

TP n° 1

1 Introduction

Le but de ce TP est :

- de se familiariser avec un projet OCaml complexe
- de réviser des notions simples sur OCaml (types de bases, modules, enregistrement)
- montrer d'autres fonctionnalités encore plus avancées : motifs, opérateurs de composition,...
- d'apprendre à tester un programme OCaml compilé en JavaScript
- d'appréhender la bibliothèque graphique et son mode de fonctionnement

Ce TP est évidemment là pour servir de base à votre Projet.

Vous pouvez faire ce premier TP seul, mais il faudra vous organiser, avant la séance 2, pour être en binômes.

Rappel : Ces séances sont des séances de *projet*. Évidemment, lors des premières séances, des TPs sont données afin de vous présenter les différentes bibliothèques et le cadre du projet. Ces séances ont aussi pour but de vous faire écrire le code « commun » (le code de base que tout jeu graphique devra implémenter d'une façon où d'une autre).

Les feuilles ne sont pas forcément à finir lors des 2 premières heures, mais leur corrigé (mis en ligne) doit être compris pour pouvoir avancer.

Enfin, il est rappelé que les séances se découpent en 2h supervisées et 2h de travail libre.

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 `main` écrite en dans `src`

`lib/` : un répertoire (à ne pas modifier) contenant les bibliothèques `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`.

3 Organisation

Le but est de mettre en place l'environnement de travail et de commencer à prendre en main la bibliothèque de rendu graphique. On recommande de travailler de cette façon :

- éditer le code avec VSCode
- ouvrir un terminal à côté avec deux onglets (ou deux terminaux)
- dans un terminal, lancer un serveur Web, par exemple avec `python3 -m http.server`, depuis les sources du projet
- dans un second terminal, taper `dune build` au fur et à mesure que le fichier est modifié. Ce second terminal peut être démarré dans la fenêtre VSCode pour pouvoir pointer facilement les erreurs.

- pointer un navigateur Web vers `http://localhost:8000`. Bien recharger avec CTRL-SHIFT-R entre deux compilations.

4 But du TP

Le but du TP est de faire un petit programme graphique, qui évoluera jusqu'à devenir (au bout de la séance 2) un clone de *Pong*. Il pourra alors être modifié pour en faire un vrai jeu, plus complexe. Le but est d'explorer l'API de la bibliothèque `Gfx` développée pour ce projet, qui encapsule deux bibliothèques :

- SDL, pour un rendu natif sous Linux (OpenGL), MacOS (OpenGL) et Windows (DirectX)
- JavaScript (l'API Canvas), pour un rendu dans le navigateur

Un inconvénient dans le support de JavaScript est que son fonctionnement est « asynchrone », pour tout ce qui touche au chargement de ressource. Charger un fichier de texte, une image n'est pas une opération bloquante, mais rend la main immédiatement pendant que le chargement se fait en tâche de fond. Cela a un impact sur la structure du code et la boucle principale de jeu.

4.1 Présentation du code

Le code se décompose en deux point d'entrées :

`prog/game_js.ml` ce fichier est converti en fichier JavaScript (`prog/game_js.bc.js`) puis chargé depuis le fichier `index.html`. Ce fichier `index.html` prépare deux zones dans la page Web, une servant d'écran graphique et l'autre servant de sortie standard (pour le débogage). Pour que la compilation réussisse, il faut que le compilateur `js_of_ocaml` soit installé (c'est le cas au PUIO).

`prog/game_sdl.ml` ce fichier est compilé en un fichier `prog/game_sdl.exe` que l'on peut lancer dans un terminal. Pour que la compilation réussisse, il faut que la bibliothèque SDL ainsi que les *bindings* pour OCaml soient installés (c'est le cas au PUIO).

Les deux fichiers appellent la fonction `Game.run`, située dans le fichier `src/game.ml`. De cette façon il est possible de passer au « vrai » programme principal des paramètres spécifiques en fonction du mode dans lequel on est (JavaScript ou natif). Dans notre exemple, le paramètre passé est un tableau contenant les noms des touches de direction (qui diffèrent selon les *backend*).

Le fichier `prog/game_js.ml` est le suivant (les premières lignes permettent de dire que l'affichage de débogage doit arriver dans l'élément HTML d'id `console` de la page Web)

```
(* Main spécifique à Javascript *)
let debug = Gfx.open_formatter "console"
let () = Gfx.set_debug_formatter debug
let () = Game.run [| "ArrowLeft"; "ArrowRight"; "ArrowUp"; "ArrowDown" |]
```

Le fichier `prog/game_sdl.ml` est le suivant, il se contente de passer le nom des touches :

```
(* Main spécifique à SDL *)
let () = Game.run [| "left"; "right"; "up"; "down" |]
```

Le fichier `src/game.ml` commence par deux définitions de types, puis un ensemble de fonctions (à compléter). Pour ce premier TP, tout notre code ira dans ce fichier du code pour créer une fenêtre de 800x600, puis y tracer un rectangle noir de 200x200 pixels, et, ce qui n'est évidemment pas réaliste pour un programme de taille plus conséquente. Cet aspect d'organisation du code sera abordé à la séance 2. Le premier type est le suivant :

```
(* Les types des textures qu'on veut dessiner à l'écran *)
type texture =
  Color of Gfx.color
  | Image of Gfx.surface

let white = Color (Gfx.color 255 255 255 255)
```

C'est le type des « textures » que l'on veut dessiner à l'écran. Soit une couleur simple (de type `Gfx.color`) soit une image (de type `Gfx.surface`, que l'on peut voir comme un tableau de pixels). Le second type est le type `config` :

```

type config = {
  (* Informations des touches *)
  key_left: string;
  key_up : string;
  key_down : string;
  key_right : string;

  (* Informations de fenêtre *)
  window : Gfx.window;
  window_surface : Gfx.surface;
  ctx : Gfx.context;
}

```

Un tel type peut être vu comme représentant l'état global du jeu. C'est ici que l'on peut stocker des informations qui doivent être accessibles globalement, par exemple la fenêtre, les informations de configurations (chemin des fichiers ou autre), puis, plus tard, le niveau dans lequel on se trouve, les points de vies du personnage, ...

Nous allons enrichir ce type progressivement pour l'instant, il ne stocke que les noms des touches de directions et les informations relatives à la fenêtre principale.

4.2 Prise en main de la bibliothèque graphique

Pour toutes ces questions, vous devez avoir sous la main la documentation de la bibliothèque graphique, disponible sur la page du cours.

Pour toute la suite, on suppose que l'on teste la version JavaScript du programme.

1. éditer le fichier `src/game.ml` ajouter à la fonction `run` du code pour créer une fenêtre de 800x600, puis y tracer un rectangle noir de 200x200 pixels, aux coordonnées 100, 100.
2. modifier le code pour créer une fonction

draw_rect: config -> texture -> int -> int -> int -> int -> unit

qui dessine un rectangle arbitraire dans la fenêtre, de la couleur donnée (vous ignorer le cas des images pour l'instant). Vous devez récupérer le contexte graphique depuis l'objet de type `config` passé en argument.

3. modifier le code pour dessiner un rectangle de 200x200 aux coordonnées 100,100 60 fois par secondes. Pour cela, définir une fonction **update : config -> float -> unit option**. Cette fonction prend en argument l'objet de configuration, l'heure où elle est appelée en millisecondes et renvoie **None** pour l'instant. Votre fonction doit
 - effacer complètement l'écran (le peindre en blanc)
 - dessiner le rectangle demandé

Votre fonction **update** doit ensuite être appelée comme ceci dans **run** :

```

let cfg = { ... } in
Gfx.main_loop (update cfg) (fun () -> ())

```

Le fonctionnement est le suivant :

— On rappelle que **update cfg** est une application partielle, donc une fonction de type

float -> unit option

— Cette fonction est appelée 60 fois par secondes par **Gfx.main_loop**, tant qu'elle renvoie **None**

— Si à un moment donné la fonction renvoie **Some v** alors la deuxième fonction passée en argument (ici qui ne fait rien) est appelée sur **v**.

Ce fonctionnement sera utile dans la deuxième partie.

4. modifier le code pour que le rectangle alterne de couleur chaque seconde (Rouge, Vert, Bleu, Rouge, Vert, Bleu, ...). Pour cela, étendre le type **config** pour contenir trois nouveaux champs :

textures : texture array contenant trois couleurs

mutable current : int l'indice de la couleur courante

`mutable last_dt : float` l'heure (en millisecondes) à laquelle on a effectué le dernier changement de couleur

Dans la fonction **update**, utiliser ces informations (ainsi que le paramètre d'heure courante passé à **update**) pour mettre à jour la couleur du rectangle dessiné

5. modifier le code pour que le rectangle se déplace en pressant sur les flèches du clavier. Afin d'obtenir un comportement fluide, on souhaite capturer deux événements. Pour chaque touche, lorsqu'elle est enfoncée, mettre un booléen à **true**. Lorsqu'elle est relâchée, mettre ce booléen à **false**. Indépendamment de cela, si le booléen d'une touche de direction est **true**, déplacer le rectangle de 10 pixels dans cette direction. Il faut donc modifier le type **config** pour y stocker les coordonnées du rectangle, comme deux champs mutables **mutable x : int** et **mutable y : int**. On créera aussi une table de hachage global nous permettant de stocker le fait qu'une touche est enfoncée. Dans la fonction **update**
 - lire si un événement est disponible (**Gfx.poll_event**) et éventuellement stocker le nom d'une touche enfoncée ou retirer de la table celui d'une touche relâchée
 - parcourir les touches enfoncées et déplacer le rectangle dans la bonne direction

De plus, en fin de fonction **update**, tester si la touche **q** est enfoncée et si oui, s'arrêter, c'est-à-dire renvoyer **Some ()**

4.3 Chargement asynchrone de ressources

Une difficulté de programmation est le chargement asynchrone de ressource (fichier texte ou image). En effet, dans le contexte de l'exécution de code en JavaScript, le navigateur Web ne peut pas « bloquer » en attendant que le contenu d'une ressource soit disponible (en attendant que les octets arrivent du réseau par exemple, car le code s'exécute dans le navigateur, mais les ressources sont sur le serveur Web « distant »).

L'idée est donc de se servir de la boucle principale (**Gfx.main_loop**). Cette dernière a le prototype suivant :

```
Gfx.main_loop : (float -> 'a option) -> ('a -> unit) -> unit
```

Elle prend deux fonctions en argument. La première est appelée de façon répétitive en passant à chaque fois en argument le nombre de millisecondes écoulés depuis le début du programme. Si cette fonction renvoie **None** alors elle est appelée de nouveau. Si elle renvoie **Some v** alors la seconde fonction est appelée sur la valeur **v** (la seconde fonction est appelée « continuation » car elle effectue la suite du calcul, et ce style de programmation est appelé *continuation passing style*). Ainsi, si on voulait faire la chose suivante :

- Charger un fichier texte puis, quand celui-ci est chargé
- charger un second fichier texte puis, quand celui-ci est chargé
- faire la somme de leur longueur et l'afficher dans la console de débogage.

Le premier code-ci dessous est **incorrect** :

```
let res1 = Gfx.load_file "fichier1.txt" in
let txt1 = Gfx.get_resource res1 in (* <-- Exception levée ici *)
let res2 = Gfx.load_file "fichier2.txt" in
let txt2 = Gfx.get_resource res2 in
Gfx.debug "%d\n%!" (String.length txt1 + String.length txt2)
```

Le code ci-dessus lèvera une exception à l'endroit indiqué, car il est possible que le fichier ne soit pas complètement chargé. Le second code ci-dessous est aussi **incorrect**

```
let res1 = Gfx.load_file "fichier1.txt" in
while not (Gfx.resource_ready res1) do () done;
let txt1 = Gfx.get_resource res1 in
let res2 = Gfx.load_file "fichier2.txt" in
while not (Gfx.resource_ready res2) do () done;
let txt2 = Gfx.get_resource res2 in
Gfx.debug "%d\n%!" (String.length txt1 + String.length txt2)
```

Le code ci-dessous fait de l'attente active, et ce faisant, monopolise le navigateur sans lui rendre la main. Ce dernier n'a donc pas l'opportunité de changer l'état interne de la ressource et la première boucle est en réalité une boucle infinie. Le code correct est :

```

1 let res1 = Gfx.load_file "fichier1.txt" in
2 Gfx.main_loop
3   (fun _dt -> Gfx.get_resource_opt res1)
4   (fun txt1 ->
5       let res2 = Gfx.load_file "fichier2.txt" in
6       Gfx.main_loop
7         (fun _dt -> Gfx.get_resource_opt res2)
8         (fun txt2 ->
9             Gfx.debug "%d\n%!"
10              (String.length txt1 + String.length txt2)))

```

La première fonction (ligne 3) est appelée de façon répétée, tant que `Gfx.get_resource_opt` renvoie `None`. Quand cette dernière renvoie `Some txt1` (le contenu du fichier), cette valeur est passée en argument à la deuxième fonction (ligne 4). Cette dernière peut alors faire de même pour le deuxième fichier. Quand ce dernier est disponible (la fonction ligne 7 renvoie `Some txt2`), son contenu est passé à la dernière fonction (ligne 8) qui a alors à sa disposition les deux contenus.

1. Sur le modèle ci-dessus, modifier votre fonction `run` pour
 - charger le fichier texte `resources/files/tile_set.txt`, ce dernier contient sur 3 lignes les noms de 3 fichiers d'images
 - pour chacun des fichiers d'images `f`, charger l'image `resources/images/f`
 - lorsque toutes les images sont disponibles, s'en servir pour créer le tableau `textures` de l'objet de type `cfg`
2. Ajouter à la fonction `draw_rect` le cas pour dessiner une image et tester le programme