Lic. Info. 3ème année — Université d'Orléans
Projet — Année 2020-2021

Barjo-Kart

Projet de Programmation en C++

BK14 : Jean-Daniel de Ambrogi, Faez Bhatti, Émilie Richard, Salma Badri

À rendre avant le 31 mars 2021, 12h

1 Simulateur

Conformément aux recommandations du sujet nous avons commencé par coder un simulateur permettant de tester des trajectoires. L'une des premières difficultés que nous avons rencontrée a été la prise en main des deux bibliothèques pour gérer les fichiers png et toml. Après avoir passé un certain temps dessus nous avons pu mettre en place nos différentes classes pour la vérifier. Nous sommes ainsi parti, d'abord, sur une classe image qui va enregistrer les caractéristiques du circuit puis une classe trajectoire qui va conserver, et bien, la trajectoire. Dans Image on a une matrice de `char` tel que n est un pixel noir, alors un obstacle, a un pixel de la zone d'arrivée et b un pixel blanc sur lequel on peut librement se déplacer (et `g` tous les pixels à la limite entre b et n qui sont d'une couleur intermédiaire). Pour la trajectoire on va la stocker dans une `map` afin d'avoir pour clé la position de l'étape dans la trajectoire. Pour nous faciliter les tests de trajectoires nous avons codé une fonction `void build_fake_trajectoire()` afin de créer une fausse trajectoire initiale puis deux fonctions, `write` et `load` qui nous ont permis de créer un toml à partir de la trajectoire puis de l'éditer facilement à la main avant de le recharger. la fonction write nous permet aussi de revoir à la main la trajectoire trouvé par la suite.

Faez : "*Pour vérifier que l'on ne dépasse pas les limites des circuits nous avons dû implémenter l'algorithme de Bresenham. Cet algorithme est souvent utilisé pour dessiner des segments de droites en utilisant uniquement les points d'une grille discrète. Donc on s'en est servi pour la construction de la trajectoire. J'ai choisi la version la plus complète et optimisée en pensant qu'elle serait bénéfique pour l'efficacité du calcul de la trajectoire. Ce qui m'a posé soucis était de bien se repérer, savoir dans quel cadran et dans quel octant on était(et de bien indenter le code pour éviter les erreurs).*"

Finalement nous avons réussi à coder le simulateur sans d'autres difficultés marquantes.

2 Premiere Approche : histoire de direction

2.1 Principe

Au début nous pensions que l'utilisation d'un algorithme de recherche des plus courts chemins allait être trop complexe à appliquer ici, notamment pour gérer le placement des sommets [1]. Nous sommes donc parti sur une première approche un peu *naïve*, celle d'aller dans la direction de la zone d'arrivée. Nous avons donc chercher à d'abord trouver un chemin en faisant des étapes successives dont la vitesse était égale à la moitié de l'accélération maximale. On aurait par la suite amélioré la trajectoire. Ainsi à chaque étape on cherche dans quelle direction se trouve le plus de pixel de la zone d'arrivée. Ensuite on regarde le potentiel de cette direction, c'est à dire combien de pixel nous sépare du prochain obstacle dans cette même direction [2] . Si le potentiel était trop faible on regarderait alors les directions voisines et si il était tout aussi mauvais on irait chercher la direction avec le plus grand potentiel. Une fois la direction trouvé on s'y dirige et ainsi de suite jusqu'à l'arrivée.

1. un par pixel ? aux angles de la figures ? uniformément sur toute la surface de l'image ?
2. nous souhaitions par la suite affiner ce potentiel par d'autres marqueurs tel que la distance à l'obstacle le plus proche de chaque pixel de la trajectoire dans cette direction

2.2 Réalisation et résultat

Jean-Daniel : "*L'une des difficultés rencontrées a été de bien calculer les différentes zones à partir de n'importe quel pixel. Nous avions choisi de calculer huit zones*[3] *et c'est tout autant d'équation de droite à trouver puis à comparer à chaque pixel de la zone d'arrivée tout en faisant attention à l'axe inversé des abscisses. Ce fut assez compliqué à gérer. Pour vérifier ces zones j'ai fait une fonction qui m'a permis d'afficher le circuit avec les zones sous forme de pgm :*"

FIGURE 1 – Une étapes sur circuit0

Par la suite il suffisait de choisir la trajectoire puis la vitesse possible. Cet approche nous a permis de réussir à trouver une bonne trajectoire pour `circuit0` que nous avons alors pu soumettre au serveur qui l'a accepté. Mais cela ne fonctionnait pas tout aussi bien pour pi.
En effet après le premier virage passé une fois arrivé a mi-chemin de la deuxième "jambe" il finissait par faire rebrousser chemin. Nous avons bien tenté de lui interdire de faire demi-tour mais arrivé en haut du circuit il faisait un tour sur lui-même, perdu, et finissait par revenir sur ses pas. Nous avons donc dû nous résigner à abandonner cette approche trop naïve.

3 Deuxieme approche : Dijkstra

Nous nous sommes alors efforcé d'implémenter un algorithme de recherche des plus courts chemins, ici Dijkstra.

3.1 Principe

Pour créer le graphe nous avons décidé de répartir uniformément des sommets sur la surface du circuit tout en respectant certaines contraintes. Ainsi nous avons centré notre grille sur le pixel de départ, alors sommet n°0. De plus chaque sommet est éloigné de son voisin par la moitié de l'accélération maximale afin de pouvoir convertir une suite de chemin en une trajectoire de manière assez sûre. De plus cela nous permet de garder une certaine précision dans nos mouvements à l'intérieur du circuit. Enfin lors de la création des sommets nous avons isolé ceux qui se situait sur des obstacles (alors numéroté -1). Lors de la création des arêtes nous avons fait attention à ne pas relier des sommets qui sont séparé par un obstacle. Ainsi nous obtenions un sous graphe qui était à l'intérieur du circuit, inaccessible à tout autre sommet de l'image[4].Par la suite il nous suffisait de convertir le chemin trouvé par l'algorithme de Dijkstra en une suite d'étapes.

3. A savoir : N,E,S,O et NE,NO,SE,SO
4. Par soucis de cohésion de numérotation des sommets et dû à l'implémentation de l'algorithme de Dijkstra nous n'avons pas supprimé ces sommets inaccessibles

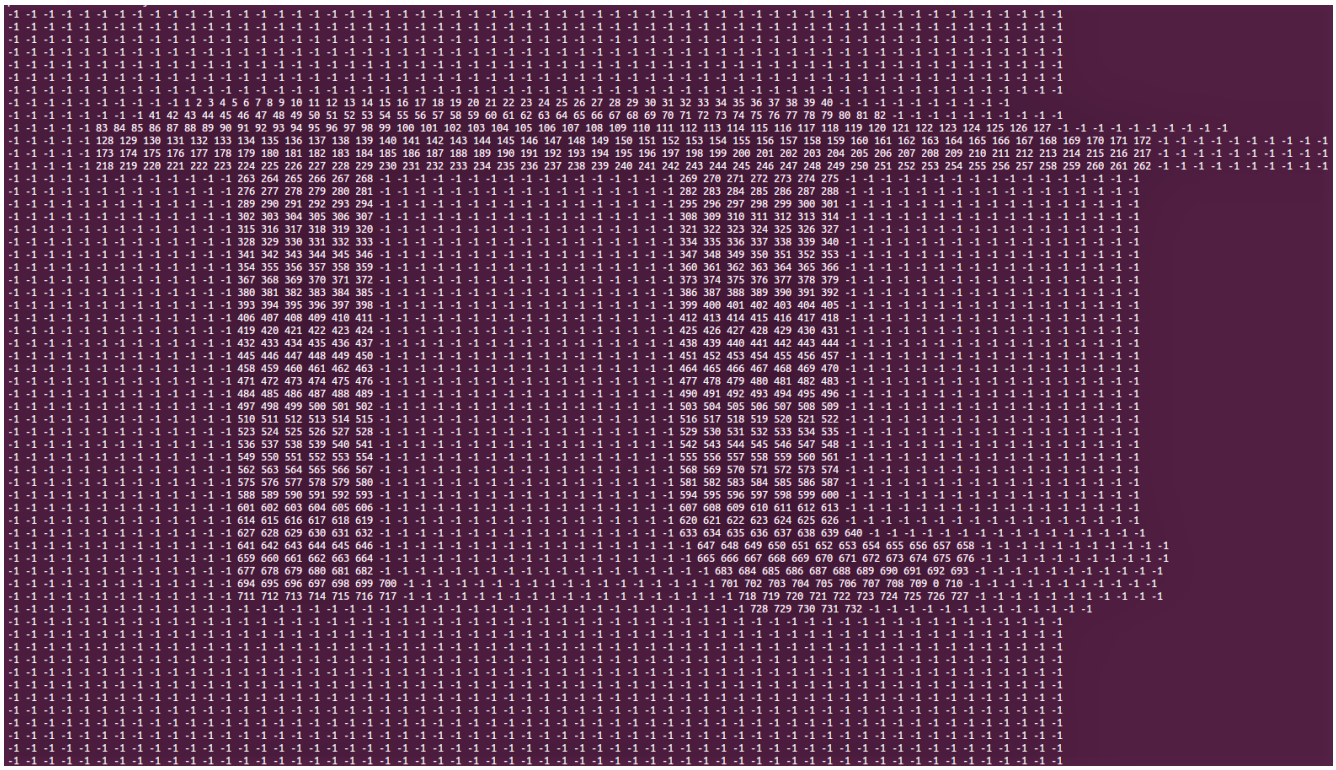

FIGURE 2 – Le graphe du circuit π

3.2 Réalisation et Résultats

La création du graphe fut assez complexe, nous avons dû faire attention à correctement gérer nos contraintes et celle secondaire entraînées par ces dernières. Par exemple le décalage de la grille de sommet pour l'aligner sur le pixel de départ poussait des sommets en dehors de l'image. La conversion a elle aussi relevé certaines difficultés mais rien de notable. Nous avons d'ailleurs dû créer une suite de fonctions afin de trouver le sommet le plus proche de la zone d'arrivée et si il ne se trouvait pas dans celle-ci de rajouter une étape afin de s'y diriger. Nous avons notamment pu réutiliser notre premier automate.
Grâce à l'algorithme des plus courts chemin nous avons été assuré de pouvoir aller jusqu'à la zone d'arrivée de π, mais non sans quelques petits virages inattendus :

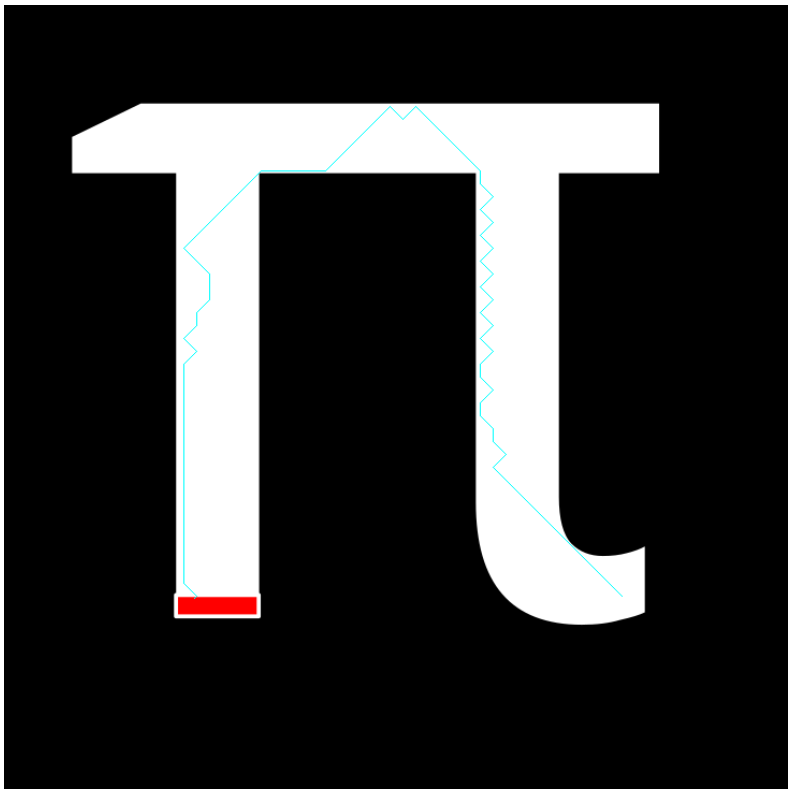

FIGURE 3 – Première trajectoire réussie sur π

4 Amélioration de la trajectoire

Afin de régler ce problème nous avons décidé de *lisser* la trajectoire et d'en profiter pour accélérer.

4.1 Principe

Pour réduire le nombre d'étapes nous sommes parti sur la recherche du plus long chemin en ligne droite. On va donc parcourir les étapes et voir jusqu'à quel étape suivante peut-on sauter sans rencontrer d'obstacle. Cette dernière devient alors une étapes. On les nommes les étapes générales. Ensuite on va décomposer chaque étape générale en sous étape, conformes à l'accélération maximale imposée. Dans chaque détail d'étape générale on calcule une phase d'accélération et une phase de décélération, cette dernière démarrant toujours au plus tard à la moitié du segment permettant de ralentir à temps pour le prochain virage.

4.2 Réalisation et Résultats

Cette partie a été très critique, on a relevé de nombreuses difficultés tant les calculs était difficiles a gérer ; par exemple pour détailler l'étape générale il faut trouver l'équation de la droite afin de trouver un pas X qui permet de respecter les contraintes d'accélérations mais qui n'est qu'un arrondi que l'on doit a chaque fois réajuster pour être sûr d'arriver sur le bon pixel. Par ailleurs il fallait réussir a gérer les trajectoire dans toutes les directions mais aussi dans les deux sens.

"Émilie : *Afin de ne pas raser les murs, lors de la sélection des étapes générales nous avions pensé à réaliser une matrice afin de voir la distance à l'obstacle le plus proche. Je voulais réaliser une matrice 3*3 afin d'y placer cette donnée mais je n'ai pas réussi à récupérer ces fameuses distances de manière sûr et efficace... Finalement nous avons contourner le problème en vérifiant que les pixels par lesquels nous passions n'était pas des voisins directes d'obstacles.* "

On a ainsi réussi à améliorer nos trajectoires, ainsi pour la fusée nous sommes passé de 117 étapes à 44 ! Cela nous a aussi permis de "lisser" la trajectoire de *pi* :

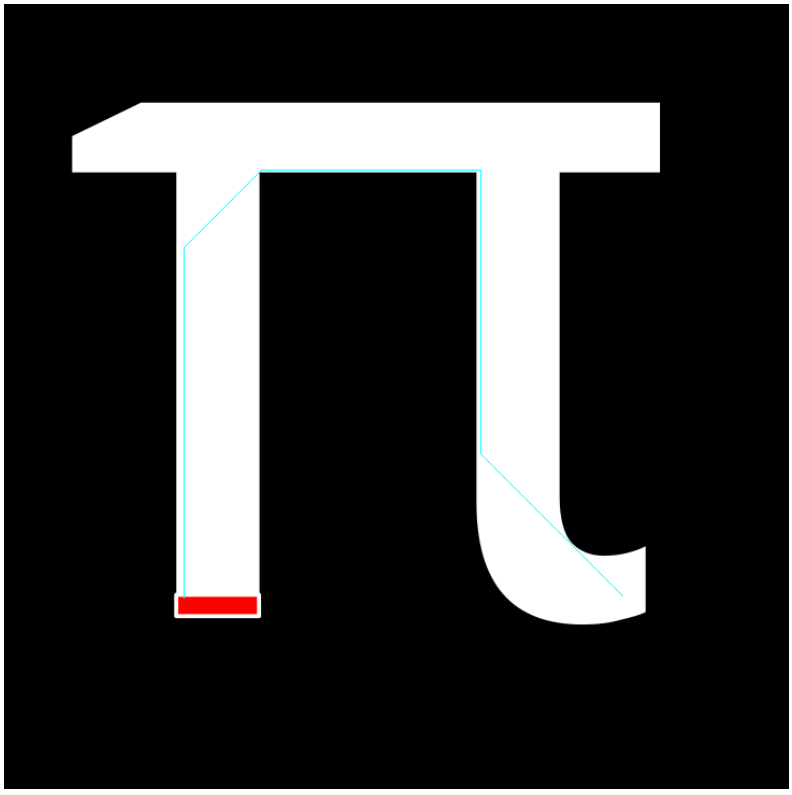

FIGURE 4 – La trajectoire améliorée sur π

5 Conclusion

Ce Projet nous a paru intéressant et nous a bien familiarisé avec C++. Parmi les points que l'on aurait amélioré après avoir eu une vue d'ensemble du projet on aurait peut-être pas dû faire appel à une matrice de type `char**`. Compliquée à gérer, on aurait dû plutôt partir sur une map de map. De plus nous n'avons pas pu aborder une amélioration des virages qui nous aurait bien aidé à réduire le nombre d'étapes. Néanmoins nous sommes satisfait de notre travail et on retient du projet que le C++ demande beaucoup de rigueur et une bonne entraide.