Projet PFA

Cours 1

kn@lri.fr
http://www.lri.fr/~kn

Plan

Projet PFA

PFA : Programmation Fonctionnelle Avancée

Le but est de faire un projet de programmation utilisant des aspects sophistiqués (système de module, typage, structures de données persistantes, algorithmes recursifs ...)

Le but est aussi de s'amuser

But final du projet

Le but du projet est de créer un **jeu**. Le choix du jeu est laissé libre mais il doit respecter quelques contraintes :

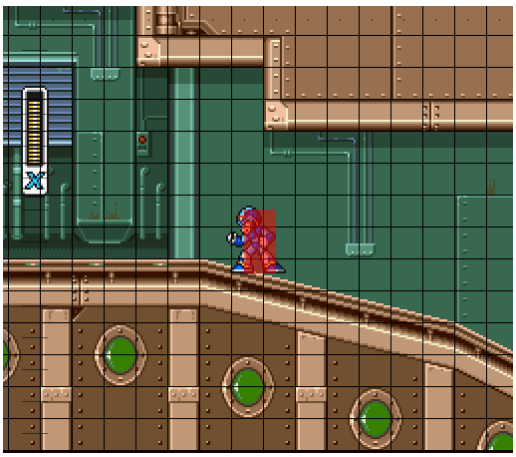
- C'est un jeu en 2D du type AABB (cf. slide suivant)
- Le jeu doit avoir une logique relativement complexe (notion de niveaux, de scores, bonus, ...)
- Le jeu doit être développé selon le modèle ECS
- Dans la mesure du possible, le projet doit être fait en binôme

Évaluation :

- Rapport + Code (détail donné plus tard sur le contenu exact) : 75%
- Soutenance : 25%

Le modèle AABB

Axis Aligned Bounding Box : dans ce modèle tous les objets du jeu sont dans des boîtes rectangulaires, alignées avec les axes verticaux/horizontaux (donc les boites ne peuvent pas « tourner »)



De nombreux jeux rentrent dans ce cadre : casse-briques, jeux de plateforme, shoot-em-up, bomberman-like, angry-bird-like, ...

Backend

Comment assurer que tout le monde va programmer un jeu graphique alors malgré la diversité des plateformes (Linux, Windows, MacOS, ...) ?

On va compiler contre code OCaml vers Javascript et l'exécuter dans le navigateur.

On fournit une petite bibliothèque simple permettant de gérer le clavier et dessiner des rectangles depuis OCaml sans avoir à connaître Javascript

⇒ La bibliothèque sera agrandie au fur et à mesure du projet (gestion des textures, de la souris, du son)

⇒ Les plus ambitieux peuvent rajouter un autre backend graphique (OpenGL, SDL, DirectX, ...)

Organisation des séances

Chaque séances : 2h supervisées, 2h libres

Les cinqs premières séances contiennent des cours utiles, les suivantes (après les vacances de février), vous travaillerez sur votre projet, nous feront le support technique

1. Présentation, rappels OCaml, modèle ECS. TP1 : Prise en main avec Pong
2. Utilisation de `git`, correction du TP1 et mise en place des binômes, discussions des choix de projets etc...
3. Concept avancé : moteur physique
4. Concepts avancé : enrichissement du moteur graphique

Plan

Plan

Notion de modules

En OCaml, un fichier `.ml` définit un **module**. C'est une unité de compilation qui définit des types et des fonctions associées. On peut rapprocher ça d'une classe en Java.

```
type t = { jour : int; mois : int; annee : int }

let bissextile a =
   a mod 4 = 0 && (a mod 100 != 0 || a mod 400 == 0)

let make j m a =
  let jmois = [| 31; 28; 31; 30; 31; 30; 31; 31; 30; 31; 30; 31 |] in
   if m >= 1 && m <= 12 then
     let jmax = if m == 2 && bissextile a then 29 else jmois.(m-1) in
      if j >= 1 and j <= jmax then
        { j = jour; mois = m; annee = a }
      else
        failwith "Jour invalide"
   else
     failwith "Mois invalide"
```

Notion de modules (suite)

```
(* suite du fichier date.ml *)
let jour d = d.jours

let mois d = d.mois

let annee d = d.annee

let equal d1 d2 =
     d1.jour == d2.jour && d1.mois == d2.mois && d1.annee == d2.annee

let hash d =
   d.jour * 17 + d.mois * 253 + d.annee

let pp fmt d = Format.fprintf fmt "%02d/%02d/%04d" d.jour d.mois d.annee
```

Notion d'interfaces

En OCaml, un fichier `.mli` définit une **interface** (ou **signature**). Cela permet de spécifier le type des valeurs du module et de masquer certaines définitions (les rendre « privées »).

```
(*  fichier date.mli *)
type date (* le type est opaque on n'affiche pas sa définition *)
val make : int -> int -> int -> date
val jour : date -> int
val mois : date -> int
val annee : date -> int
val equal : date -> date -> bool
val hash : date -> int
val pp : Format.formatter -> date -> unit
```

Dans le code ci-dessus, la fonction `bissextile` est masquée.

Ce concept peut être rapproché des interfaces de Java : c'est un « contrat » que remplit le module, il définit exactement ces valeurs et ces types.

Notion de sous-module

On peut définir des sous-modules dans un fichier.

```
  (* fichier shape.ml *)
module Point =
  struct
  type t = int * int
  let origin = (0, 0)
  let create x y = (x, y)
  ...
end

module Rect =
  struct
  type t = { top_left : Point.t;
             width: int;
             height : int;
  }
  let create p w h = { top_left = p; width = w; height = h }
  let top_left p = p.top_left
  ...
end
```

Notion de sous-module

Cela s'applique aussi aux interfaces.

```
  (* fichier shape.mli *)
module Point :
  sig
  type t
  val origin : t
  val create : int -> int -> t
  ...
end

module Rect =
  struct
  type t
  val create : Point.t -> int -> int -> t
  val top_left : t -> Point.t
  ...
end
```

Utilisation

Si on compile le code précédant, on peut l'utiliser ainsi:

```
(* fichier autre.ml *)

 let p1 = Shape.Point.create 1 1
 let p2 = Shape.Point.create 10 23
 let r1 = Shape.Rect.create p1 10 4
```

Évidemment, on a le droit de mettre des sous-modules dans des sous-modules,... mais on a rarement une profondeur plus grande que 2.

Signatures et types de modules

On peut définir une signature indépendemment de tout module. Cela permet de forcer un module à avoir **un certain type**. Cela permet aussi de vérifier que plusieurs modules ont les mêmes interfaces.

```
(* fichier shape.ml *)
module type S =
sig
   type t
   val create : Point.t -> int -> int -> t
   val area : t -> float
end
module Point = ...
module Rect : S =
struct
...
end

module Ellipse : S =
struct
...
end
```

Foncteurs

On veut souvent pouvoir paramétrer du code par un autre code.
Par exemple, on veut pouvoir paramétrer une fonction de tri, par une fonction de comparaison. On peut le faire simplement en utilisant de l'ordre supérieur.

```
let l = [ 1; 10; .... ]
let l_asc = List.sort compare l
let l_desc = List.sort (fun x y -> - compare x y) l
```

Considérons maintenant un exemple un peu plus complexe: les tables de hachage. Pour pouvoir utiliser un type de données comme clé d'une table il faut avoir défini sur ce type :

- Une fonction de hachage, qui renvoie un entier
- Une fonction d'égalité qui dit si deux valeurs sont égales

Foncteurs (2)

On pourrait définir toutes les fonctions du module `Hashtbl` comme ceci :

```
let create () = ... (* table vide *)
let find hash_f eq_f e table = ...
let add hash_f eq_f e v table = ...
let remove hash_f eq_f e v table = ...
...
```

- Ce serait très fastidieux. On devrait repasser à chaque fois les mêmes fonctions pour le type de données qu'on utilise comme clé.
- Ce serait dangereux, rien n'interdit de passer une fonction de hachahge ou d'égalité différente pour `add` et `find`.

On souhaite dire « toutes les fonctions de ce module prennent en paramètre implicitement les mêmes fonctions `hash_f` et `eq_f`.

Foncteurs (3)

Un **foncteur** est un module prenant en argument un autre module :

```
(* Fichier hashtbl.ml *)
module type HashedType = sig
  type t
  val equal : t -> t -> int
  val hash : t -> int
end

module Make (K : HashedType) =
struct
  type 'a t = ...
  let create () = ...
  let find e table =
     let he = (K.hash e) mod table.size in
     let bucket = table.data.(he) in
     snd (List.find (fun (k2, v2) -> K.equal e k2) bucket )
  ...
end
```

Foncteurs (4)

On peut maintenant utiliser un tel foncteur comme ceci :

```
(* fichier main.ml *)
module DateTable = Hashtbl.Make(Date)
(* DateTable est une table de hachage dont les clés sont de dates *)

let log = DateTable.create ()
let () =
   DateTable.add (Date.create 1 1 2021) "Premier évènement" log;
   DateTable.add (Date.create 2 1 2021) "Autre chose" log;
   ...
```

Foncteurs (5)

Lorsque le module passé en argument est petit, on peut le définir localement :

```
module IntTable = Hashtbl.Make(struct
                                 type t = int
                                 let equal a b = a == b
                                 let hash a = a
                               end)
```

Ouverture de modules

Il est parfois utile d'ouvrir un module. Cela a pour effet de rendre visible le contenu du module comme s'il avait été définit dans le module courant :

```
(* fichier autre.ml *)
 open Shape
 let p1 = Point.create 1 1  (* plutot que Shape.Point.create *)
 let p2 = Point.create 10 23
 let r1 = Rect.create p1 10 4 (* plutot que Shape.Rect.create *)
```

Ouverture de modules (2)

Attention, si deux modules définissent les mêmes identifiant, le second écrase le premier.

```
(* fichier autre.ml *)
 open Shape
 open Point (* contient une fonction create *)
 open Rect  (* contient une fonction create avec
               un type différent *)

 let p1 = create 1 1  (* erreur de typage *)
 let p2 = create 10 23
 let r1 = create p1 10 4
```

Le compilateur OCaml peut être configuré pour afficher un warning ou une erreur dans ce cas.

Inclusion de modules

On veut parfois inclure un module dans un autre. C'est différent de la directive `open`. Le code inclus est comme « copié-collé » dans le module qui reçoit l'inclusion.

```
module A = struct
  let r = 42
  let f x = x + 1
end

module B = struct
  open A
  let y = x (* x est visible ici *)
end

module C = struct
  include A
  let y = x (* x a été copié *)
end
```

Dans le code ci-dessus, `B.x` n'existe pas mais `C.x` et `C.f` oui.

Inclusion de modules (2)

Un cas d'utilisation est l'extension de module ou de signature, sans avoir à copier/coller :

```
module type HashedStrType = sig
     include HashedType    (* contient
                              type t
                              val hash : t -> int
                              val equal : t -> t -> bool *)
     val to_string : t -> string
end
```

Inclusion de modules (3)

Supposons que l'on veuille proposer notre module date.ml et que ce dernier contienne aussi le sous-module Table définissant une table de hachage ayant les dates comme clé :

```
(* fichier date.ml *)
module D = struct
  type t = { jour : int; mois : int; annee : int }
  let create j m a = ...
  let hash d = ...
  ...
end
module Table = Hashtbl.Make (D)
include D
```

Dans le code ci-dessus on a :

- le type date dans un module `D` pour le passer au foncteur `Hashtbl.Make`
- inclus `D` « à plat » dans le fichier, ce qui permet de faire `Date.create ...` plutot que `Data.D.create ...`
- `D` n'est pas mentionné dans `date.mli` donc pas visible de l'extérieur.