus ni moins que d'un objet fonctionnant comme une référence cette dernière contient une valeur que l'on peut modifier. Le module contient aussi une fonction `def : 'a -> 'a Component.t` qui premet de créer une telle référence. Voici quelques exemples d'utilisation :

```
1  let r = Component.def 42 (* creation d'une référence contenant 42 *)
2  let () = r#set 18   (* on met 18 dans la référence *)
3  let () = Printf.printf "%d\n"  r#get  (* on affiche le contenu *)
```

4.2 `component_defs.ml`

Ouvrir le fichier `component_defs.ml`. Ce dernier contient la définition de plusieurs classes.

1. Lire le fichier. On remarque que des composants élémentaires sont des classes contenant :
   — un attribut (forcément privé en OCaml) introduit par le mot clé `val` et contenant une référence
   — une méthode du même nom que l'attribut et donnant accès à la référence
   Par exemple la classe `position` définit une unique méthode `position` qui contient une référence vers un vecteur. Créer une classe `box` permettant de représenter les objets du jeu : murs, balle, raquette. Une telle classe doit avoir un identifiant (une chaîne de caractères), une position, un rectangle et une couleur.
2. Ouvrir maintenant le fichier `component/box.ml` et compléter la fonction

   `create : string -> int -> int -> int -> int -> Gfx.color -> box`

   Cette dernier prend dans l'ordre l'identifiant de la boîte, les coordonnées x et y, la largeur, la hauteur et la couleur de la boîte.
3. Ouvrir le fichier `game.ml` et créer 7 boites :
   — 2 murs horizontaux bleus de largeur 10 situés en haut et en bas de la fenêtre
   — 2 murs verticaux transparents (en vert sur la figure ci-dessous) situés à gauche et à droite de la fenêtre
   — 1 carré noir de 10 pixels de côté, représentant la balle, en mettant son coin supérieur gauche en (50, 295)
   — 2 « raquettes » noires de 10 pixels de large et 100 pixels de hauteur, centrée verticalement et situées à 30 pixels des bords.
   Attention, votre programme n'affiche pas encore ces rectangles, il faut juste les définir.

4.3 system/draw.ml

On s'intéresse maintenant au système de dessin. Les systèmes ont une interface fixe, décrites par le module `System`.

1. Compléter le code de la fonction `update : float -> drawable Seq.t -> unit` du fichier `draw.ml` cette dernière doit :
   — récupérer la fenêtre (grâce à `Global.window()`)
   — récupérer le contexte et la surface correspondante
   — effacer la fenêtre (y dessiner un rectangle blanc)
   — afficher tous les composants enregistrés pour ce système
2. Déclarer ce système dans `system_defs.ml`
3. modifier la fonction `create` du fichier `component/box.ml` pour enregistrer chaque boîte crée auprès du système de dessin. On prendra garde de convertir le type explicitement avec l'annotation `:> drawable`
4. Créer une fonction `update` dans `game.ml`, y appeler `Ecs.System.update_all` pour exécuter tous les systèmes et affecter la fonction `update` dans la boucle principale. Constater que les rectangles s'affichent (60 fois par secondes).

5 Déplacement

Ajouter un composant `velocity` contenant un vecteur vitesse et modifier le composant `box` pour contenir une vitesse.

1. Créer un système `sytems/move.ml` qui permet de déplacer un objet en fonction de son vecteur vitesse.
2. Donner (dans `game.ml`) un vecteur vitesse initial non nul à la balle, et l'enregistrer dans le système `Move_system`. Constater que la balle se déplace.

6 Collisions

Lire le fichier `systems/collision.ml`. Ce dernier est une version très naïve de la gestion de collision. Enregistrer ce module comme un système et constater que les collisions sont gérées. Il faudra ajouter 2 fonctions au module `Global` :

```
val scoring : unit -> int
val set_scoring : int -> unit
```

qui permet de se souvenir du joueur qui vient de marquer un point.

Terminer ensuite en ajourant la gestion du clavier (touches e/d pour le joueur 1 et u/j sur le joueur 2).