M1 Informatique - Projet de PC2R

RAPPORT

Arènes Vectorielles Synchrones

Etudiants
Suxue Li
Julien Xavier

15 avril 2019

Table des matières

1 Introduction

Le devoir de programmation de cette année a pour objectif de réaliser une application de jeu clients-serveur où les joueurs jouent en concurrence dans un jeu de course dans l'espace, représenté par une arène torique.
Les parties s'organisent en sessions, les joueurs doivent atteindre des points particuliers nommés *objectifs* pour gagner une session. Des extensions ont été ajoutées par rapport à la version de base, vous trouverez une description plus complète dans le manuel du jeu à la fin de ce rapport. Pour ce projet, nous avons fait le choix d'implémenter la partie serveur en **Ocaml** et la partie client en **Java**.
Vous trouverez dans ce rapport, une présentation globale de l'architecture du projet, nos choix d'implémentation et les difficultés que nous avons rencontré au cours du développement. Enfin, un petit manuel de jeu vous guidera pour commencer à jouer. Et pour terminer, quelques remarques que nous nous sommes faites nous-même.

2 Architecture générale du projet

2.1 Principe de communication

Pour réaliser la communication entre le serveur et les clients, nous avons utilisé un système de communication par socket, ce qui nous permet de réaliser des sessions TCP, afin d'envoyer des informations dans les deux sens de manière fiable.
La communication est basé sur le protocole texte donné dans le sujet. Le serveur utilise des canaux du type in_channel et out_channel, et le client utilise la classe BufferedReader et PrintStream.

2.2 Serveur

Nous avons préféré utiliser le côté fonctionnel du language Ocaml pour implémenter notre serveur. Le fichier **server.ml** contient tout le code du serveur. Toutes les valeurs constantes ont été définies au début du fichier, entre autres, une variable *current_session* qui centralise toutes les données de la session en cours, notamment les joueurs qui sont présents, avec toutes les informations qui les accompagnent.
Nous avons un point d'entrée principale dans le fichier : la fonction *start_server* qui lance le serveur et écoute sur le port 2019. Mais avant cela, il lance deux threads : *start_session* et *server_refresh_tick_thread* qui permettent respectivement, d'attendre la connexion d'au moins un joueur puis de patienter un délai avant de lancer le jeu avec les joueurs présents, et, de faire les actualisations nécessaires des positions de chaque joueur *server_refresh_tickrate* fois par seconde.
Pour chaque client qui se connecte, un thread est lancé. Nous nous basons sur le partage des ressources code et données permis par le système de threading en Ocaml. En effet, une fois lancé, un thread est autonome et effectue le code qui lui est attribué, tous les threads ont alors accès aux mêmes fonctions de service, et aux mêmes données, en l'occurence *current_session*. Il était important de conserver l'intégrité du contenu de la variable. Nous avons donc utilisé un seul Mutex, et une

seule Condition. Le mutex s'appelle *mutex_players_list*, il est là pour limiter l'accès à la liste des joueurs de la session en cours, pour éviter toute lecture - écriture, et écriture - écriture simultanée. Nous avions plusieurs mutex au début du projet, mais nous les avons retiré au fur et à mesure de l'avancement car cela devenait compliquer à gérer. Pour cause, l'utilisation de plusieurs mutex impliquait des lock et unlock imbriqués, chaque fonction ayant son lot de modifications à faire sur les données, et les appels pouvant s'imbriquer, on arrivait facilement à des inter-blocages. Nous avons donc opté pour la sûreté par rapport à l'efficacité. Et enfin la Condition *least1player* est signalée à chaque connexion de joueur, permettant au thread qui lance la session de démarrer un compte à rebours.

2.3 Client

Pour la partie client, nous avons fait une structure qui s'inspire de l'architecture MVC où le Modèle serait toutes les classes de représentation de données de session, rangé dans le sous-répertoire *data* telles que **Ship**, **Player** ou encore **Point** etc. Et la Vue et le Contrôleur seraient fusionnés au niveau des classes **SpaceRun**, **Drawer**.
Nous utilisons JavaFX pour l'interface graphique du client. Le squelette de l'interface graphique est produit à l'aide d'un logiciel tierce : *SceneBuilder* qui nous permet de construire et d'agencer une fenêtre (scene) en "Drag&Drop". Ceci génère un fichier main.fxml qui décrit les différents éléments JavaFX composant la fenêtre de jeu.
La classe SpaceRun gère beaucoup de chose, dont l'interface utilisateur. Elle utilise les instances de éléments JavaFX à partir du fichier *main.fxml*. C'est donc la classe permettant de lancer l'application de jeu côté client, elle constitue le processus principal du client. Trois autres threads s'exécutent en concurrence : Receive, AnimationTimer et SendNewCom. La classe **Receive**, étant Runnable est constamment en attente de protocoles du serveur, elle récupère les protocoles reçues et fonction de celle-ci, délègue le traitement à la méthode associée d'une instance de SpaceRun. La classe **SendNewCom** gère l'envoi des nouvelles commandes du client depuis le dernier tick, à une fréquence de *server_tickrate*. La classe **AnimationTimer**, permet de rafraichir l'interface graphique, avec une fréquence de 60, qui normalement, selon le sujet devrait être à *refresh_tickrate*, mais nous n'avons pas trouvé le moyen de changer la fréquence de rafraîchissement d'un AnimationTimer.
Ces threads accèdent à des ressources en lecture, il n'y a donc pas besoin de les synchroniser.
Enfin, pour faire l'affichage de l'espace de jeu, nous utilisons une classe **Drawer** qui permet à l'aide du contexte graphique du canvas, passé en argument du constructeur, de dessiner chacun des éléments du jeu, qu'il s'agisse des véhicules, lasers, bananes/pièges, fond d'écran ou encore les astéroïdes.

3 Extensions

Jeu de Combat :
Les vaisseaux peuvent jeter des bananes derrière eux afin de perturber les déplacements de l'ennemi ou de soi-même s'il touche lui aussi la banane lors de ses déplacements. Chaque vaisseau a un quota de banane disponible au début d'une session, le serveur lui accorde une banane en plus pour chaque point supplémentaire acquis. Ce quota est réinitialisé à chaque début de session. Les vaisseaux peuvent aussi tirer des boules de laser afin de stopper l'ennemi dans son élan (vecteur

vitesse nul). Les boules de laser évoluent de même la même façon dans un monde thorique, et elles disparaissent au contact d'un astéroïde (obstacle).

Jeu de Course :
On ajoute un mode de jeu course, les joueurs doivent parcourir n (donné en paramètre au lancement de SpaceRun) points de contrôle (checkpoints) dans un ordre prédéfini. Dans ce mode, les objectifs sont numérotés, le prochain que le joueur doit toucher est numéroté en vert, le premier joueur qui parcourt tous les objectifs remporte le tour, et voit son score incrémenter d'un point. Comme sur la version de base, la session s'arrête quand un joueur atteint *wincap* points.

Chat :
Un joueur peut envoyer deux types de messages. Un message public, que tout le monde reçoit et qui s'affichera en noir dans le chat sous la forme "<source»<message> pour les autres clients et « you>message » pour celui qui l'a envoyé. L'envoie s'effectue en écrivant simplement dans la zone de saisie du chat puis en appuyant sur le bouton SEND. Un messages privé sera cependant, de deux couleurs différents : pour les messages privés entrants, ils s'affichent en orange sous la forme « dm :<source»<message> », et pour les messages privés sortants, ils sont en vert et sous la forme « to :<destinataire»<message>".
Pour envoyer un message privé, il faudra écrire dm/<destinataire>/<message> dans la zone de text et appuyer sur le bouton SEND. Attention, on ne peut pas envoyer de message privé à soi-même ou à un joueur non-connecté et cela sera notifié par un message en rouge dans la zone de texte, le texte d'erreur disparaît lorsqu'on clique sur le champs de saisie.
La zone de chat nous sert également d'affichages systèmes (rouge) : par exemple, à la connexion d'un nouveau joueur, les joueurs présent recevront une notification sous forme de message dans la zone de chat. De même lorsque un gagnant est déclaré, la zone du chat affiche le gagnant de la session.

4 Manuel du jeu

Prérequis technique :
La version la plus récente d'OCaml est requise pour lancer le serveur (version 4.07.0).
La version minimale recommandée de Java supportant notre client est Java 8.

Démarrage :
A la racine du projet, le répertoire "server" contient le code source du serveur. Vous trouverez également un Makefile permettant de compiler vers un exécutable « server », et pour lancer cet executable : **"./server <nombre d'objectif> <nombre d'obstacle>".** Le mode de jeu dépend du premier argument : *./server 1 5* lance le serveur en mode de jeu normal, *./server n 5* (n > 1) lance le serveur en mode de jeu course avec n objectifs, et 5 obstacles d'après l'exemple. De même dans le répertoire « client », vous trouverez un Makefile pour compiler vers un executable SpaceRun, à exécuter avec la commande *java SpaceRun.*
Notre serveur et notre client sont faits pour se connecter en local, sur le port 2019, mais vous pouvez facilement modifier ce comportement dans le fichier « server.ml » et dans le fichier « SpaceRun.java » et « Constants.java ».
ATTENTION : Veuillez à bien démarrer le serveur avant de lancer un client.

Guide du jeu :
La première fenêtre que se présente à vous est une petite fenêtre de connexion au jeu, entrez votre pseudo en minuscule, et appuyez sur le bouton « CONNECT ». En cas d'échec, vous trouverez le motif du refus de connexion en rouge, à côté du bouton. Une fois la connexion effectuée, en fonction de la phase de la session en cours, vous vous trouverez soit en jeu directement, soit en attente du début de la session. Dans les deux modes de jeu, il est possible pour les joueurs de rejoindre ou de quitter le jeu à tout moment.
Les commandes pour naviguer :

- **q** pour faire une rotation dans le sens antihoraire
- **d** permet de faire une rotation dans le sens horaire
- **z** permet d'appliquer une impulsion au vaisseau

Les commandes pour combattre :

- **p** pour poser une banane (piège)
- **l** pour tirer un boule de laser.

Attention, il faudra que le focus ne soit pas sur la zone de chat pour que les commandes soient prises en comptes.

5 Remarques

Le projet a été implémenté dans son ensemble, avec quelques extensions en plus. Cependant, on peut relever 2 petits bugs et quelques points d'insuffisance.
Un premier bug qui apparaît sous certaine condition, lorsque un vaisseau entre en collision avec un astéroïde selon un certain angle, un certain vecteur vitesse. Au lieu de "rebondir" le vaisseau reste collé à l'astéroïde, il est comme dirait-on pris au piège, car à chaque fois que le serveur calcule une nouvelle position du vaisseau, le vaisseau et l'astéroïde sont toujours en collision, ce qui inverse constamment le vecteur vitesse du vaisseau. Et pour se sortir de la il faut enchainer rapidement les commandes thrust. Nous avons donc essayé de régler ce problème en ajoutant une variable *is_ colliding* de type bool au joueur, qui permet de savoir si un vaisseau est déjà en collision, si oui, on ne fait rien tant que le vaisseau n'est pas sorti de la zone de collision, et si le vaisseau dépasse la zone de collision, et entre en zone de sécurité, la valeur de *is_ colliding* est remise à true. Mais cette implémentation ne marche pas complètement, certes cela a réduit la fréquence d'apparition de ces accrochages, mais il peut toujours se produire.
Le deuxième bug ne se présente pas souvent. Rarement, mais cela est déjà arrivé, un objectif peut s'initialiser "sous" un astéroïde, alors que nous avons utilisé coté serveur, une fonction donnant des coordonnées valides, soit, celles qui ne sont pas en collision avec aucun des astéroïdes. Une piste d'amélioration pourrait être envisagée au niveau de l'architecture du client. Nous voulions utiliser le design pattern Observer, dans lequel l'interface observerait les données, et à chaque modification des données, l'interface pourrait se rafraîchir toute seule. Cela n'a pas été mis en place tout de suite, et nous avons appris un peu tard qu'il est déprécié depuis Java 9. Nous avons donc gardé la structure telle quelle, mais ce n'est pas la plus efficace, ni la meilleure/bonne façon de faire peut-être.