Ecole Nationale Supérieure de Techniques Avancées

Rapport de projet - IN104

Louis GOSSELIN, Florian MOREL

Mai 2022

1 Structure du code

Pour la réalisation de ce projet nous avons divisé notre code en 6 fichiers : main.cpp, menu.cpp, game.cpp, pod.cpp, checkpoint.cpp, utils.cpp.

Les 5 fichiers *.hpp sont utilisés pour définir les classes et fonctions et relier ces différents fichiers *.cpp au main.

Le main est construit en 2 gros blocs. Le premier correspond à la partie MENU et le second à la partie GAME. Pour gérer le passage d'une partie à l'autre, nous avons utilisé une variable *gamemode*. Cette variable prend la valeur 0 si on souhaite se rendre dans le menu et 1 si on souhaite se rendre dans le jeu. Elle est naturellement initialisée à 0 pour que le joueur se trouve dans le menu à l'ouverture du jeu.

2 Prise de décision d'un pod

Lors de chaque mise à jour de la physique du jeu, un pod va devoir prendre une décision afin d'accéder à son prochain checkpoint et donc de pouvoir espérer gagner la course. Cette décision est prise en fonction de l'état du pod (sa position, sa cible, sa vitesse...) et de celui de la partie. La fonction **getDecision** renvoie alors une décision qui correspond à la cible qu'il doit viser ainsi que la puissance qu'il doit fournir pour l'atteindre le plus rapidement possible.

Pod manuel

Dans le menu, il est donné la possibilité au joueur de choisir un pod manuel. Si le joueur en décide ainsi, l'instance *manualpod_* du pod BSGCylon est initialisée à 1. Lors du passage dans **getDecision**, la partie "manual pod" sera lue.

Cette partie consiste simplement à lire la position de la souris sur la grille et donner cette position en target au pod. On fixe la puissance du pod à la puissance maximale (100). Une piste d'amélioration du jeu pourrait consister à récupérer l'état de la barre d'espace pour augmenter la puissance si cette dernière est pressée par le joueur.

Pod automatique

Nous avons mis en place 2 types de pods automatiques différents. Le pod naïf et le pod intelligent. Ceci explique pourquoi seulement 2 pods sont présents dans notre jeu.

À chaque pas de temps physique, le pod naïf va choisir de cibler le centre de son prochain checkpoint et fixer sa puissance à son maximum (100).

Le pod intelligent va également choisir de cibler son prochain checkpoint mais va réguler sa puissance en fonction de l'angle qu'il fait avec la direction de sa cible avec la fonction **power**.

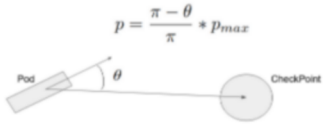

FIGURE 1 – schéma de la modulation de la puissance du pod intelligent

Après plusieurs tests nous nous sommes rendus compte que notre pod intelligent perdait la plupart des courses contre le pod naïf mais avait un rayon de mise en orbite autour d'un

checkpoint plus faible. Cette mise en orbite est dûe à l'angle maximal que peut faire le pod en tournant.

Vérification de l'angle de rotation du pod

Dans le sujet, il est précisé que le pod ne peut pas faire une rotation d'un angle de plus de $\frac{\pi}{10}$. Nous avons donc créé la fonction **verifAngle** dans utils.cpp qui renvoie une cible qui respecte ce critère pour le pod à chaque prise de décision.

Le principe est de multiplier le vecteur qui part du pod vers la cible par une matrice de rotation avec un coefficient α qui permet de fixer la nouvelle cible à un écart d'angle de $\frac{\pi}{10}$ avec l'angle du pod.

Après plusieur tests, nous avons remarqué qu'un angle de rotation maximal de $\frac{\pi}{10}$ obligeait les pods à faire des virages trop grands. Nous avons donc fixer l'angle maximal à $\frac{\pi}{3}$ afin de rendre le jeu jouable tout en évitant les demi-tours qui sont physiquement impossibles.

3 Mise à jour des graphismes du jeu

La mise à jour de la position réelle des pods ne se fait seulement qu'entre 2 temps physiques (100 ms) dans la fonction **updatePhysics**. Si l'affichage du pod n'est actualisée que lors de cette mise à jour, le visuel du jeu serait saccadé. Pour éviter ce phénomène, une fonction **updateGraphics** a été créée. Elle permet d'afficher les pods de façon plus récurente en suivant la vitesse du processeur. Elle réalise une interpolation linéaire classique.

```
192     float k = (physicsTime - frameTime)/(frameTime-lastFrameTime);
```

Ce coefficient k, défini dans la fonction **updateGraphics**, permet de compter le nombre de positions que le pod va prendre avant d'arriver à sa prochaine position réelle.

4 Détection du passage d'un pod dans un checkpoint

Afin de détecter si un pod est passé par un checkpoint, nous avons vérifié si le centre du pod passait à l'intérieur du cercle avec le test suivant : "norm(pod.pos_ - allCPs_[pod.nextCP_])<550".

Le problème est que le centre de la sprite peut passer par le checkpoint sans que le centre du pod n'y passe. Ceci aurait été un problème à régler par la suite.

5 Fonctionnalités supplémentaires

Afin d'égayer l'expérience utilisateur sur le jeu, nous avons ajouté quelques fonctionnalités :

- Un menu qui permet de sélectionner un nombre de checkpoints et le mode manuel. Grâce à la fonction **coordCP** du fichier utils.cpp, les checkpoints sont positionnés aléatoirement et de façon à ne pas se superposer. Le nombre de checkpoints à afficher peut être choisi de façon aléatoire.
- Un compte à rebours de 3 secondes qui implique le démarrage du jeu lorsque frameTime.asMilliseconds() > 3000 (ligne 135 du main).
- L'affichage du vainqueur à la fin de la partie avec une variable *sf : :Texte winner_* dans la classe Game.