Rapport
Arène Vectorielle Synchrone

Julien Bissey
Kelly Seng

15 avril 2019

1 Introduction

Ce projet est divisé en deux parties, Serveur et Client. Le serveur a été codé en OCaml, le client en Java. Un manuel d'utilisation est disponnible dans le dossier doc. Le dossier src contient les sources du client et du serveur.

Concernant les extensions :

- le modèle physique est basé sur des chocs élastiques.

2 Serveur

Le serveur est composé d'une thread principale lançant les autres threads, une thread recevant les nouvelles connections et associant les nouveaux clients à une nouvelle thread, une thread par client qui reçoit les requêtes de ce client et y répond si nécessaire, et enfin une thread qui contrôle le déroulement du jeu, met à jour les positions des différents objets, et en informe les clients.

On peut refuser la connection à un client pour deux raisons : soit le serveur gère déjà le nombre maximal de joueurs, soit le client veut utiliser un pseudo déjà pris par un autre joueur.

On conserve tous les objets de l'arène dans un unique tableau dans le module Arena. On a choisi un tableaux car on a souvent besoin d'accéder à un élement particulier de la structure (pour exécuter les requêtes de type NEWCOMS notamment).

On conserve également les joueurs dans un tableau pour les mêmes raisons. De plus, on associe aux joueurs les channels de leur client :

val players : ((in_channel * out_channel) * Player.t) array

Cela nous permet d'avoir facilement accès à tous les channels dans la thread "game" qui se charge d'envoyer les messages à tous les clients. On utilise une fonction avec un argument optionel pour envoyer un message à tous les clients si l'argument n'est pas donné :

```
let message ?(id = (-1)) cmd =
match id with
|(-1) -> List.iter (fun x -> output_string (snd (fst x)) (Command.FromServer.to_string
cmd)) (real_players ())
|id -> output_string (snd (fst players.(id))) (Command.FromServer.to_string
cmd)
```

Toutes les constantes utilisées par le serveur sont situées dans le module Values, on peut donc facilement les modifier.

Concernant les ajouts au formulaire :
Lorsqu'un nouveau joueur se connecte à une partie déjà en cours, en plus du WELCOME, on lui envoie également un SESSION et un NEWOBJ pourqu'il ait immédiatement connaissance des autres joueurs et de l'objectif actuel.

Comme les chocs élastiques font se déplacer les obstacles, il faut pouvoir les identifier lorsqu'on donne leurs nouvelles coordonnées aux clients. On remplace donc le message ocoords des requêtes WELCOME et SESSION par coords (un nom en plus des simples coordonnées) et on ajoute un autre message vcoords à TICK. Pour annuler ces changements et respecter la norme du formulaire, il faut lancer le serveur avec l'option -comp. Le modèle physique sera alors une inversion du vecteur vitesse d'un vaisseau lors d'une collision avec un obstacle.

En ce qui concerne la concurrence, le type objet contient un mutex, ce qui permet de maximiser la concurrence, contrairement à un mutex unique que l'on aurait pu placer dans le module Arena.

```
(type t = mutable id : int; mtx : Mutex.t;
mutable coord_x : float; mutable coord_y : float;
mutable speed_x : float; mutable speed_y : float;
mutable angle : float; mass : float; radius : float)
```

Afin de toujours attendre la bonne durée entre chaque tick de calcul de position, on encadre les calculs par des appels de temps d'exécution :

```
let start = Sys.time () in
...
let wait_time = 1. /. Values.server_tickrate -. (Sys.time () -. start) in
(if (wait_time > 0.)
then Thread.delay (wait_time)
```

else print_endline "please decrease Values.server_tickrate");

On utilise la même méthode pour le client.

3 Client

Le client est composé d'une thread principale lançant les autres threads, une thread déplaçant les objets de l'arène indépendament du serveur, une thread recevant les messages du serveur, modifiant les données du jeu en conséquence, et affichant les informations envoyées par le serveur dans le terminal (arrivée et départ de nouveaux joueurs, et scores notamment), et enfin une thread consacrée à l'interface graphique du client.

La thread déplaçant les objets s'occupe uniquement des mouvements et des collisions, elle ne s'intéresse donc pas à l'objectif, le serveur seul décide si un joueur gagne un point ou non.

L'interface graphique est composé d'une unique fenêtre affichant l'arène. A part pouir l'objectif, nous avont utilisé des polygônes pour représenter les différents objets de l'arène. Puisque tous les objets ont une hitbox en forme de cercle, nous avons utilisé des polygônes ayant une forme proche d'un cercle.

Pour les vaisseaux : le polygône lie 3 points du cercle formant un triangle, ainsi que le centre du cercle, cela donne une forme de flèche permettant de facilement repérer l'orientation du vaisseau.

```
public synchronized int[][] getPaintDataPolygon() {
int [][] data = new int[2][4];
data[0][0] = (int) (posX + radius * Math.cos(angle + 2.  * Math.PI / 3) +
Arena.half_width);
data[0][1] = (int) (posX + radius * Math.cos(angle) + Arena.half_width);
data[0][2] = (int) (posX + radius * Math.cos(angle - 2.  * Math.PI / 3) +
Arena.half_width);
data[0][3] = (int) (posX + Arena.half_width);
data[1][0] = (int) (posY + radius * Math.sin(angle + 2.  * Math.PI / 3) +
Arena.half_height);
data[1][1] = (int) (posY + radius * Math.sin(angle) + Arena.half_height);
data[1][2] = (int) (posY + radius * Math.sin(angle - 2.  * Math.PI / 3) +
Arena.half_height);
data[1][3] = (int) (posY + Arena.half_height);
return data;
}
g2D.fillPolygon(player_ship[0], player_ship[1], 4);
```

Pour les obstacles : un polygône de 12 points choisis aléatoirement, de tel sorte

qu'ils soient suffisamment proches du cercle pour que leur forme soit en accord avec leur hitbox tout en étant irréguliers. De plus, on fait tourner chaque obstacle dans un sens et à une vitesse aléatoires. On a donc besoin d'ignorer la valeur d'angle envoyée par le serveur à chaque TICK (on pourrait ne pas l'envoyer, mais cela forcerait à traiter les vaisseaux et les obstacles encore plus différement pour un gain de performance négligeable).

Pour quitter le jeu, il suffit de fermer la fenêtre. Cela n'envoie pas de message EXIT au serveur (de fait, le client tel qu'on l'a écrit n'envoie jamais de message EXIT), mais le serveur a été fait de telle sorte qu'il considère un client qui ne répond plus comme un client sortant du jeu de manière contrôlée.

De même, si le serveur s'arrête de manière imprévue, le client va en informer le joueur dans le terminal et quitter le jeu.

En ce qui concerne la concurrence, les méthodes de la classe Object sont synchronized, ce qui donne le même résultat que pour le serveur.