Let's move on !

A lone some hacker

1 Previously in LI220

La semaine dernière, nous avons vu les bases d'un modèle de jeu. Les différents objets sont vus comme des boites sur lesquelles différentes opérations sont disponibles.

2 Utilisation générale de la SDL

La SDL est une bibliothèque multi-plateforme distribuée sous la licence zlib. Elle fournit un ensemble de services dont la vidéo, la gestion de certains périphériques, le son, le temps et l'affichage de texte.

La principale notion liée à l'affichage graphique est celle de *surface*. Tout élément graphique manipulable par la SDL est une surface que ce soit l'écran ou le cercle que vous cherchez à dessiner.

L'initialisation de la SDL vous donne une surface de base représentant l'écran. Chaque fonction de dessin de la SDL vous renvoit une surface. Comment afficher quelque chose à l'écran ? En fusionnant la surface représentant l'objet que vous souhaiter afficher avec la surface de l'écran. Cet opération est connue sous le nom de *blitting*. On dit qu'on *blit* une surface sur une autre.

L'image disponible à la racine de l'archive du projet est représentée par une surface. Pour afficher les différents éléments du jeu, on *blitte* des sous-parties de celle-ci. On appelle ce genre d'images des *sprites*.

Dans ce projet, nous utilisons la méthode dite du *double buffering*. Ainsi les différents opérations de *blitting* sont effectuées sur un buffer qui n'est pas affiché à l'écran. C'est à vous d'appeler la fonction flip qui permet de passer le *buffer* en arrière-plan au premier plan, et ainsi d'afficher à l'écran la nouvelle scène.

Dans un jeu vidéo classique, on affiche environ 60 *frames per seconds* (ce qui correspond à la fréquence de rafraîchissement de votre écran). Cela signifie que la fonction flip est appelée 60 fois par seconde !

La gestion des périphériques est liée à la gestion des évènements. Ceux-ci nous sont fournis par la SDL. En arrière-plan, un programme écoute les différents périphériques et renvoit les évènements lorsque la fonction SDLEvent.get est appelée.

3 Objectif

Permettre au joueur de se déplacer pour éviter les ennemis arrivant vers lui. Pour cela, vous devez utiliser le module que vous avez développé la semaine dernière pour la gestion du clavier.

4 Module G

Pour éviter de perdre du temps sur l'apprentissage de la SDL, un module, G, est mis à votre disposition. Il réalise pour vous les différentes opérations liées à l'affichage.

Lorsque vous devrez créer le fichier game.ml, celui-ci devra impérativement contenir la ligne suivante après les différentes ouvertures de modules

```
open Graphics

module G = (val (init ()) : G)
```

Ceci vous permettra d'utiliser le module G au sein du fichier.

Celui-ci vous propose les fonctions suivantes
— player,
— bullet,
— ennemy,
— star,
— score,
— flip,
— clean,
— ticks,
— delay,
— press_escape_to_continue
— win,
— game_over

Ce module permet de «cacher» les détails liés à la SDL [1]. Vous pouvez donc afficher le joueur, une balle, un ennemi, une étoile ou encore le score.

5 Ouvrir une fenêtre

Avant de commencer à coder le jeu à proprement parler, on va commencer par ouvrir une simple fenêtre dans laquelle, on retrouve le vaisseau du joueur. Cette étape nous permet de passer en revue quelques fonctions du module fourni et d'aborder la compilation séparée.

Une erreur courrante consiste à appeler la fonction permettant d'ouvrir une fenêtre et de vouloir directement voir le résultat. Le problème d'appeler cette fonction sans rien faire derrière est que le programme atteint la fin de son exécution rapidement. Ceci entraine une ouverture et une fermeture quasi-immédiate de la fenêtre.

Il faut donc introduire une «attente» derrière l'ouverture de la fenêtre. C'est ici que la fameuse boucle de jeu commence son intervention. C'est elle qui va nous permettre de progresser. Souvenez-vous cette boucle n'avait que deux actions, mettre à jour l'état du jeu et afficher la résultat à l'écran.

Dans un premier temps, on considère la fonction update équivalente à la fonction identité.

Créer un fichier nommé game.ml dans lequel se trouve une fonction update équivalente à l'identité et une fonction display vide pour le moment.

Créer maintenant une fonction *play* dont le corps appelle *update* et *display* en boucle.

Pour pouvoir afficher l'état du jeu, il faut décider de ce que contient l'état. Pour le moment, l'état a le type suivant

```
type state =
  { p : BoundingBox.Circle.t }
```

La fonction display doit afficher l'état puis *flipper* la scène.

1. Pour ceux qui sont intéressés les sources sont disponibles.

6 Compilation séparée

Ce projet commence à ressembler à un vrai programme. Celui-ci dispose de plusieurs fichiers et repose sur plusieurs bibliothèques (la bibliothèque standard du langage OCaml et la SDL).

Pour créer le programme, il faut compiler chacun de ces fichiers puis les assembler avec les bibliothèques nécessaires. Chaque fichier est ce qu'on appelle une unité de compilation. Le compilateur OCaml prend en entrée un fichier source et produit un fichier objet. L'assemblage de l'ensemble des fichiers objets est réalisé par un éditeur de liens.

L'option -c permet de compiler un fichier source en un fichier objet. Si vous passez une liste de fichiers objets en paramètre (sans l'option -c), l'éditeur de liens produit un exécutable nommé a.out. Pour nommer cet exécutable différemment, vous pouvez utilisé l'option -o suivi du nom, lui-même suivi de la liste des fichiers objets.

Pour réaliser cela, nous allons utilisé l'outil ocamlfind disponible dans les dépots OPAM. Cet outil permet de travailler avec les différentes bibliothèques OCaml sans avoir à gérer soi-même les chemins vers les fichiers correspondants.

```
ocamlfind ocamlc -c geometry.ml
ocamlfind ocamlc -c boundingBox.ml
ocamlfind ocamlc -package sdl -c events.ml
ocamlfind ocamlc -package sdl -c game.ml
ocamlfind ocamlc -package sdl -linkpkg -o game geometry.cmo
          boundingBox.cmo events.cmo game.cmo
```

Attention, l'éditeur de lien OCaml n'est pas très fûté, l'ordre des fichiers est important. Si un fichier A dépend de B, B doit apparaître avant !

7 Let's do this !

L'ensemble des actions du jeu dépendent des évènements produits par le joueur. Ces évènements manifestent l'appui de différentes touches. Les fonctions permettant de mettre à jour l'état du jeu vont donc dépendre de ces touches enfoncées. Ces touches se divisent en deux catégories, une première concerne l'application de manière générale et une seconde intervient directement au niveau du jeu.

Voici les fonctions que vous devez implémenter

```
move_player : Sdlkey.t -> state -> state
update : Sdlkey.t -> state -> state
updates : state -> state
```

La fonction *update* permet de gérer les différentes catégories de touches. Si une touche liée au déplacement du joueur est enfoncée alors on appelle la fonction *move_player*. Sinon on effectue le reste des actions prévues. Si une touche enfoncée n'est pas une touche gérée par le jeu, on choisit de l'ignorer.

updates est une fonction qui récupère l'ensemble des touches enfoncées (souvenez-vous du module *Events* de la semaine dernière) et appelle *update* sur chacune des touches.

À cette étape, le vaisseau devrait commencer à se mouvoir !

Attention, le repère orthogonal de la SDL n'est pas celui qu'on dessine habituellement sur papier !

Pour observer le déplacement de votre vaisseau, vous devez mettre à jour la fonction *display* de sorte à ce qu'elle affiche le vaisseau à l'aide du module *G*.

8 Gestion des tirs

Actuellement, l'état du jeu permet de prendre en compte le joueur et ses déplacements. Pour ajouter un peu de piment au jeu, il serait bon d'ajouter la possibilité de tirer des missiles. Pour cela, il est nécessaire de modifier l'état du jeu pour mémoriser l'ensemble des balles tirées. Sachant que l'opération principale liée à ces balles est la détection de collision avec les ennemis, on les représentera par des *bounding boxes*.

Attention, que se passerait-il si on ne faisait que tirer ? Chaque tir ajoute un missile à la liste des missiles mais aucune action n'est prévue pour retirer des missiles de cette liste. Ainsi, elle ne ferait que croître, créant une fuite mémoire importante.

Pour pouvoir faire le tri entre les missiles encore utiles de ceux qui ne le sont plus, il faut introduire la notion de terrain. Un missile quittant le terrain sans avoir touché un ennemi est tout simplement retiré de la liste des missiles.

Comment représenter le terrain ?

Modifier le type *state* en prennant en compte ces nouveaux éléments. Les missiles se déplacent à une certaine vitesse ! La fonction de mises à jour de l'état fera en sorte de déplacer l'ensemble des missiles tirés à chaque tour !

Il est temps d'introduire des adversaires à votre hauteur !

9 Gestion des ennemis

Cette étape nous permettra d'aborder l'un des derniers concepts nécessaires à la réalisation d'un mini jeu vidéo, la génération de nombres «aléatoires».

Produire des nombres aléatoires est difficile car un ordinateur est une machine prévisible. Lorsqu'une machine produit, pour une même séquence d'opérations, un même résultat, on dit qu'elle est déterministe.

Pour produire de tels nombres, on utilise des suites mathématiques et une graine. Cette graine va apporter le caractère non-prévisible du générateur de nombres «aléatoires».

Comment fait-on pour obtenir une telle graine ?

Plusieurs solutions sont possibles. Voici quelques informations utilisées par le noyau Linux pour en générer une

— intervalle de temps entre différents appuis de touche,

— intervalle de temps entre les interruptions systèmes,

— ...

Mélanger ces différentes sources permet d'obtenir une information difficilement prévisible par un attaquant.

Néanmoins, Un problème persiste. Que se passe t-il au démarrage de votre ordinateur ? La séquence de démarrage étant bien souvent identique, il se pourrait qu'on obtienne finalement toujours la même information. Pour éviter cela, une certaine quantité d'information est sauvegardée à l'extinction de votre ordinateur et restaurée au démarrage.

OCaml propose le module *Random* pour générer des nombres de manière pseudo-aléatoire. Pour un jeu vidéo, la source permettant de générer la graine n'est pas très importante. On se contente de celle fournit par le système, c'est-à-dire *Random.self_ init*.

Vous disposez alors de toutes les fonctions du module *Random* dont notamment *int* et *float* qui permettent de générer respectivement un entier et nombre flottant.

On représente les ennemis par des *bounding boxes* et c'est à vous d'imaginer une stratégie de génération d'ennemis. La stratégie doit être conçue dans un **fichier séparé que l'on nommera strategy.ml.**

10 Le temps

Pour le moment, le nombre de *frames per seconds* n'est pas géré. Ainsi, on se retrouve dans les travers soulevés par le premier sujet.

Le module *G* vous offre deux fonctions liées aux opérations permettant de retrouver un peu de contrôle. *ticks* permet d'obtenir le nombre de *ticks*[2] écoulées depuis le lancement de l'application. *delay* permet d'arrêter votre programme un certain nombre de milli-secondes.

Votre boucle principale doit donc mesurer le temps qui s'écoule et arrêter votre programme si celui-ci est en avance sur le nombre de *frames* à afficher par seconde. On ne s'occupera pas du retard potentiel que votre application pourrait prendre. Dans ce cas, on obtiendra un effet de *lag* mais on ne cherchera pas à y remédier.

2. Ce sont des milli-secondes