Approfondissement : Shoot 'em up en OCaml (Semaine 1)

A lone some hacker

Les trois prochaines semaines seront dédiées à la réalisation d'un mini jeu vidéo. Ce projet vous permettra d'appréhender les bases du développement d'un jeu vidéo.

Grossièrement, un jeu vidéo se résume au code suivant

```
let rec run state =
  if not (continue state) then
     ()
  else
    let state' = update state in
    display state';
    run state'
```

```
while !continue do
  update state;
  display state
done
```

Les prochaines semaines permettront de plonger au sein des différentes expressions de cette boucle. On abordera notamment les notions liées au temps, au modèle du jeu, à l'interface avec le joueur et au rendu.

1 Présentation

On vous propose de développer un *shoot 'em up*[1], où l'objectif principal sera de rester en vie tout en détruisant les ennemis s'attaquant à vous.

1.1 Qu'est-ce qu'un modèle de jeu ?

Un modèle consiste à structurer l'état d'un jeu vidéo. Imaginez que vous souhaitiez développer un jeu de combat. Comment faire la différence entre le fait de s'être fait toucher à la tête et avoir paré une attaque à l'aide de ses bras ? Un modèle bien conçu vous permettra de faire une telle distinction. De plus, tester la collision entre un bras et une jambe est une opération complexe nécessitant de définir ce que sont une jambe et un bras. Un modèle apporte une abstraction et ramène ce problème à celui de la collision entre deux figures géométriques bien connues : des polygones pour un monde en deux dimensions et des polyèdres pour un monde en trois dimensions. Cette vision est un peu simpliste mais permet d'avoir une idée de ce qui se passe en arrière-plan d'un jeu. Par exemple, si l'on prend le jeu Mario, on s'aperçoit que les images utilisées sont relativement rectangulaires. Ainsi, on peut tout à fait considérer un ensemble de personnages comme un ensemble de rectangles. La détection des collisions revient à tester le chevauchement de deux rectangles.

1.2 Gestion d'un périphérique

Contrairement à ce que l'on pourrait imaginer gérer un périphérique matériel est une tâche ardue. Par exemple, un joystick classique vous renvoit de manière continue l'axe le long duquel vous vous déplacez ainsi qu'une valeur comprise entre -32768 et 32767 pour déterminer l'inclinaison. Ici, nous nous limiterons, dans un premier temps, à la gestion du clavier.

Le principal problème que nous chercherons à résoudre est la gestion de l'appui continu sur une touche car les différentes bibliothèques sur lesquelles nous reposons ne signalent un évènement que lorsqu'une touche est enfoncée ou relâchée.

1. jeu de tir

1.3 Rendu de la scène

Le rendu à l'écran est important pour l'utilisateur. Néanmoins le faire proprement est une opération complexe. Certains effets visuels sont liés à certaines technologies matérielles. C'est notamment le cas du *tearing*[2]. Cet effet apparait lorsque les fréquences du flux vidéo et de rafraichissement de l'écran ne sont pas synchronisées. Si le flux vidéo possède une fréquence plus rapide, il est possible que deux images se superposent à l'écran. Plusieurs solutions ont été apportées à ce problème. Parmi celles-ci, on retrouve l'utilisation de plusieurs buffers, chacun correspondant à une image écran. On ajoute également un mécanisme de verrouillage empêchant tout accès à la zone mémoire correspondant à celle en cours d'affichage. Ainsi, la prochaine image se prépare en arrière plan sur l'un des buffers non projetés. Une autre solution consiste à autoriser l'écriture sur la zone mémoire en cours d'affichage seulement aux endroits qui ont déjà été projetés.

Fort heureusement, la SDL[3], à condition que le matériel dont vous disposez en ait les capacités, gère, pour nous, cette synchronisation.

1.4 Le temps

Pour finir, le temps. En général, il est difficile de gérer le temps en informatique. Dans notre cas, cette gestion nous permettra de contrôler le nombre d'images par seconde affichées à l'écran. Pourquoi gérer le temps alors qu'il suffit de laisser la boucle d'exécution faire son travail ? Tous les ordinateurs ne disposent pas de la même puissance de calcul. Ainsi sur un ordinateur puissant, la mise à jour de l'état du jeu ira bien plus vite que sur un ordinateur moins puissant. Cette différence fait que le joueur utilisant le pc puissant va plus vite que l'autre. On va donc faire en sorte que les deux ordinateurs affichent le même nombre d'images par seconde. Cette technique est relativement naïve et l'on pourra discuter des améliorations possibles. Notamment celles basées sur la notion d'interpolation.

2 Développement d'un modèle

Ce modèle nous permettra de représenter l'état du jeu vidéo à tout instant. Finalement, ce que l'utilisateur verra à l'écran ne sera qu'une interprétation possible du modèle. On pourrait se contenter de ne changer que les images et de changer le nom du jeu pour créer un nouvel opus. Cette pratique est courante dans l'industrie du jeu vidéo.

2.1 Les bases

La première étape consiste à développer les outils géométriques sur lesquels nous allons baser le modèle. Pour commencer, vous devez créer le fichier geometry.ml[4] muni de la signature suivante

```
module Vector : sig
  type t

  val create : float -> float -> t
end

module Point : sig
  type t

  val create : float -> float -> t
  val move : Vector.t -> t -> t
end
```

Questions : Pourquoi le type t est-il défini de manière abstraite ? Quelle est l'utilité de la fonction create ?

2. déchirement en français
3. http ://www.libsdl.org/
4. Un fichier est un module en OCaml

2.2 Représentation des différents objets

Un *shoot 'em up* est généralement composé des éléments suivants
— un terrain,
— un vaisseau pour le joueur,
— de vaisseaux ennemis,
— d'armes en tout genre,
— d'un fond étoilé (ici, il faudra de l'imagination).

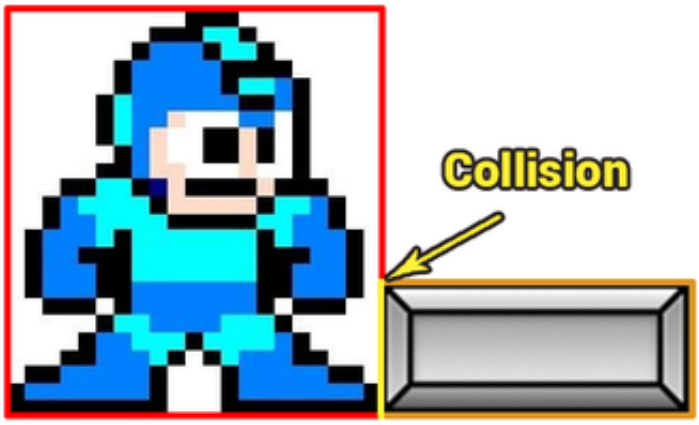
La difficulté de programmation d'un tel jeu provient, principalement, de la détection de collision entre ces différents objets. Un modèle correctement conçu doit rendre cette opération faisable et, si possible, efficace.

Question : Quelle figure géométrique simple pourrait représenter chacun des objets précédents ?

Dorénavant, nous ne parlerons plus de vaisseaux ou de joueur mais simplement de figure géométriques. Toutes les actions du jeu doivent pouvoir s'exprimer à travers ce modèle. Cette figure va nous servir de boite[5]. On raisonne maintenant avec des boites. La détection de collision entre deux objets revient à détecter le chevauchement entre deux boites. Maintenant, vous devez créer un fichier boundingBox.ml dont la signature est

```
module Circle : sig
  type t

  val create : Geometry.Point.t -> float -> float -> t
  val move : Geometry.Vector.t -> t -> t
  val collide : t -> t -> bool
  val collide_with_any_of : t -> t list -> bool
end
```



3 La gestion du clavier

Pour pouvoir interagir avec le jeu, il est nécessaire de prendre en compte au moins un périphérique de l'ordinateur. Dans notre cas, ce sera le clavier.

3.1 Les évènements

Pour développer ce jeu, on se base sur la SDL. C'est une bibliothèque nous offrant un certain nombre de primitives permettant, entre autres, de dessiner à l'écran, gérer des périphériques de jeu (clavier, souris, manette, ...), jouer du son...La gestion des périphériques se fait à travers des évènements. Nous ne nous intéressons qu'à ceux nécessaires à la gestion du clavier. Ils sont définis dans le module

5. En anglais, on parle de bounding box

Sdlevent de la Sdl, KEYUP et KEYDOWN. Ces évènements contiennent le symbole de la touche correspondante. Ces symboles sont représentés par le type Sdlkey.t.

3.2 Gestion de l'appui continu sur une touche

L'inconvénient principal des évènements pour la gestion du clavier est qu'ils ne se produisent que lors d'un changement d'état. Or l'appui sur une touche de manière continue ne modifie en aucun cas l'état.

Solution proposée : utilisation d'une structure de données permettant d'associer une touche à son état (enfoncée ou non). Ainsi, on crée une indirection entre les évènements et la gestion des touches proprement dite. On va chercher à confiner cette gestion dans un fichier events.ml qui à travers une fonction get_keys nous renverra la liste des touches enfoncées à un moment donné. Ceci nous permettra également de gérer l'appui simultané et ainsi de permettre au joueur le déplacement en diagonal. Le fichier events.ml doit avoir la signature suivante :

```
type key

val get_keys : unit -> Sdlkey.t list
val update : Sdlevent.event -> unit
val updates : Sdlevent.event list -> unit
```

On ne prendra en compte que les touches suivantes
— flèches directionnelles,
— echap (*escape*),
— espace.

TO BE CONTINUED

— déplacement du vaisseau dans l'espace,
— génération d'ennemis,
— gestion des collisions,
— intelligence artificielle,
— pattern de tirs ennemis,
— son,
— mode multijoueurs,
— ...