Rapport de Projet de PC2R

ASTEROIDS

CHIV Willy
DONNOT Cyann

2018-2019

https://github.com/wichiv82/Projet_PC2R

Manuel d’utilisation

Installation :

Nous avons choisi pour ce projet, d’utiliser le langage Python pour implémenter le serveur et Java pour le client.
Le client utilise le module Javafx pour pouvoir créer l’interface graphique, à part cela nous n’utilisons pas d’autres modules spéciaux.

Utilisation :

Pour pouvoir démarrer le serveur, il suffit juste de paramétrer dans le main du fichier Server.ipynb l’adresse IP avec le port.
Pour démarrer un client, il faut exécuter le fichier ClientGraphique.java avec comme 1er paramètre l’adresse IP du serveur et ensuite en second paramètre le port de celui-ci.

Le lobby du jeu est lancé et attend les clients pendant quelques secondes. Lorsque le serveur lance la partie, chaque client lance chacun une interface graphique du jeu.

Les commandes pour le jeu sont :
- z pour appliquer un boost de vitesse (thrust)
- q pour tourner dans le sens antihoraire (anticlock)
- d pour tourner dans le sens horaire (clock)
- esc ou ECHAP pour quitter le jeu

Description du projet

La structure du projet est composée de 2 grandes parties : le serveur et le client.

Client

Chez le coté client , nous avons besoin de différentes classes pour pouvoir représenter les entités du jeu. Nous avons donc la classe Voiture (qui devrait être vaisseau mais c'est un détail :D), une interface Objet qui représente toutes les entités non joueurs, la classe Piece et la classe Obstacle qui implémentent celle-ci.
La classe ClientGraphique gère le jeu côté utilisateur et la classe Client gère la connexion entre le jeu et le serveur.

Nous sommes partis sur le fait que tous les objets pouvaient être ronds et donc possèdent tous une position sur la map avec un radius qui représente leur rayon. Piece possède une méthode isObjectif() renvoyant True permettant de le différencier des autres objets du jeu (ici nous n'avons que les obstacles).

Les Voiture possèdent un nom (celui du joueur correspondant), un Point vitesse et un entier direction pour pouvoir se déplacer, une valeur turnit pour obtenir l'angle de rotation en radians, une vitesse maximale et un score.
On dispose aussi de méthodes thrust() pour effectuer un boost de vitesse dans la direction du véhicule(), rotation(), clock() et anticlock pour changer la direction et reverseVitesse() pour gérer la vitesse dans les cas de collision.

La classe ClientGraphique lance le Client et attend que celui-ci confirme que le jeu est lancé pour lancer l'interface graphique. Elle contient tous les éléments graphiques à dessiner comme les voitures, l'avant des véhicules, les objets et leur version « physique dans le jeu». On dispose dans la 1ere moitié du code de la classe, des méthodes permettant de gérer les collisions et les positions des objets dans l'arène torique (Une voiture dans le jeu n'aura pas les mêmes coordonnées que sur l'interface graphique car le point d'origine a été centré au milieu contrairement au coin supérieur gauche de l'écran habituellement)
nextPosition() permet d'obtenir la position d'un véhicule au prochain tick et move() « avance » le jeu d'un cran et déplace les entités (c'est le tick). Il repère les collisions et réactualise la position de tous les objets graphiques.
Nous avons ensuite un gestionnaire d'inputs du clavier keyPressed qui lit les commandes effectuées par le joueur, effectue l'action correspondante et les ajoute à

une liste d’inputs qui sera transmise au serveur (sendInputs()) par le biais de la classe Client.
Un AnimationTimer timer qui rafraîchit l’interface graphique à une fréquence dépendant du refresh_tickrate.
La méthode init() qui initialise les objets/véhicules lorsqu’on reçoit la commande « SESSION » du seveur et start() qui démarre l’interface graphique.
Le main se contente de créer un objet Client et attend le signal de celui-ci pour pouvoir appeler init() et start().

La classe Client gère le lobby et indirectement toutes les communications avec le serveur. Elle commence par gérer le lobby en demandant le nom de l’utilisateur puis attend le lancement de la partie. Lorsque le jeu est lancé, elle donne un signal au ClientGraphique et lance 2 classes sur des threads : CommunicationEnvoiServeur et CommunicationReceptionServeur.
CommunicationEnvoiServeur envoie la liste des inputs effectués par l’utilisateur au serveur à une fréquence dépendant de serveur_tickrate.
CommunicationReceptionServeur lit les messages du serveur et les interprète pour mettre à jour une copie locale des voitures et des objets. On utilise ici une copie locale car lorsque le ClientGraphique aura besoin de copier les nouvelles informations des entités, on limitera les blocages.

Serveur

Coté Client, nous utilisons la librairie python socket qui permet la gestion de la communication client serveur ainsi que la librairie theading qui permet l’utilisation de thread.
Python possède quelques avantages non négligeables pour la gestion d’un serveur. En effet, socket permet de donner via settimeout() une durée de vie maximale aux différentes commandes d’acceptation de client ou d’envoi. Ce qui nous à permis de faire un lobby sans passer par un chrono externe (qui serait implémenté via un thread).
Deplus, les lock permettent une gestion de la concurrence assez facilement. Nous avons donc seulement un lock par client qui permet de copier et de rentrer sa commande sans avoir de problèmes d’accès simultanés.
Pour ce qui est de la gestion des client, nous utilisons un thread de réception par client puisque cette action est bloquante.
La boucle de jeu est indépendant du serveur, elle se déroule dans un thread qui tourne en arrière plan.
Le serveur comprends le lobby puis lance le jeu, il attends la fin de ce dernier avant d’expulser tout ses clients. Il fini en fermant le port.
L’envoie aux clients se fait directement via une classe (héritant du dictionnaire python). Cette dernière nous sert de carnet d’adresse. Elle peut être appelé par n’importe quel thread.

Extensions

Nous avons actuellement implémenté 2 extensions : l'interface graphique et le calcul côté serveur, ce qui a nous permit de simuler les lags et les « rollbacks » qu'on retrouve dans les jeux en ligne par exemple.
On remarque que plus l'écart entre le serveur_tickrate et le refresh_tickrate/ serveur_refresh_tickrate est important, plus les rollbacks se font ressentir.
Avec un refresh_tickrate/ serveur_refresh_tickrate de 50, et pour un serveur_tickrate de 50 aussi, le jeu est à la fluidité maximale. Pour un serveur_tickrate de 40, on ressent un lag visuel mais négligeable pour la jouabilité mais en dessous de 30, le jeu devient très désagréable à jouer et les vaisseaux font constamment des rollbacks.

Illustrations

```
private CommunicationEnvoiServeur envoi;
private CommunicationReceptionServeur reception;
private int serveur_tickrate;
```

```
DataOutputStream writer = new DataOutputStream(s.getOutputStream());
BufferedReader reader = new BufferedReader(new InputStreamReader(s.getInputStream()));
```

```
synchronized(nom) {
    envoi = new CommunicationEnvoiServeur(writer, nom, serveur_tickrate);
    synchronized(demarrer) {
        reception = new CommunicationReceptionServeur(reader, demarrer);
    }
}

Thread t_envoi = new Thread(envoi);
Thread t_reception = new Thread(reception);
t_reception.start();

try {
    synchronized(demarrer) {
        demarrer.wait();
    }
} catch (InterruptedException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
t_envoi.start();

try {
    t_envoi.join();
    t_reception.interrupt();
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

Images de la classe Client

La gestion des 2 classes de reception et d’envoi est effectuée sur 2 threads et communiquent avec des signaux pour pouvoir détecter le début du jeu, la fin du jeu par la terminaison du programme de l’utilisateur (cas ECHAP chez CommunicationEnvoiServeur) ou la fin de la partie par le serveur(cas WINNER chez CommunicationReceptionServeur).
Ici, nous ne sommes pas sur le modèle du cycle envoi puis réception qu’on manipule habituellemnt. On peut toujours effectuer des envois sans pour autant attendre la réponse d’un serveur pour nous débloquer (et inversement)., ce qui rend les interactions beaucoup plus « dynamiques ».