Rapport Projet PC2R 2018-2019

Auteurs :

- *Nathan GERDAY* 3520055
- *Pierre GOMEZ* 3520013

Date de rendu : 14/04/2019

Remarque :
Il est conseillé de lire le rapport à l'addresse suivante pour avoir un format plus agréable et mieux mis en page (https://github.com/nathangerday/Report_2019/blob/master/Rapport.md)

Sommaire

Introduction

Choix des langages

- Le serveur est codé en **Java**. Nous n'utilisons pas de librairie particulière en dehors de celle fourni directement dans le langage.
- Le client est codé en **Python 3**. Toute l'interface graphique est gérée avec le module **Pygame**.

Manuel

Ce manuel a été testé sur Ubuntu ainsi que sur Debian

Installation

Serveur

Nous utilisons la version de Java 1.8. Le serveur devrait cependant fonctionner avec toutes les versions récentes de Java.

Nous avons utilisé un fichier ***build.xml*** afin de pouvoir compiler tout le serveur avec Ant. Il suffit pour cela d'utiliser la commande :

```
ant compile
```

Client

Nous utilisons Python 3 dans la version 3.6.7.
Les versions plus récentes fonctionnent également. Les versions plus anciennes de Python 3 devrait fonctionner pour la majorité d'entres elles bien que nous ne les ayons pas testées.

Il faut également installer la librairie Pygame. Le plus simple est de passer par l'installeur de modules pour Python qui s'appelle Pip.

Installation de pip (en root) :

```
apt-get install python3-pip
```

Installation de Pygame (pas en root) :

```
python3 -m pip install pygame
```

Utilisation

Démarrer le serveur

Une fois le serveur compilé avec `ant compile`, il suffit de le lancer avec la commande :

```
java -cp bin/ server.Serveur
```

Le serveur va ensuite va tourner sur le port indiqué et gérer les connexions jusqu'à ce qu'on l'arrête avec une interruption **Ctrl+C**.

Démarrer le client

Pour lancer un client depuis la racine du projet, on exécute la commande :

```
python3 client/client.py
```

Chaque exécution de la commande lancera un nouveau client.

Comment jouer ?

Le port par défaut est 45678. Pour le modifier sur le serveur, il faut aller dans le fichier `src/constants/Constants.java`. Pour le client, on peut modifier le port ainsi que l'addresse hôte dans le fichier `client/const.py`.

Le REFRESH_TICKRATE, qui définit notamment la vitese d'animation du client est à 30 par défaut. Nous avons pu remarquer, en testant le projet sur une machine virtuelle peu performante, que si l'ordinateur n'arrive pas à suivre ce tickrate, cela peut fausser la synchronisation entre le client et le serveur et donc afficher des états peu cohérents.

Avant tout, on démarre le serveur, comme indiqué dans la partie précédente.

Ensuite, lorsque l'on lance un client, un premier écran permet d'entrer une nom d'utilisateur. Une fois le nom entré, on peut appuyer sur **Entrée** pour se connecter automatiquement au serveur.

Si on est le premier client, on arrive alors dans une phase d'attente pendant laquelle on ne peut pas faire d'action autre que de parler dans le chat.

Une fois la partie commencée, on arrive sur un écran avec des obstacles (les ronds blancs), un objectif (le rond jaune), le vaisseau du joueur (en gris) et les vaisseaux des autres joueurs (en rouge) s'il y en a.

Le but, par défaut, est d'être le premier à récupérer 3 objectifs. Dans ce mode, les objectifs sont partagés par tout les joueurs. Une fois la partie gagnée par un joueur, on retourne en phase d'attente avec tout les joueurs de la session.

On peut rejoindre à tout moment une session en cours ou en attente en connectant un nouveau client.

Les actions possibles sont:

- Tourner dans le sens antihoraire => Touche **Q** ou **Flèche gauche**
- Tourner dans le sens horaire => Touche **D** ou **Flèche droite**
- Donner une impulsion => **Barre espace**
- Tirer => Touche **E**
- Quitter => **Echap**
- Commencer à écrire dans le chat et envoyer un message => **Entrée**
- Arreter d'écrire dans le chat => **Echap**

Pour envoyer un message privé, il faut écrire un message de la forme :
`/w <username> <message>`

Toutes les informations ainsi que messages du chat s'affichent dans un log en bas à gauche de la fenêtre. Les messages disparaissent après un certain temps.

Les scores de chaque joueurs sont affichés en haut à droite de l'écran triés en ordre décroissant.

On peut attaquer à tout moment et si un tir touche un adversaire, il s'arrête instantanément.

On peut également lancer une partie en mode Course. Pour cela, pendant la phase d'attente (au lancement de la session ou après qu'un joueur ait gagné la partie), il faut qu'un joueur écrive dans le chat : `/race`.

En mode Course, on voit alors 2 objectifs:

- L'objectif courant (en jaune clair), qui correpond à l'objectif qu'il faut récupérer actuellement.
- L'objectif suivant (en jaune foncé), qui correspond à la position du prochain objectif une fois que le courant aura été récupéré.

Description du projet

Travail réalisé / Extensions

Nous avons réalisé 3 extensions en plus des parties A, B et C du sujet.
Tout est fonctionnel et, bien que nous n'ayons pas testé nous-même, le serveur et le client respectent le protocole et devraient donc être compatibles avec d'autres. Lors de l'ajout d'extensions, nous avons veillé à conserver la compatibilié avec un client / serveur n'incluant pas ces extensions.

Le seul point sur lequel nous avons dû faire un choix et qui pourra donc créer un problème de compatibilité est au niveau de la gestion des coordonnées. Nous avons fait le choix d'avoir des données "abstraites" côté serveur et c'est ensuite au client de les convertir en coordonnées réelles en fonction de la taille de sa fenêtre. Côté serveur, nous avons des coordonnées entre -1 et 1, avec le point (0,0) au centre, le point (-1, -1) en haut à gauche et le point (1, 1) en bas à

droite. Tout le protocole utilise donc ces données abstraites. Cela nous permettrait en théorie d'avoir des tailles de fenêtre variable pour chaque client et de toujours garder des affichages cohérents.

Les extensions réalisées sont :

- Un chat tel que décrit dans le sujet avec affichage des messages directement dans la fenêtre de jeu.
- La possibilité de tirer sur les autres joueurs pour les immobiliser
- La possibilité de lancer une partie en mode "course" en écrivant `/race` pendant la phase d'attente. A la différence de la course décrite dans le sujet, on ne voit pas tout les objectifs mais seulement celui actuel ainsi que le suivant s'il y en a un.

Client

Nous allons tout d'abord décrire le fonctionnement général du client, sans rentrer dans les détails.

Le client initialise une fenêtre graphique avec un menu permettant de se connecter au serveur avec le protocole ***CONNECT***. Une fois la connexion réussie, on lance la fenêtre correspondant à une session en jeu. Cette fenêtre est composée d'une boucle infini réalisant les opérations suivantes : 1) Envoi des données au serveur en fonction de l'état actuel du jeu et des inputs. Lorsqu'on est en jeu, pour envoyer une fois tout les ***tickrate server***, on utilise une variable indiquant quand le dernier envoi a été fait, et on compare cette variable au ***tickrate server*** afin de savoir si on doit envoyer ou non. Cela fonctionne car on sait que le ***refresh tickrate*** du client est bien supérieur au ***server tickrate***. 2) Réception des données du serveur, parsing et application du comportement correspondant au message reçu. 3) Gestion des entrées claviers de l'utilisateur 4) Mise à jour des entités du client (Player, Attack, ...). Cela correspond à la prédiction, ces données seront potentiellement écrasées à la prochaine réception des données du serveur. 5) Affichage des entités sur la fenêtre. 6) Attente jusqu'a la fin du tick, afin d'avoir ***refresh tickrate*** ticks par seconde.

Le client est implémenté en utilisant principalement les classes de Python. Nous allons donc ici faire une présentation rapide de ces éléments et de leur rôle.

- Le fichier `const.py` contient toutes les constantes qui permettent de définir le comportement du jeu côté client. Il y a notamment les valeurs ***PORT*** et ***HOST*** qu'il peut être utile de changer pour connecter à un autre serveur, ainsi que les constantes du REFRESH_TICKRATE et SERVER_TICKRATE.
- Le fichier `send_serveur.py` contient les différentes fonctions pour envoyer des messages respectant le protocole au serveur.
- **Client** (dans `client.py`) représente la fenêtre d'interface du client avec quelques unes de ses proprités (titres, tailles, ...)
- **Menu** (dans `menu.py`) représente un écran dans lequelle on peut saisir un pseudo et se connecter à une session en appuyant sur Entrée
- **MultiplayerGame** (dans `multiplayer_game.py`) correspond à toute la gestion de la communication avec le serveur. On y trouve notamment les méthodes pour gérer tout

les comportement par rapport à chaque commande du protocole ainsi que toutes les variables représentant l'état courant du jeu. C'est ici que la boucle infini dont nous avons parlé plus haut à lieu, dans la méthode `main_loop()`.

- **Arena** (dans `arena.py`) contient toutes les entités d'une partie et s'occupe de faire le lien entre leurs coordonnées abstraites et les coordonées réelles de la fenêtre pour les dessiner, ainsi que de les mettre à jour.
- **Player** (dans `player.py`) représente toutes les informations sur l'état d'un joueur de la partie ainsi que les comportements des input utilisateurs.
- **Goal** (dans `goal.py`) représente un objectif de la partie et permet de verifier s'il est collectable à partir de coordonnées données.
- **Obstacle** (dans `obstacle.py`) représente un obstacle de la partir et permet de verifier si un Player est en collision avec.
- **Attack** (dans `attack.py`) représente un tir dans la partie avec sa position et son vecteur vitesse.
- **Score** (dans `score.py`) permet d'afficher des lignes de textes dans la fenêtre indiquant le score de chaque joueurs.
- **Logger** (dans `logger.py`) permet d'afficher des lignes de textes dans la fenêtre représentant les messages du chat ainsi que les informations intéressantes pour un joueur envoyées par le serveur.
- **InputBox** (dans `input_box.py`) représente un boite de dialogue permettant d'écrire du texte et de réaliser une action donnée lors de l'appui sur la touche Entrée.

Serveur

De même que pour le client, nous allons décrire le fonctionnement général du serveur avant de rentrer dans les détails.

Au lancement du serveur, un premier Thread se lance sur la classe Serveur. Ce thread va faire une boucle infini sur un mécanisme d'écoute de connexion de client. A chaque fois qu'un client se connecte au serveur, il va créer un nouveau Thread Connexion pour ce client qui sera lié à une Session. Ce Thread Connexion sera chargé de gérer toute la communication avec ce client précis ainsi que de modifier la Session, qui correspond à l'ensemble des ressources partagées entre tout les clients, en fonction des messages envoyés par le client. Le comportement de la boucle Connexion pour un client donné est donc:

1) Recevoir le message respectant le protocole du client. 2) Appliquer les modifications dans la Session partagée correspondant à ce message. 3) Si nécéssaire, envoyer un message réponse au client connecté et potentiellement à tout les autres clients de la session.

Pour faire cela, nous avons implémenté des méthodes dans Session, permettant d'appliquer chaque commande du protocole et qui se chargent de gérer les synchronisations. Ces méthodes s'occupent également de prévenir les Connexion liées de chaque client pour lesquels il faut envoyer une réponse.

Enfin, lorsque la phase est en mode jeu, il y a également un dernier Thread qui s'occupe de lancer régulièrement les ticks serveurs afin de mettre à jour sur le serveur et d'envoyer le protocole *TICK* à tous les clients connectés.

Le code du serveur est organisé en 3 parties distinctes : les constantes (package `constants`), les éléments de représentation de l'état du jeu (package `game_elements`) et les élements de gestion de la concurrence et du protocole (package `server`).

Nous allons ici expliquer un peu plus en détails le rôle et fonctionnement de chaque classe.

Package `constants` :

- **Constants** : De même que pour le client, contient toutes les constantes utile dans l'application avec notamment le *PORT* sur lequel écouter. Les constantes du REFRESH_TICKRATE et SERVER_TICKRATE s'y trouvent également.

Package `server` :

- **Serveur** : Attend les connexions des clients et crée envoi un nouveau thread Connexion dans un pool de threads lorsqu'un client se connecte. Il lie alors cette Connexion à un Session.
- **Connexion** : Gère toute les communications avec un client et modifie la Session à laquelle il est lié en fonction des communications.
- **Session** : Représente l'état de la partie actuelle. Elle est partagée entre toutes les Connexion et gère tout les accès concurrents de manière "safe".
- **ProtocolManager** : Classe contenant des méthodes statiques permettant de créer des String respectant le protocole en fonction des données brutes de la Session

Package `game_elements` :

- **Player** : Représente toute les données d'un joueur de la partie ainsi que différentes méthodes pour intéragir avec (notamment les collisions).
- **Objectif** : Représente l'état d'un objectif et de vérfier s'il est collectable par un joueur donnée.
- **Obstacle** : Représente un obstacle et permet de vérifier s'il est en collision avec un autre élément.
- **Attack** : Représente un tir et avec son état courant et permet d'intéragir avec un joueur si jamais il entre en collision.

Points pertinants

Synchronisation

Sur le serveur, toute la partie donnée partagée est représentée par la classe Session. C'est sur cette classe que chaque client va pouvoir avoir un impact sur l'état du jeu. Il faut donc bien évidemment géré les accès concurrents à tout moment dans le programme.

Pour cela, nous avons décidé d'utilise mot clé `synchronized` de Java qui répond parfaitement à nos attentes.

Cependant, nous avons fait le choix de ne pas rendre l'ensemble des méthodes de la classe

Session `synchronized` qui verrouillerait sur l'ensemble de l'instance de Session et serait donc un peu trop "fort" pour nous, il y aurait une perte d'efficacité.

Nous avons donc fait le choix de créer des Objects java pour chaque éléments qui risquent d'être modifiés avec des accès concurrents. Nous ne pouvions pas synchroniser directement sur les variables étant données que si nous changions l'instance, par exemple en changeant l'objectif, nous perdrions le verrou et des accès non controlés pourraient alors avoir lieux.

```
private final Object userLock = new Object(); // Verrou sur la liste des instanc
es de Player et des Connexions actuellement dans la Session
private final Object phaseLock = new Object(); // Verrou sur la phase actuelle d
u jeu
private final Object objectifLock = new Object(); // Verrou sur l'objectif coura
nt (ou la liste d'objectifs en mode course)
private final Object attacksLock = new Object(); // Verrou sur la liste des atta
ques encore présentes
```

Nous n'avons pas créé de verrou pour la liste d'obstacle car elle est créée une unique fois puis ne sera jamais modifiée jusqu'à la fin de la partie.

Afin de garantir qu'il n'y ait pas d'interblocage, nous avons également défini que les synchronisations sur plusieurs verrous devaient toujours être effectuées en respectant l'ordre de déclaration des verrous.

Cette méthode nous permet donc de verrouiller uniquement ce qui est nécéssaire tout en garantissant qu'il ne peut pas y avoir d'interblocage entre les Connexions.

Gestion de la session courante

Un des problèmes que nous avons rencontré est le fait que lorsque le premier client se connecte, il faut que le serveur entre en phase d'attente pendant un certain temps. Cependant, on ne peut pas utiliser `Thread.sleep()` afin d'attendre la durée souhaite, puisque cela bloquerait le thread appelant, en l'occurence la Connexion avec le premier client. Or, on veut que cette connexion continue d'écouter le client et puisse encore lui envoyer des informations, par exemple si un autre joueur se connecte.

A la première connexion il nous font donc lancer un autre thread qui sera chargé uniquement d'appeler la méthode `start()` après un certain temps.

Nous utilisons pour cela une méthode de la classe Executors de Java.

```
private void scheduleStart() {
    ScheduledExecutorService sch = Executors.newSingleThreadScheduledExecutor();
    Runnable task = new Runnable() {
        public void run() {
            start(); // Demarre la phase de "jeu"
        }
    };
    sch.schedule(task, delayBeforeStart, TimeUnit.SECONDS);
}
```

Un problème similaire s'est posé pour avoir des ticks servers à un fréquence régulière de manière indépendante des communications avec les clients. Nous créons donc un thread chargé d'appeler tick tant que la phase est "jeu".

```
private void autoTick() {
    Runnable task = new Runnable() {
        public void run() {
            while (true) {
                synchronized (phaseLock) {
                    if (!phase.equals("jeu") && !phase.equals("ingame_race")) {
                        // On ne veut pas appeler tick si on est pas en cours de je
u.
                        return;
                    }
                }
                tick();
                try {
                    Thread.sleep(1000 / Constants.SERVER_TICKRATE);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    };
    new Thread(task).start();
}
```

Tickrates

Nous allons ici nous intéresser aux problèmes rencontrés par le fait d'avoir 2 tickrates différents et indépendants pour le client et le serveur.

Le premier problème est au niveau de la mise à jour de la position selon le vecteur. Etant donné que le vecteur de chaque joueur est le même pour le serveur et pour le client, il nous font donc prendre en compte cette différence d'échelle d'un côte ou de l'autre. Nous avons décidé de le gérer sur le serveur, ce qui implique donc que le serveur doit connaître le *REFRESH_TICKRATE* du client.

L'autre principal problème est au niveau des collisions. En effet, lors de la détéction d'une collision nous renvoyons le vaisseau à sa position précédente. Cela est fait côté serveur, mais également côté client afin d'avoir une prédiction précise. Cependant, le *REFRESH_TICKRATE* du client était plus élevé que le *SERVER_TICKRATE*, la position précédente du client est plus proche de l'obstacle que celle du serveur et un décalage entre les 2 se crée alors. Lors du prochain tick, le serveur corrige donc la prédiction du client et cela a pour effet de téléporter légerement le vaisseau du joueur sur l'interface. Ce problème étant cependant quasiment invisible pour un *SERVER_TICKRATE* suffisamment haut, nous avons décider de ne pas complexifier le code pour le gérer.

Nous avons également réfléchi à l'extension proposée de faire les calculs côté client et de les vérifier côté serveur, ce qui aurait permis, entre autre, de régler le problème précédent.

Cependant cela créait un autre problème : si le client envoi ses coordonnées à un instant t0 et que le serveur reçoit les coordonnées, les valide, et les renvoit à l'instant t1, le vaiseau sur le client à alors bougé depuis l'instant t0 et donc cela le téléporte en arrière. L'extension censé rendre le jeu plus fluide a alors l'effet totalement opposé. Etant donné les tickrates différents et indépendants, il est quasiment impossible d'essayer de prédire pour le serveur où se trouvera le client à l'instant t1. La solution est donc que si jamais le serveur valide les coordonnées envoyées par le client, il ne doit pas réenvoyer la coordonnée à l'instant t1. Ainsi le client se déplace sans aucune intervention du serveur tant que les coordonnées sont validées. Mais cela nécéssite de totalement rompre la compatibilité avec les clients n'implémentant pas cette extension, ainsi que de complexifié le code d'envoi de tick aux clients puisqu'une position d'un client validée ne doit pas lui etre réenvoyé, mais elle doit tout de même être envoyée à tout les autres clients. Nous avons donc décider de ne pas implémenter cette extension.

Compatibilité Protocoles avec extensions

Tout au long de l'implémentation des extensions, nous avons fait en sorte de conserver à tout moment la compatibilité avec les projets n'implémentant pas ces extensions.

Cela a nécéssité de :

- Garder une rétro-compatibilité avec le protocole du sujet.
- Ne pas envoyer de messages correspondant aux protocoles ajoutés par les extensions sans action de la part utilisateur (Par exemple, on enverra donc jamais d'informations liées aux tirs tant que le joueur n'a pas tirée une fois)

Les commandes ajoutées aux protocoles sont :

- Le protocole du chat dans le sujet
- (C -> S) `NEWCOM2/comms+S1/` => Envoi les informations de NEWCOM classique et ajoute le fait qu'on a tirer une fois. (Nous avons fait en sorte de ne pouvoir tirer qu'une fois par tick).
- (C -> S) `RACE/` => Envoi au serveur une requête pour que le prochaine partie soit une course et non une partie standard.
- (S -> C) `TICK2/vcoords/vcoords/` => Envoi les informations sur l'ensemble des joueurs et également l'ensemble des tirs présents dans la partie avec leur position, vecteur et direction.
- (S -> C) `NEWOBJ/ocoords/scores/` => Si on est en partie standard, le ocoords contient la coordonée de l'unique objectif et est donc identique au format "coord" par défaut. En mode course, il contient l'objectif courant ainsi que l'objectif suivant.

Pool de threads

Le serveur, au moment d'accepter les connexions clients et de créer le thread Connexion pour le client en question passe par une pool de thread de la classe Executors de Java. Cela a peu d'influence sur l'ensemble du projet mais apporte 2 avantages non négligeables :

- Cela permet de limiter simplement le nombre de client à tout moment et donc de garantir une certaine sureté sur le serveur qui ne pourra pas être surcharger.
- La gestion de la pool de thread faite par Executors permet de pas recréer des threads à chaques nouvelles connexions, ce qui limite le coût en performance des connexions.

```
ExecutorService threadPool = Executors.newFixedThreadPool(20);
try {
    while (true){
        Socket client = ecoute.accept();
        Connexion c = new Connexion (client, currentSession);
        threadPool.submit(c);
    }

}catch (IOException e){System.err.println(e.getMessage());System.exit(1);}
```