

TP n° 2

1 Introduction

Le but de ce TP est :

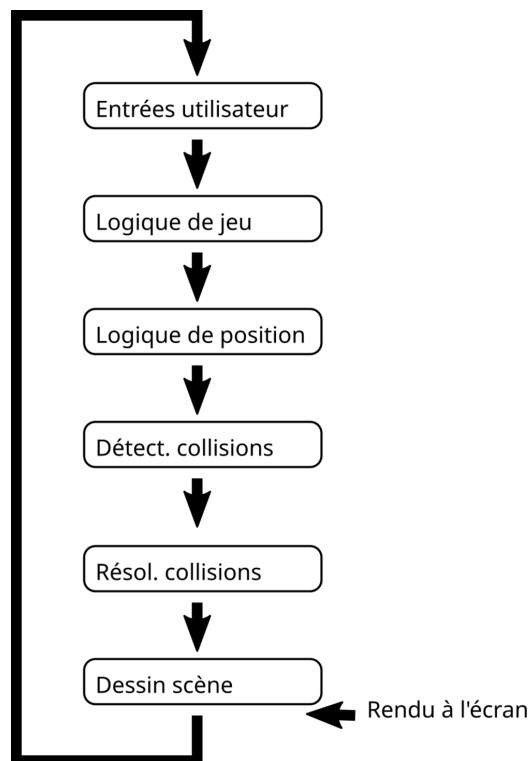
- de se familiariser avec la notion d'Objets en OCaml
- de se familiariser avec le patron de conception ECS

Ce TP est évidemment là pour servir de base à votre Projet. **Attention** : la feuille est très longue, avec beaucoup de choses à *lire*. Vous pouvez prendre votre temps et la faire sur 2 séances. Une fois comprise, vous aurez toutes les briques de bases pour programmer votre jeu.

2 Le modèle ECS

2.1 Boucle principale de jeu

Un jeu standard est composé d'une boucle principale de jeu. Idéalement, un « tour » de boucle effectue tous les calculs nécessaires à l'affichage d'une *frame* du jeu. Sur du matériel standard l'écran se rafraîchit avec une fréquence de 60Hz, il faut donc calculer 60 *frames* de jeu par seconde. Un tour de boucle doit donc durer un 60^{ème} de seconde ou 16.6667ms. Un exemple de boucle de jeu est la suivante :



Évidemment, il s'agit ici d'un jeu dans lequel il y aurait « des collisions » (jeu de plateforme, jeu de tir, ...). D'autres types de jeux auraient d'autres étapes intermédiaires, mais les étapes de gestion des entrées en début de boucle et de dessin de la scène en fin de boucle sont toujours présentes.

2.2 Le modèle ECS

Le modèle ECS (*Entity - Component - System*) est un patron de conception adapté à la programmation d'un jeu complexe. Il est en particulier adapté à la grande *variété* d'objets que les jeux peuvent avoir à manipuler : personnage principal, NPC, ennemis, plateformes, murs, textes, bonus, images de fond/paysage, ...

On constate en effet, que tous les objets ne sont pas concernés par toutes les étapes de la boucle de jeu. Par exemple, pour un jeu de plateforme de type Mario :

- le personnage principal, les ennemis et le score doivent être dessinés
- le personnage principal, les ennemis, les murs doivent pouvoir rentrer en collision (mais pas le score)
- le personnage, les ennemis, les plateformes, doivent pouvoir bouger (mais pas le score ni les murs)

De plus, on peut vouloir une gestion fine des collisions :

- une collision entre deux ennemis les fait rebondir (ou se traverser)
- une collision entre un mur/sol/plafond et autre chose arrête l'autre chose
- une collision entre personnage et ennemi :
 - si le personnage rencontre l'ennemi par le haut, alors l'ennemi meurt
 - sinon le personnage perd des points de vie

Pour un jeu complexe, il est hors de question de faire une seule grande fonction gérant tout cela. On décompose donc le code comme ceci :

entité : Chaque objet du jeu (personnage, mur, plateforme, ...) est une *entité*

composant : Chaque entité est associée à un ensemble de *composants* (position, vitesse, masse, points de vie, valeur du score, ...)

système : Chaque étape de la boucle principale est effectuée par un *système* (dessin de la scène, calcul des collisions, ...)

On doit donc associer des composants à des entités, puis enregistrer les entités auprès de systèmes. Une entité associée au système de dessin doit posséder une position et une texture. Une entité associée au système de mouvement doit posséder une position et une vitesse.

La plupart des bibliothèques de création de jeu (par exemple Unity) proposent un modèle ECS non typé :

- les entités sont des entiers (ID uniques)
- chaque composant est modélisé par un type (par exemple une position est une paire d'entiers) et associé à l'entier unique via une table de hachage
- chaque système contient une table de hachage dans laquelle sont stockés les entités enregistrées

Le système de dessin parcourt sa liste d'entité, puis pour chaque entité récupère le composant position et texture et les dessine à l'écran. L'un des problèmes de cette approche est qu'elle est non typée. Rien n'empêche le programmeur d'ajouter l'objet de score aux systèmes qui gère les déplacements. Cela provoque une erreur dynamiquement car, lorsque le système va chercher à récupérer le composant vitesse du score, il n'y aura rien dans la table de hachage.

Ce projet est donc un prétexte pour utiliser une bibliothèque ECS typées, qui démontre des aspects particulièrement sophistiqués du typage d'OCaml.

3 Objective Caml

Un aspect méconnu d'OCaml est sa couche objet (le « O » signifiant Objective, et désignant la couche Objet ajoutée par Didier Rémy et Jérôme Vouillon au langage Caml Special Light en 1996). On en donne une introduction rapide, juste nécessaire pour comprendre la bibliothèque *Ecs*. Une définition de classe en OCaml ressemble à ceci :

```
class point x y =
  object
    val mutable x = x
    val mutable y = y
    method get = (x, y)
    method set i j = x <- i; y <- j
  end

let p1 = new point 0 0
(* val p1 : < get : int*int; set : int -> int -> unit > *)
let () = p1#set 1 1
let () =
  let x, y = p1#get in Printf.printf "%d, %d\n" x y (* affiche 1,1 *)
```

Ce code déclare une classe **point**. Les arguments **x** et **y** sont ceux passés au constructeur. La classe contient deux attributs modifiables, initialisés avec les paramètres, et deux méthodes **get** qui ne prend pas d'argument et qui renvoie le couple des coordonnées et une méthode **set** prenant deux arguments et modifiant les attributs. Les attributs sont **toujours** privés et les méthodes peuvent être publiques ou privées. Pour les besoins de notre jeu, on n'utilisera pas d'attribut et uniquement des méthodes publiques. L'appel de méthode se fait avec #, le . étant déjà utilisé pour l'accès aux champs des enregistrements.

On peut évidemment étendre un objet par héritage. On peut aussi exécuter du code d'initialisation comme ceci :

```
class colored_point x y c =
  let c = if List.mem c [ "red"; "white"; "black"; "blue" ] then c
          else "black"
in
  object
    inherit point x y (* appel du constructeur de la classe parente *)
    val mutable c = c
    mutable color = c
    mutable set_color nc = c <- nc
  end

let p2 = new colored_point 1 2 "yellow"
(* val p2 : < get : int*int; set : int -> int -> unit;
    color : string; set_color : string -> unit > *)
```

L'interaction entre les objets et le polymorphisme d'OCaml est subtile :

```
let call_get o =
  let x, y = o#get in x + y
(*
  val call_get : < get : int * int; .. > -> int
*)
```

Ici, le type de **o** est automatiquement déduit comme étant :

< get : int * int; .. >

C'est-à-dire, le type de n'importe quel objet qui possède une méthode **get** renvoyant une paire d'entiers et éventuellement d'autres méthodes (symbolisées par ..). À cause de cette généricité, l'algorithme d'inférence de type doit parfois être aidé par des annotations, en particulier lorsque l'on veut utiliser une classe comme l'une de ses classes parentes :

```
let point_list = [ p1; (p2 :> point) ]
```

La notation (**e** :> **c**) où **e** est une expression et **c** une classe indique que l'on souhaite considérer la valeur renvoyée par **e** comme étant de la classe **c**. Le compilateur OCaml ne peut pas tout le temps déduire que cela est correct et demande donc au programmeur cette annotation (le mélange de sous-typage et polymorphisme rend l'inférence de types indécidable).

Enfin, on peut définir des signatures de classes comme ceci :

```
classe type printable_hashable :
  object
    method print : string
    method hash : int
  end
```

Ici, on a défini une signature de classe (en termes Java on appellerait ça une interface), on peut s'en servir lors de l'écriture de fonctions :

```
1 let f (o : printable_hashable) =
2   Printf.printf "Objet: %s, hash: %d\n" o#print o#hash
```

4 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 bibliothèques `Gfx` (TP1) et `Ecs` (TP2)

La version JavaScript du projet (plus portable) peut être construite 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

5 Types de base

1. Ouvrir le fichier `src/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 normalise un vecteur donné (le transforme en un vecteur de même direction et de norme 1.0).
2. Ouvrir le fichier `src/core/rect.ml` et le lire. Les seuls points importants pour cette séance sont la définition du type et la présence d'une fonction `rebound` qui renvoie `None` si deux rectangles ne s'intersectent pas et renvoie le vecteur de rebond si elles s'intersectent.
3. Ouvrir le fichier `src/core/cst.ml`. Ce dernier contient toutes les constantes utiles au jeu (taille de l'écran, taille des murs, ...). C'est une bonne pratique de regrouper tous ces constantes au même endroit, plutôt que de les disperser dans le code.
4. Ouvrir le fichier `src/core/global.ml` et le lire. Ce dernier contient la définition d'un type stockant l'état global du jeu, ainsi que deux fonctions `get/set` qui permettent de l'initialiser et de le récupérer globalement, depuis n'importe quel module.

6 Lecture du code

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 trois modules.

6.1 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 `init : 'a -> 'a Component.t` qui permet de créer une telle référence. Voici quelques exemples d'utilisation :

```
1 let r = Component.init 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 *)
```

6.2 Composants, Systèmes et Entités

1. Ouvrir le fichier `component_defs.ml`. Ce dernier contient la définition de plusieurs classes. Lire le fichier. On remarque que des composants élémentaires sont des classes ayant la structure très simple suivante :

- l'argument `unit` passé au constructeur
 - la définition d'une variable `r` locale à la classe contenant une référence objet vers une valeur par défaut
 - une unique méthode contenant le nom du composant que l'on veut représenter
 - le nom de la classe importe peu, mais il est choisi pour évoquer celui de la propriété (ou composant)
- Par exemple la classe `position` définit une unique méthode `position` qui contient une référence vers un vecteur. Le fichier se compose de trois parties :

- (a) La définition des composants (position, boîtes, textures, ...)
- (b) La définition des interfaces, celles-ci sont les types attendus par les systèmes (`collidable`, `drawable`)
- (c) La définition des entités à proprement parler : les objets de notre jeu (`wall`, `ball`, `player`)

Pour la définition d'interfaces et d'entités, il est demandé de **toujours** hériter de `Entity.t`. Cette dernière est une classe minimale, définie par la bibliothèque `Ecs` qui constitue la classe de base des entités. Les entités de base n'ont qu'une propriété, `name : string` qui permet de stocker un nom particulier pour certaines entités, à des fins de débogage. Le module `Entity` définit aussi des tables de hachage efficaces dont les clés sont des entités et qui sont utilisées en interne par la bibliothèque `Ecs`. Une interface aura donc toujours la form :

```
open Ecs

class type moninterface =
  object
    inherit Entity.t
    inherit composant1
    inherit composant2
    inherit composant3
    ...
  end
```

et une entité sera toujours de la forme :

```
class monentite () =
  object
    inherit Entity.t ()
    inherit composant1 ()
    inherit composant2 ()
    inherit composant3 ()
    ...
  end
```

2. Ouvrir le fichier `src/system/draw.ml` et le fichier `src/system/system_defs.ml`. Le fichier `draw.ml` contient en particulier :
 - un type `t`, défini comme égal à `drawable`
 - une fonction `init : float -> unit` ne faisant rien, mais pouvant servir à initialiser des valeurs en début de jeu

- une fonction `update : float -> t Seq.t -> unit`. Cette dernière prend en argument l'heure courante en millisecondes (mais n'en fait rien), la liste de toutes les entités enregistrées dans un conteneur `Seq.t` (type standard d'OCaml similaire aux listes) et ne renvoie rien. La fonction `update` récupère l'état global du jeu, et de celui-ci la fenêtre et le contexte graphique. Elle repeint la fenêtre en blanc, puis pour chaque entité enregistrée, la dessine sur l'écran. Comme on le voit, chaque entité est de type `drawable` et possède donc des composants `position`, `box` et `texture`.

Le fichier `system_defs.ml` utilise le foncteur `System.Make` de la bibliothèque `Ecs`. Ce dernier prend en argument un module possédant trois éléments :

- un type `t` (* Le type `t` tel que `drawable` *)
 - une fonction `init : float -> unit` (* la fonction d'initialisation *)
 - une fonction `update : float -> t Seq.t -> unit` (* la fonction de mise à jour *)
- et renvoie un module ayant la signature :

```
module type S =
sig
  type t                (* le même type t *)
  val init : float -> unit (* la même fonction d'initialisation *)
  val update : float -> unit (* la fonction de mise à jour, appelée
                             sur toutes les entités enregistrées *)

  val register : t -> unit  (* ajoute d'une entité au système *)

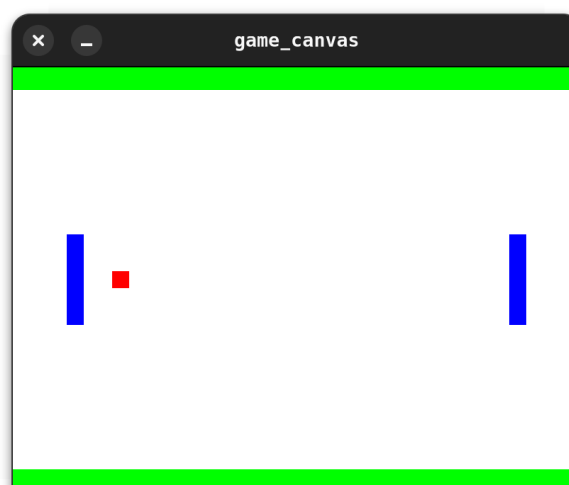
  val unregister : t -> unit (* retire une entité du système *)

  val reset : unit -> unit  (* retire toutes les entités du système *)
end
```

- Ouvrir le fichier `src/components/wall.ml`. Ce dernier contient une fonction `wall` permettant de construire un mur dont les paramètres sont donnés. Comme on le voit, une fois l'objet initialisé, ses composants sont mis à jours, puis l'objet est enregistré auprès des deux systèmes. Comme les types `wall` et `drawable` ou `collidable` ne sont pas égaux, mais en relation de sous-typage on utilise une conversion explicite. De plus, on utilise la facilité syntaxique : `Draw_system.(register (e :> t))` (noter la place des parenthèses) qui rend le type `Draw_system.t` visible dans les parenthèses, sans avoir besoin de mettre le nom du module devant.
- Lancer le jeu :

```
$ dune build
$ python3 -m http.server
# Puis ouvrir l'adresse http://0.0.0.0:8000 dans son navigateur
# Rafraîchir la page web avec CTRL-SHIFT-R
```

Vous devriez obtenir un jeu ressemblant à :



5. Ouvrir le fichier `game.ml` et le lire. On y voit :
 - la fonction de mise à jour de la boucle principale. À ce stade :
 - `Player.stop_player ()` ne fait rien
 - `Input.handle_input ()` gère les entrées au clavier, il pourra être lu dans un second temps
 - `Collision_system.update` est la fonction principale de la gestion de collision, qui ne détecte rien pour l’instant
 - `Draw_system.update ()` est la fonction de mise à jour du système de dessin.
 - La fonction `run` crée la fenêtre graphique, les différents éléments du jeu et enfin appelle la boucle principale de jeu.

7 Ajout des déplacement

Deux types d’entités peuvent se déplacer :

- la balle
- les joueurs (les « raquettes »)

Les déplacements vont être réalisés par l’ajout d’un composant `velocity` contenant un vecteur. Faire les modifications suivantes au code :

1. Ajouter dans `component_defs.ml` un composant `velocity`
2. Rajouter une interface `movable`. Cette dernière doit hériter d’`Entity.t` (comme toutes les interfaces et entités de notre jeu), de `position` et de `velocity`
3. Ajouter le composant `velocity` aux classes `player` et `ball`
4. Créer un fichier `systems/move.ml`. Ce dernier doit définir un système, c’est à dire un type `type t = movable`, une fonction `init : float -> unit` (probablement sans effet) et une fonction `update : float -> t Seq.t -> unit` qui modifie la position courante de chaque entité enregistrée en y ajoutant sa vitesse. Dans un vrai jeu, pour assurer de la fluidité, on utiliserait aussi l’heure courante, mais ce n’est pas utile ici. Déclarer ce système dans le fichier `src/systems/system_defs.ml`
5. Modifier `src/core/player.ml` aux endroits indiqués pour rajouter la vitesse et enregistrer le joueur auprès du `Move_system` créé à la question précédente
6. Modifier `src/core/ball.ml` aux endroits indiqués pour rajouter la vitesse et enregistrer la balle auprès du `Move_system` créé précédemment
7. Modifier `src/game.ml` pour appeler `Move_system.update` dans la fonction `update`
8. Croiser les doigts
9. Normalement les touches E/D font monter/descendre la raquette de gauche, les touches U/J font monter/descendre celle de droite et la touche G lance la balle. Vous devriez constater que des déplacements sont possibles, mais pour l’instant aucune collision n’est détectée

8 Détection de collisions

Nous allons mettre en place un mécanisme *générique* de détection de collisions. Ce mécanisme est implémenté par les deux composants `tagged` et `resolver`. Le composant `resolver` contient une (référence vers une) méthode `resolve` de type `Vector.t -> tag -> unit`. Cette méthode est appelée par le `Collision_system` lorsqu’il détecte que 2 objets sont en collision. Le premier argument est le vecteur de rebond, utilisé lors de la collision entre la balle et l’un des murs horizontaux ou une raquette. Le second argument de type `tag` stocke un type algébrique *extensible*. Ce dernier est défini au milieu du fichier `src/components/component_defs.ml` puis étendus dans `wall.ml` et `player.ml`. En effet, il n’est pas possible en OCaml de tester dynamiquement le type des objets (pas d’équivalent de `instanceOf`) de Java. On contourne cela en stockant dans l’objet un composant nous permettant de savoir de quelle nature il est.

1. Éditer le fichier `src/components/player.ml`. Dans la fonction de création, ajouter un gestionnaire de collision :

```
...
e#resolve#set (fun _ t ->
  match t#tag#get with
    Wall.Hwall (w) -> (* ici w est un objet de type wall
```

```

        On a detecté une collision entre la raquette et un mur horizontal.
        Redéplacer la raquette e pour qu'elle ne rentre pas dans le mur
        et mettre sa vitesse à 0. On pourra tester la position de w pour
        savoir si c'est le mur du haut ou du bas.
        *)
    | _ -> ()
);

```

Tester votre programme. Normalement les raquettes ne devraient plus rentrer dans les murs du haut ou du bas.

2. Selon le même principe, rajouter un gestionnaire de collision à la balle :

```

...
e#resolve#set (fun n t ->
    match t#tag#get with
    | Wall.Hwall _ | Player.Player -> (* ici la balle est en collision
        avec un des murs horizontaux ou une des raquettes.
        Le vecteur n passé en argument contient soit { x = -1.0; y = 1.0 }
        soit { x = 1.0; y = -1.0 }. Il suffit de le multiplier à la
        vitesse de la balle, composante par composante pour obtenir le
        symétrique du vecteur n et faire rebondir la balle.
        *)
    | Wall.Vwall (i, _) -> (* On est entré en collision avec le mur i=1
        pour gauche ou i=2 pour droite. Il faut arrêter la balle,
        la repositionner en face de la raquette correspondante puis
        mettre le champ 'waiting' de l'objet global à i (1 ou 2).
        Dans cet état, une pression sur la touche G fera repartir la
        balle.
        *)
    | _ -> ()
);

```

Si tout se passe bien, vous pouvez maintenant jouer à pong.