Développement de simulations spatiales dans le système solaire

Stage facultatif de fin de L2 MPCI

Benjamin LAGOSANTO

Encadré par
François BRUCKER

Année 2023/2024

Table des matières

Remerciements

Avant de commencer ce rapport, je tiens à remercier mon encadrant de stage François Brucker pour cette idée innovante et qui correspond parfaitement avec ce que je cherchais dans ce stage de fin de L2. J'ai eu la chance de m'amuser sur un projet intéressant, en recevant des conseils précieux et des avis critiques sur ma production, par quelqu'un d'aussi fasciné que moi par ce projet. Je remercie aussi tous les enthousiastes qui m'ont donné des idées et des améliorations à apporter. Je ferai de mon mieux pour que le jeu puisse voir le jour quand il sera prêt. Enfin, merci au rapporteur de consacrer du temps à ce projet.

Introduction

Dans ce stage de deux mois au Laboratoire d'Informatique et Systèmes, il m'a été proposé de développer des simulations physiques d'objets en mouvement dans le système solaire. Dans les faits, ces simulations constituent la base d'un jeu d'exploration et de commerce spatial de type 4X en multijoueur, et j'ai donc pris soin de programmer une interface graphique, une architecture client/serveur et une gestion des touches en complément du coeur du projet.

Pour que ce rapport reste concis, je me concentre ici sur la construction et l'optimisation du moteur physique, mais l'ensemble du projet peut-être consulté sur Github à l'adresse https://github.com/ben3dninja/solar4x et exécuté en local (conseillé pour apprécier le projet dans sa globalité).

Cahier des charges Pour préciser davantage les objectifs du stage, le but est dans un premier temps de pouvoir calculer à court et moyen terme les trajectoires de vaisseaux spatiaux plongés dans le système solaire, en prenant en compte l'influence gravitationnelle d'un grand nombre d'astres, dont les positions changent continuellement. La simulation doit pouvoir être faite en temps réel (on doit pouvoir "voir" les vaisseaux bouger autour des astres quand le jeu progresse) mais aussi en amont afin de pouvoir planifier des opérations.

Il doit également être possible d'altérer la trajectoire d'un vaisseau, via des poussées instantanées dans des directions précises.

Enfin, les manoeuvres doivent pouvoir êtres communiquées efficacement entre les machines, et les simulations doivent donc être réfléchies pour s'exécuter de façon déterministe afin que chaque joueur ait le même état du système devant lui à tout instant.

Outils de développement Concernant les technologies utilisées, l'entièreté du développement est pour l'instant réalisée en langage Rust [1], reconnu pour sa rapidité d'exécution (de part son caractère bas-niveau) mais aussi pour sa gestion des erreurs (le plus souvent signalées à la compilation) qui en font un outil particulièrement adapté pour le développement d'un jeu qui, dans notre cas, se veut rapide, simple et modulaire.

La gestion des sources se fait avec Git [2] et Github, deux outils auxquels les étudiants sont rarement introduits en licence, de part leur caractère appliqué, mais qui sont fondamentaux dans d'industrie du développement logiciel.

Ma familiarisation avec ces outils ayant eu lieu hors du cadre universitaire et bien avant le début du stage, il serait trop long de la détailler dans ce rapport.

Plan du rapport A partir du cahier des charges, on peut distinguer deux grandes parties : la mise en place des astres du système solaire et de leur mouvement périodique, qui comprend la collecte des données et la résolution des équations horaires ; et l'ajout des vaisseaux, dont en particulier l'intégration des équations du mouvement et l'édition des trajectoires.

Dans ce rapport, j'ai suivi cet ordre qui se trouve aussi être chronologique, en insistant sur les choix de design qui ont été faits à la fin de la première partie du stage pour optimiser l'application et assurer son bon développement à l'avenir.

Présentation du laboratoire

Le Laboratoire d'Informatique et Systèmes est une unité de recherche qui mène des travaux à la fois fondamentaux et appliqués dans quatre "pôles" : le calcul, la science des données, l'analyse et le contrôle des systèmes, et le signal et l'image.

J'ai été encadré par François Brucker, qui travaille dans l'équipe ACRO (Algorithmique, Combinatoire et Recherche Opérationnelle) du pôle calcul. Mon stage n'est pas en lien direct avec un sujet de recherche du laboratoire, cependant il apporte tout de même un intérêt pour de futurs travaux : vérifier que le langage Rust est un outil adapté au développement d'algorithmes et à un travail théorique.

En effet, ce stage a nécessité l'implémentation de plusieurs algorithmes (principalement des parcours d'arbres en parallèle, mais il y a aussi des exemples plus fins comme celui de la mise à jour des influenceurs dans le calcul des prédictions que l'on survolera dans la partie 2.3) mais on peut aussi prendre l'exemple de la communication réseau et du protocole QUIC [3], dont l'implémentation en Rust utilise des algorithmes de cryptographie pour apporter des échanges sécurisés.

Ainsi, ce projet s'inscrit comme une sorte d'expérience algorithmique dans les activités de recherche du LIS, et lors du développement j'ai eu l'occasion de rencontrer certaines structures algorithmiques comme les B-tree.

Partie 1

Mise en place des astres du système solaire

La première étape dans le calcul des trajectoires de vaisseaux dans le système solaire, c'est de mettre en place les astres et leurs mouvements. En effet, la masse d'un vaisseau étant de très loin négligeable devant celle d'une planète ou même d'une lune, on peut se permettre de négliger l'attraction gravitationnelle des vaisseaux pour le calcul du mouvement des astres. On se retrouve donc avec un calcul en deux étapes, qui marquent les deux parties de ce rapport.

La première étape a ainsi été de calculer les positions successives des astres, dans le référentiel du Soleil. En toute rigueur, il aurait fallu choisir le barycentre du système solaire comme référentiel afin de considérer l'attraction des planètes sur le Soleil, mais pour ces prototypes de simulations nous ne recherchions pas ce niveau de précision.

Quelques définitions

- Lorsqu'un ou plusieurs astres en orbitent un autre, on appellera celui-ci le **parent** et ceux qui l'orbitent les **enfants** (par exemple les planètes sont les enfants du Soleil, et la Lune est l'unique enfant de la Terre). On distinguera les **coordonnées absolues** des enfants, qui seront calculées par rapport au repère cartésien centré sur le Soleil, des **coordonnées relatives** dont l'origine est la position du parent.
- Le **plan de l'écliptique** est le plan géométrique moyen des trajectoires des planètes autour du Soleil.

1.1 Conventions pour la description d'orbites

Pour commencer, il a été important de comprendre la façon dont les orbites sont décrites dans la littérature.

Un cours de première année de mécanique montre que dans le cas d'un système à deux corps (c'est-à-dire deux points matériels seuls et soumis uniquement à la force de gravitation), chacun des corps se déplace en traçant une ellipse avec le barycentre comme foyer. Ici, on fera aussi l'approximation que les enfants d'un astre n'exercent pas d'attraction gravitationnelle sur leur parent, c'est-à-dire que le parent est confondu avec le barycentre au lieu de l'orbiter. C'est une grosse approximation, mais les calculs auraient été nettement plus complexes si on avait dû considérer le barycentre entre une planète et tous ses satellites. Cependant pour la plupart

des cas, ce n'est pas très gênant (par exemple le barycentre entre la Terre et la Lune se situe à l'intérieur de la Terre, à 1700km sous la surface).

Enfin, puisque les différents enfants d'un astre sont considérés sont suffisamment éloignés par rapport à leurs masses respectives, on suppose alors qu'ils n'interfèrent pas, ou peu de sorte à ce que leurs trajectoires restent elliptiques autour de leur astre parent.

On sépare un calcul complexe en une série de petits calculs indépendants.

Ainsi, on trouve dans la littérature que les orbites sont décrites via 6 coordonnées qu'on appelle **éléments orbitaux** , dont 5 qui permettent de construire l'ellipse et de la placer dans l'espace, et une qui donne la position de l'astre sur l'ellipse à un instant précis. On note que ces coordonnées sont au nombre de 6 parce qu'en mécanique il faut et il suffit de 6 coordonnées, 3 pour la quantité de mouvement et 3 pour la position, pour prédire une trajectoire dans un environnement donné.

Il existe différents jeux d'éléments orbitaux, mais la plupart comportent (figure 1.1) :

— le demi-grand axe (a) et l'excentricité (e) de l'ellipse, qui donnent la "forme" de la trajectoire dans le plan du mouvement

— 3 angles qui décrivent l'orientation de l'ellipse dans l'espace, par rapport à un repère de référence. Dans notre cas, le point vernal (une direction [presque] fixe dans l'espace) et le plan de l'écliptique fixant le repère.

— un angle indiquant la position de l'astre sur l'ellipse à un instant donné (souvent le 1/01/2000 à midi)

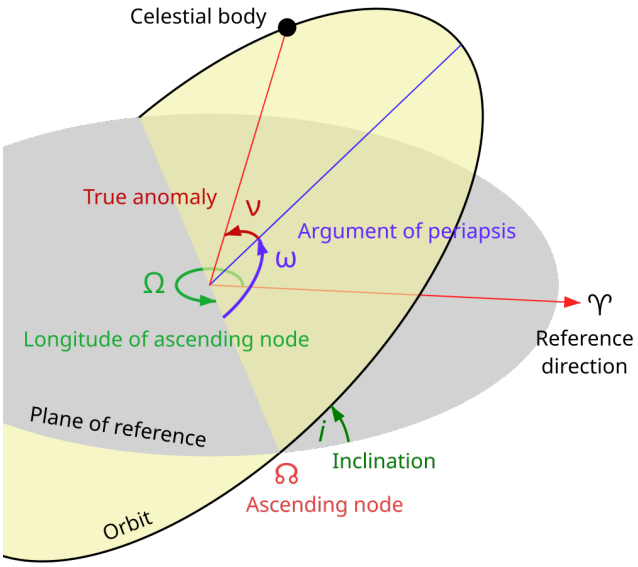

FIGURE 1.1 – Éléments orbitaux courants (figure de Lasunncty sur Wikipédia, CC BY-SA 3.0)

A partir de ces éléments, on peut calculer la position et la vitesse d'un astre en coordonnées cartésiennes à tout instant en seulement quelques opérations, qui sont décrites explicitement sur le site web de la NASA [4].

1.2 Collecte des données

Pour pouvoir commencer à simuler, nous avons ainsi eu besoin des éléments orbitaux d'un nombre suffisant d'objets du système solaire. Dans notre cas, il a été établi que les planètes, leurs satellites ainsi que quelques astéroïdes/planètes naines/comètes suffisaient largement dans un premier temps. On a donc pu se tourner vers une API REST (c'est-à-dire des données accessibles par HTTP) écrite par un français [5] et qui nous a apporté le nécessaire au format json [6], avec quelques données supplémentaires comme la masse, qui a été utile pour la force de gravitation par la suite.

A partir de ces données, j'ai pu écrire un programme qui désérialise ces données avec la bibliothèque (ou "crate" dans l'écosystème Rust) `serde` (https://github.com/serde-rs/json dans mon cas) et qui calcule la position des astres par rapport à leur parent à un instant donné.

Le calcul se fait en deux étapes : on établit le mouvement dans le plan de la trajectoire en résolvant numériquement l'équation de Kepler (à l'aide de la méthode de Newton en optimisation, décrite dans ce cas particulier [4]), puis on applique une matrice de rotation (qui est en fait le produit de trois rotations correspondant aux angles d'Euler [7]) pour obtenir la position relative au parent en cartésien.

1.3 Coordonnées absolues et affichage d'une carte spatiale

Naturellement, la prochaine étape est de calculer les coordonnées absolues des astres à partir de leurs coordonnées relatives (cf 1).

Pour cela, il est utile de modéliser les astres comme un arbre, selon les définitions de parent et d'enfants qu'on a posées. Sous cet angle, on remarque qu'un parcours préfixe (ou en largeur par exemple), c'est-à-dire où l'on traite le parent avant les enfants, convient parfaitement pour calculer les coordonnées absolues des astres. Par exemple, pour calculer la position absolue de la Lune, on ajoute sa position relative à la position absolue de la Terre, qui est elle-même égale à la position relative de la Terre autour du Soleil, l'origine du repère.

Ici, j'ai fait le choix d'implémenter un parcours en largeur, à l'aide d'une queue à laquelle on ajoute à chaque itération des couples contenant les enfants de l'élément suivant ainsi que leurs coordonnées absolues. Ce parcours est optimal puisqu'on calcule exactement une fois la position absolue de chaque astre.

On a à nouveau décomposé le travail en deux temps : on calcule (éventuellement en parallèle) les positions relatives de tous les astres, puis on met à jour les positions absolues (on peut aussi calculer les positions absolues des enfants d'un même astre en parallèle, ce qui rend le calcul très rapide). Il ne reste plus qu'à construire une interface graphique qui affiche les astres à la position qui convient à des intervalles de temps successifs et on obtient une carte spatiale. Dans mon cas je l'ai fait dans le terminal à l'aide de la crate `ratatui` [8], et le résultat est visible dans le mode "Exploration" de l'application (figure 2.2)

1.4 Parallélisation et modèle ECS

Jusqu'à maintenant, toute l'exécution du programme est séquentielle et fixe : à chaque tour de boucle, on met à jour les positions relatives, puis les positions absolues et enfin l'affichage.

Cependant, d'après [9], dans l'industrie du jeu vidéo, on a plutôt tendance à utiliser le design pattern de la "game loop". Le principe est simple, le jeu consiste en deux boucles imbriquées :

- La boucle principale, qui exécute la boucle interne puis met à jour l'affichage
- La boucle interne, qui s'exécute le nombre de fois nécessaire pour simuler un pas de temps fixé dt (par exemple 10 fois si $dt = 0.1s$ et s'il s'est déroulé un temps $\Delta t = 1s$ depuis la dernière itération de la boucle principale)

Cela permet de complètement découpler le framerate de l'affichage du pas de temps de la simulation, et donne un affichage qui paraît plus fluide sans avoir à diminuer le pas de temps. D'un autre côté, cela garantit que sur deux machines différentes, il s'est exécuté le même nombre d'étapes de simulation après une même durée du programme. C'est très utile pour la partie 1.5.

Attention cependant, on ne parle pas encore de parallélisation puisque le programme s'exécute quand même une instruction après l'autre. Quand j'ai voulu mettre en parallèle mon programme, plusieurs options se sont présentées à moi.

D'abord, j'ai écrit le code "à la main", en utilisant les primitifs de Rust, en particulier les threads et les channels. Cette voie est la plus fastidieuse, puisqu'on doit s'assurer que les ressources sont utilisées efficacement, et qu'elles ne sont par exemple pas modifiées simultanément. D'où l'utilité de certains objets comme les mutex ou les reference counters (tout est détaillé dans le Rust Book accessible depuis [1]), et certains patterns comme le double buffering [9]. Elle donne cependant le plus de contrôle puisque chaque mise en parallèle est minutieusement réalisée, et j'ai réussi à aller assez loin (voir commit 6558954) sur cette voie.

Cependant, la quantité de code augmentait beaucoup pour un petit nombre d'ajouts seulement, et il devenait difficile de continuer à répondre aux objectifs du stage en temps et en heure, donc j'ai fait le choix d'utiliser Bevy [10], un moteur de jeu qui a déjà bien fait ses preuves et qui est disponible sous forme de crate. Ce qui le rend si puissant, c'est son utilisation du modèle Entity Component Systems, qui devient très populaire dans l'industrie moderne du jeu vidéo (la vidéo de Timothy Ford, un des développeurs d'Overwatch m'a bien servi [11]). L'idée est simple :

- L'ensemble des objets qui vivent dans le jeu sont appelées des Entités et se voient attribuées un identifiant unique (par exemple dans notre cas, chaque astre ou vaisseau est une entité).
- A chaque entité on peut ajouter différents Composants, c'est-à-dire un trait commun à certaines entités (par exemple, j'ai choisi d'ajouter un composant "Orbite" aux astres, qui stocke les éléments orbitaux et qui possède des méthodes pour calculer les coordonnées cartésiennes, mais ils ont aussi un composant "Position" et même un modèle 3d).
- Enfin, une game loop exécute différents Systèmes en parallèle, c'est-à-dire des fonctions qui filtrent les entités selon certains composants et qui effectuent des tâches groupées (par exemple, j'ai écrit un système qui parcourt tous les composants "Orbite" et qui les mets à jour en fonction d'une ressource globale "Temps", puis un autre système qui parcourt les "Positions" et les "Orbites" et qui réalise le parcours d'arbre décrit dans la partie 1.3).

Les avantages de ce modèle sont nombreux. Tout d'abord, il est extrêmement parallélisable : en découpant bien les systèmes, on a souvent l'occasion d'en faire tourner beaucoup en même temps. Aussi, on a énormément de contrôle sur l'ordre d'exécution puisqu'on peut grouper les systèmes, les ordonner et même ajouter des conditions ; sans oublier la programmation événementielle qui m'a été très utile. Enfin, on ne perd quasiment rien en rapidité puisque,

dans le cas de Bevy du moins, son utilisation extensive des tables de hashage fait qu'accéder aux composants d'une entité est presque aussi rapide qu'un accès classique aux données par pointeur.

M'étant intéressé à cette façon de penser lors de mes expériences avec Rust au préalable, j'ai pu rapidement faire la transition vers Bevy et le code s'en est retrouvé plus clair, plus facile à tester et je disposais alors de l'ensemble du moteur de rendu qui vient avec Bevy. Le seul inconvénient, ce sont les temps de compilations qui sont devenus considérablement plus longs, ce qui peut-être très frustrant quand on a l'habitude des scripts Python qui sont interprétés en direct.

C'est dans cette phase que j'ai le plus appris, puisque j'ai pu voir en détail à quoi ressemblait un projet Rust à grande échelle comme Bevy, et comprendre les gros choix de design et les petites astuces d'optimisation qui y sont faits.

Pendant toute la durée du projet, j'ai porté beaucoup d'attention à comprendre les outils et abstractions que j'utilise, de sorte à apprendre autant que si je les avais écrits de mes mains.

1.5 Préparation au multijoueur

La dernière chose que j'ai pris le temps de faire avant de passer à la mécanique des vaisseaux, c'est m'assurer que le jeu puisse fonctionner en multijoueur.

J'ai alors pu appliquer les connaissances que j'ai acquises lors d'un précédent projet sur l'intranet MPCI pour mettre en place un prototype d'architecture client-serveur. En particulier, j'ai utilisé la crate `bevy_quinnet` qui se base sur le protocole QUIC [3] pour apporter à Bevy des éléments de communication réseau via des canaux. Grâce à l'ECS qui nous permet d'exécuter certains systèmes sous des conditions précises (par exemple si on est client ou serveur), il est assez simple d'écrire un code très peu redondant qui permette de faire tourner un serveur autoritaire (c'est-à-dire qui vérifie périodiquement que les clients sont à jour, et qui impose son état sinon) qui serve d'intermédiaire entre les clients.

Ce qui m'a demandé plus de réflexion en revanche, c'est de faire en sorte que les simulations soient déterministes, i.e. qu'elles s'exécutent de façon identique sur deux machines différentes à des moments différents. Pour cela, j'ai utilisé un système de "ticks" :

- Un tick de simulation (ou `simtick`) est le pas de temps du moteur physique. Il représente par exemple 1h dans le jeu, et le nombre de simticks écoulé depuis le début de la partie permet de décrire entièrement l'état des astres. Du côté des vaisseaux, on verra que c'est également le pas de temps de l'intégrateur, ce qui nous assure qu'après un certain nombre de simticks, pour une position et une vitesse initiale donnée, on obtiendra toujours le même résultat (même si la simulation tourne plus lentement sur une machine que sur une autre).
- Un tick serveur (ou simplement `tick`) est un nombre fixé de simticks tel que lorsque le serveur passe au tick suivant, il va envoyer une mise à jour et vérifier que tout se passe bien pour les clients (pour les astres il va simplement envoyer le tick courant et les laisser mettre à jour leur simtick si besoin). C'est par exemple le pas temps atomique entre deux manoeuvres pour les vaisseaux, et je l'utilise aussi pour faire les calculs plus lourds comme la mise à jour corps influenceurs (cf 2.2).

J'aurai dû passer par plusieurs idées de systèmes avant celui-là mais il a l'air de convenir pour

l'instant, même s'il n'est pas parfait (par exemple, on ne peut pas faire de manoeuvre qui demande une précision inférieure à un simtick).

En bref, j'ai compris que les machines se parlent mieux quand le temps est un entier.

1.6 État intermédiaire de l'application

A la fin de cette première partie (à environ un mois dans le stage), l'application est seulement descriptive : elle permet de consulter la position des astres à un instant donné (ainsi que d'autres données utiles via des menus) et on peut observer leur mouvement en temps réel, de façon synchronisée via un serveur. Pour qu'elle devienne vraiment interactive, il reste encore à ajouter les vaisseaux, et en particulier les changements de trajectoires.

Partie 2

Simulation et édition des trajectoires de vaisseaux

Une fois que le système solaire était prêt, j'ai pu m'attaquer aux trajectoires des vaisseaux. Dans un premier temps, je me suis concentré sur l'ajout d'un objet passif, qui "flotte" simplement entre les astres, affecté par le champ de gravitation. Dans les faits, on cherche donc à calculer les positions successives d'un objet soumis à plusieurs forces.

2.1 Description physique du problème et méthodes d'intégration

On se place dans le référentiel du Soleil (supposé galiléen) et on considère un système de masse m négligeable devant celles $(M_i)_i$ des astres, soumis uniquement à la force de gravitation qui s'écrit ainsi :

$$\vec{F} = \sum_i \mathscr{G} \frac{mM_i}{||\vec{r} - \vec{r_i}||^3}(\vec{r} - \vec{r_i})$$

avec $\mathscr{G}$ la constante de gravitation universelle, $\vec{r}$ la position du système et $(\vec{r_i})_i$ les positions des astres. En appliquant le principe fondamental de la dynamique, on obtient donc une équation différentielle du second ordre :

$$\frac{\mathrm{d}^2\vec{r}}{\mathrm{d}t^2} - \mathscr{G} \sum_i \frac{M_i}{||\vec{r} - \vec{r_i}||^3}(\vec{r} - \vec{r_i}) = 0$$

Ne sachant pas résoudre cette équation analytiquement, j'ai du choisir une méthode de résolution numérique.

Après les cours d'initiations aux méthodes numériques du 3e semestre de la licence, un choix naturel était le schéma d'Euler explicite. Cependant, après quelques essais j'ai rapidement constaté que les vaisseaux avaient tendance à sortir spontanément de leur orbite lorsque je les positionnais autour de la Terre. J'ai d'abord cru que c'était l'attraction du Soleil qui perturbait leur mouvement, mais en réessayant autour du Soleil et sans aucune planète, le même phénomène se produisait.

J'en ai conclu que les erreurs d'intégration ne se compensaient pas, et avaient au contraire tendance à s'accumuler pour augmenter l'énergie du système. Après un peu de recherche, j'ai appris qu'il existe une classe d'intégrateurs appelés *symplectiques* qui ont la propriété de conserver

une certaine quantité caractéristique des systèmes Hamiltoniens (c'est-à-dire ceux qui suivent les lois de la mécanique). Cette qualité fait qu'en moyenne l'énergie du système est plutôt bien conservée même avec un pas de temps non négligeable devant la période du mouvement, et le résultat reste convaincant après une longue simulation.

En particulier, la méthode que j'ai choisie est l'intégrateur "saute-mouton" (ou "leapfrog" en anglais) d'ordre 2 qui est présenté en détail dans un article sous le nom de "méthode de Strörmer-Verlet" [12] et qui y est prouvé être symplectique. En effet, après l'avoir implémenté, le problème d'instabilité de l'orbite avait disparu et le vaisseau restait sagement sur sa trajectoire circulaire.

Pour une comparaison plus visuelle entre cette méthode et celle d'Euler, on peut trouver une animation sur la page Wikipédia de l'intégration leapfrog [13].

2.2 Sphère d'influence et optimisations

Une fois la théorie établie et l'implémentation primitive en place, j'ai pu passer aux différentes optimisations,.

Le plus gros problème à régler étant celui de la somme sur tous les astres lors du calcul de la force de gravitation : si un vaisseau est en orbite autour de la Terre par exemple, il est totalement inutile de calculer la contribution de Neptune sur l'accélération.

On comprend immédiatement qu'on a besoin d'une distance à partir de laquelle un astre n'est plus considéré comme un influençant la trajectoire du vaisseau (autrement, on dit que l'astre est un **influenceur**). Il y a plusieurs possibilités pour ce choix, celle que j'ai choisie étant la **sphère de Hill**. Le principe est le suivant :

Supposons qu'un vaisseau V soit en orbite autour d'un astre A, enfant d'un astre B. On cherche à savoir la distance VA à partir de laquelle le vaisseau sera attiré d'avantage par B que par A. Il suffit donc de trouver le point où la force d'attraction de B sur V compense exactement la force d'attraction de A sur V ajoutée à la force centrifuge. On obtient ainsi le rayon de Hill, à partir duquel il est acceptable de considérer que le vaisseau n'est plus soumis à l'attraction de A, c'est-à-dire que A n'est plus un influenceur.

Si on y réfléchit, c'est une approximation assez grosse puisque lorsque le vaisseau sort de la sphère de Hill, il passera instantanément d'un état d'équilibre des forces à un état plus ou moins violemment déséquilibré en faveur de l'astre parent. Pour limiter cet effet, on pourrait définir le rayon d'influence comme deux ou trois fois le rayon de Hill, mais après quelques observations j'ai conclu qu'un rayon de Hill était suffisant.

Pour ne pas avoir à retrouver les corps influenceurs à chaque simtick (c'est là qu'il faut parcourir tous les astres), j'ai choisi de les stocker dans un composant qui est mis à jour tous les ticks (avec 1 tick = 10 simticks) puis de calculer l'accélération en parcourant cette liste, qui en général comporte moins de 5 astres. La complexité du calcul s'effondre, pour une précision qui reste complètement correcte ; et le tout en conservant le caractère déterministe puisque les ticks sont synchronisés sur toutes les machines et sont basés sur les simticks.

2.3 Affichage de prédictions des trajectoires

A cette étape, l'essentiel du moteur physique est en place, et il ne reste plus qu'à transformer un simulateur en un jeu.

J'ai donc commencé par permettre de calculer une prédiction de la trajectoire à partir d'une position et d'une vitesse initiale, qui est particulièrement utile pour se placer sur une orbite spécifique. Dans les faits, construire cette prédiction revient simplement à faire toutes les simulations en avance, c'est-à-dire faire avancer un temps virtuel, déplacer les astres et calculer une étape d'intégration, en prenant soin d'utiliser les bon influenceurs.

Cependant, le fait qu'on souhaite prédire la trajectoire d'un unique vaisseau introduit une notion de localité : si il est en orbite autour de la Terre, il est inutile de simuler les mouvements des nombreux satellites de Jupiter par exemple. En fait, j'ai remarqué que l'on pouvait s'en sortir en simulant seulement les influenceurs, ainsi que les enfants de l'influenceur principal (où l'**influenceur principal** est celui qui possède le plus petit rayon de Hill).

Par exemple, si le vaisseau orbite la Terre, alors on choisit de simuler le Soleil, la Terre (ses deux influenceurs) ainsi que la Lune (l'unique enfant de la Terre, l'influenceur principal), ce qui nous permettra de faire des manoeuvres autour de la Lune si on le souhaite. On remarque qu'on n'a pas besoin de simuler le mouvement des autres planètes puisque leurs sphères de Hill n'intersectent pas celle de la Terre. Si le vaisseau quitte l'influence de la Terre, alors l'influenceur principal devient le Soleil, et il suffit de simuler le mouvement des planètes (ainsi que des astéroïdes, etc.) mais pas des lunes, puisque pour entrer dans la sphère d'influence d'une lune il faut d'abord entrer dans celle de sa planète parent.

En fait, je n'ai trouvé qu'un seul cas où cette approximation pose problème : celui des astéroïdes géocroiseurs. En effet, les astéroïdes, qui sont des enfants du Soleil comme les planètes, peuvent avoir une orbite qui croise celle de la Terre et donc une sphère de Hill accessible depuis le vaisseau sans que l'astéroïde soit simulé. Cependant, leur masse est suffisamment faible pour avoir une attraction négligeable sur le vaisseau, et j'ai choisi de les ignorer pour ces prédictions.

Cette localité prise en compte, il est assez simple de faire le calcul des prédictions, et l'affichage se fait en créant de nouvelles entités et en ramenant toutes les positions par rapport à l'astre de notre choix. Le résultat est assez satisfaisant (figure 2.3).

2.4 Noeuds de manoeuvre et propulsions instantanées

Dans cette dernière partie, je vais détailler les mécanismes que j'ai mis en place pour que l'utilisateur puisse modifier la trajectoire du vaisseau.

Je me suis grandement inspiré du jeu Kerbal Space Program pour établir le système de noeuds de manoeuvre.

Le principe est le suivant : une **manoeuvre** est une suite de différents **noeuds**, et chaque noeud est une poussée instantanée dans une certaine direction à un temps précis (c'est-à-dire un changement du vecteur vitesse). Une telle structure peut-être représentée comme un dictionnaire, dont les clés sont des ticks et les valeurs sont des vecteurs à ajouter à la vitesse aux ticks correspondants. Ce dictionnaire est ordonné par temps croissant, de sorte qu'on puisse appliquer les poussées en un temps minimal à l'aide d'un itérateur. Ainsi, la structure `BTreeMap` de la librairie standard de Rust convient très bien, d'autant plus qu'on peut la sérialiser facilement (dans notre cas au format Toml [14]).

Un objectif important pour ce système de manoeuvres, c'est qu'elles soient réutilisables. Autrement dit, on doit pouvoir réutiliser un fichier de manoeuvre dans différentes conditions (temps, distance d'orbite, astre parent, etc.). Un bon exemple est le *transfert de Hohmann* (figure 2.1) qui consiste à passer d'une orbite circulaire à une autre via une orbite elliptique

intermédiaire et deux poussées instantanées dans la direction prograde (dans le sens du mouvement).

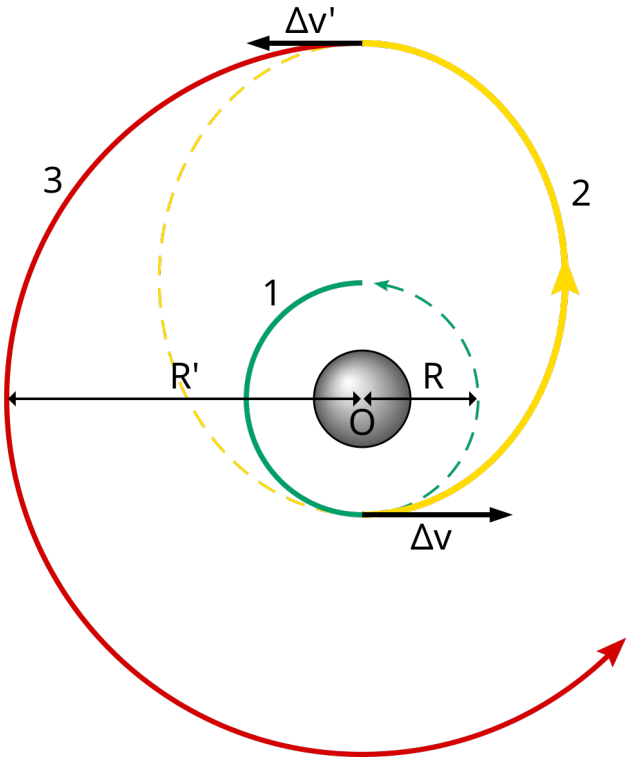

FIGURE 2.1 – Transfert de Hohmann (Par Leafnode — Own work based on image by Hubert Bartkowiak, CC BY-SA 2.5, https ://commons.wikimedia.org/w/index.php ?curid=1885233)

Pour rendre ce transfert réutilisable, il est donc judicieux d'exprimer les coordonnées de la poussée dans une base adaptée, et qui contient en particulier le vecteur tangent à la trajectoire (même direction que la vitesse, de norme 1). Après un peu de gymnastique du produit vectoriel, on peut construire une base "locale" telle que le transfert de Hohmann soit simplement une succession à intervalle bien calculé de deux poussées dans la direction du vecteur tangent. Pour faire la même manoeuvre plus tard, il suffit donc de "déplacer" le noeud de manoeuvre dans le temps, et le programme se chargera de traduire les poussées dans la base locale en vecteurs dans la base globale via un produit matriciel.

Finalement, il m'a suffit de faire une interface graphique simple qui s'intègre avec le système de prédictions pour obtenir un moyen pratique de contrôler la trajectoire du vaisseau et de réaliser des manoeuvres plus ou moins complexes, comme une mise en orbite autour de la Lune depuis la haute atmosphère terrestre (vous pouvez trouver la vidéo sur YouTube à l'adresse https://youtu.be/oBsKqAGuwC8).

2.5 État final de l'application

Après les deux mois passés à développer cette application, elle permet (en plus des fonctionnalités décrites dans la section 1.6) de placer des vaisseaux dans le système solaire à des coordonnées bien précises (que le programme peut calculer automatiquement pour les mettre

en orbite), puis prédire les positions futures et les altérer avec des propulsions instantanées bien planifiées. Le serveur décide quand cette phase de préparation s'achève, et lance alors un signal à tous les clients de commencer la simulation, au cours de laquelle les astres ainsi que les vaisseaux suivent leur mouvement en temps réel. Pour que les joueurs ne puissent pas connaître à l'avance les manoeuvres des adversaires (dans l'optique d'un futur jeu de commerce/combat), elles sont stockées sur le serveur seulement, et les propulsions sont communiquées en temps réel. Enfin, la simulation prend fin, et les joueurs peuvent à nouveau planifier des manoeuvres et créer d'autres vaisseaux.

Conclusion

Pour conclure, j'ai pu écrire étape par étape la base d'un jeu d'exploration spatiale en multijoueur dans le système solaire. J'ai commencé par diviser le travail en deux parties distinctes, qui m'auront chacune pris un mois. En premier lieu, j'ai construit une simulation des astres à partir de données réelles, en séparant les tâches le plus possible afin de paralléliser les calculs. Par la suite, j'ai utilisé une méthode numérique bien choisie pour résoudre l'équation différentielle du mouvement des vaisseaux, et par un ensemble d'approximations justifiées j'ai pu minimiser les calculs et obtenir des prédictions sur un intervalle de temps suffisamment large. Enfin, en m'inspirant des interfaces déjà établies dans ce domaine, j'ai implémenté un système de propulsions instantanées, en prenant en compte l'aspect multijoueur à chaque étape du développement.

Dès le début du stage, il était clair que le projet ne serait encore qu'un embryon à la fin des deux mois, et les objectifs à court terme étaient plus ou moins flous. Pourtant, on arrive maintenant à distinguer les prochaines étapes du développement. En particulier, il serait utile de disposer d'un système d'évènements auxquels les utilisateurs pourraient s'abonner, avec des réactions automatiques lorsque la flotte commence à s'étendre. Aussi, pour accroître la modularité, un accès aux données de simulation depuis une API serait intéressant.

Pour prendre un peu de recul, ces deux mois de stage m'ont permis de me consacrer à temps plein sur un projet qui m'a passionné. J'ai eu l'occasion d'apprendre à gérer un projet plus ambitieux qu'à mon habitude, mais surtout de réfléchir par moi-même sur de nombreuses problématiques que je rencontrerai à nouveau dans ma carrière scientifique, qui ont été aussi larges que le calcul parallèle mais également aussi spécifiques que l'intégration symplectique. A l'occasion, j'ai pu utiliser en pratique les outils que j'ai rencontrés durant mes cours de L2, en particulier l'algèbre linéaire et les structures arborescentes.

Finalement, j'ai décidé de mettre à disposition le jeu en Open Source, afin de maximiser les chances que son développement soit poursuivi, en espérant que le choix assez audacieux du langage Rust ne soit pas une barrière pour ce projet innovateur.

Bibliographie

[1] *Rust Programming Language*. en-US. URL : https://www.rust-lang.org/ (visité le 06/08/2024).

[2] *Git - Reference*. URL : https://git-scm.com/docs (visité le 06/08/2024).

[3] *IETF QUIC Working Group*. en. URL : https://quicwg.org/ (visité le 10/08/2024).

[4] *Approximate Positions of the Planets*. URL : https://ssd.jpl.nasa.gov/planets/approx_pos.html (visité le 01/08/2024).

[5] *L'OpenData du Système solaire*. fr. URL : https://api.le-systeme-solaire.net/ (visité le 10/08/2024).

[6] *JSON*. URL : https://www.json.org/json-en.html (visité le 10/08/2024).

[7] *Euler angles*. en. Page Version ID : 1237676212. Juill. 2024. URL : https://en.wikipedia.org/w/index.php?title=Euler_angles&oldid=1237676212 (visité le 10/08/2024).

[8] *Ratatui*. en. URL : https://ratatui.rs/ (visité le 10/08/2024).

[9] Robert Nystrom. *Game programming patterns*. eng. OCLC : 1002670025. Los Gatos : Robert Nystrom, 2014. ISBN : 9780990582915.

[10] *Bevy Engine*. en. URL : https://bevyengine.org// (visité le 11/08/2024).

[11] *Overwatch Gameplay Architecture and Netcode*. URL : https://www.youtube.com/watch?si=swQzGBvrWSlgP39T&v=W3aieHjyNvw&feature=youtu.be (visité le 11/08/2024).

[12] Ernst Hairer, Christian Lubich et Gerhard Wanner. « Geometric numerical integration illustrated by the Störmer–Verlet method ». en. In : *Acta Numerica* 12 (mai 2003), p. 399-450. ISSN : 0962-4929, 1474-0508. DOI : 10.1017/S0962492902000144. URL : https://www.cambridge.org/core/product/identifier/S0962492902000144/type/journal_article (visité le 12/08/2024).

[13] *Leapfrog integration*. en. Page Version ID : 1220184662. Avr. 2024. URL : https://en.wikipedia.org/w/index.php?title=Leapfrog_integration&oldid=1220184662 (visité le 12/08/2024).

[14] *TOML : Tom's Obvious Minimal Language*. URL : https://toml.io/en/ (visité le 10/08/2024).

Annexe

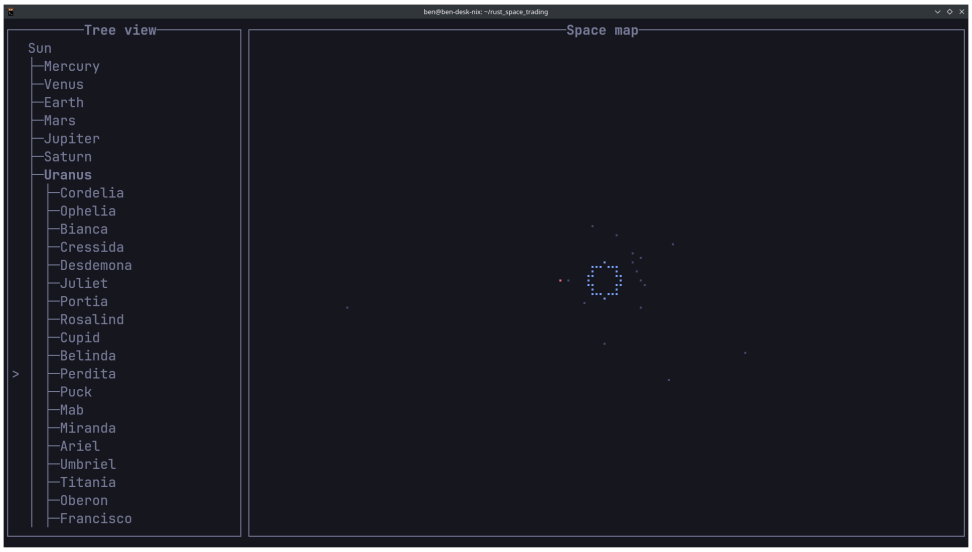

FIGURE 2.2 – Vue dans le terminal d'Uranus ainsi que quelques-uns de ses satellites

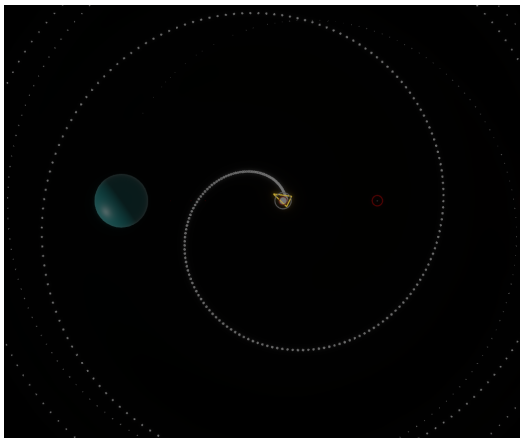

FIGURE 2.3 – Trajectoire d'un vaisseau initialement en orbite autour de Io, mais subissant l'attraction de Jupiter jusqu'à en être sorti de son orbite. Les positions sont ramenées par rapport à Io (chaque point représente une position future du vaisseau sur l'écran si Io restait au centre pendant toute la manoeuvre)