|  |
| --- |
| Atelier |
| Pre-TPI |
| Astral Simulation |

|  |
| --- |
| Roth Ethan, Comtesse Evan  3IND-3TPMa |

Table des matières

[Introduction 2](#_Toc187688598)

[Choix du projet 2](#_Toc187688599)

[Objectif final de la simulation 3](#_Toc187688600)

[Stockage des données 3](#_Toc187688601)

[Implémentation des astres 3](#_Toc187688602)

[Fonctionnalités 3](#_Toc187688603)

[Déplacements 3](#_Toc187688604)

[Aspect physique 4](#_Toc187688605)

[Valeurs utilisées et conversion 4](#_Toc187688606)

[Calculs et fonctions 4](#_Toc187688607)

[Fonctions de calcul du rayon 5](#_Toc187688608)

[Fonction de calcul de la vitesse angulaire 5](#_Toc187688609)

[Fonction de calcul de la rotation sur l’axe Z (roll) 5](#_Toc187688610)

[Sauvegarde d’anciennes fonctions 5](#_Toc187688611)

[Aspect graphique 6](#_Toc187688612)

[Post-Processing 6](#_Toc187688613)

[Éclairage 6](#_Toc187688614)

[Difficultés rencontrées 8](#_Toc187688615)

[Protocole de test 8](#_Toc187688616)

[Conclusion 8](#_Toc187688617)

[Sources 9](#_Toc187688618)

# Introduction

Dans le cadre de cet atelier, le travail attendu était de fournir un projet en binôme d’une valeur de cent période, dans le domaine de la programmation. Nous avons alors choisi de réaliser un projet à l’aide de la librairie graphique Raylib. Bien que la librairie soit originairement écrite en C, un utilisateur du nom de [Chris Dill](https://github.com/raylib-cs/raylib-cs) s’est affairé depuis plusieurs année à créer un binding en C# pour celle-ci, ce qui nous a permis de l’utiliser sans souci avec nos connaissances actuelles. Deux sujets de projets ont été exigés pour permettre aux enseignants de trancher sur la décision de celui qui serait réalisé. En fin de compte, un seul verrait le jour et sa confirmation nous a été donnée rapidement.

## Choix du projet

Comme précisé, il a nous a été demandé d’amener deux idées de projet qui seraient réalisables. Les voici :

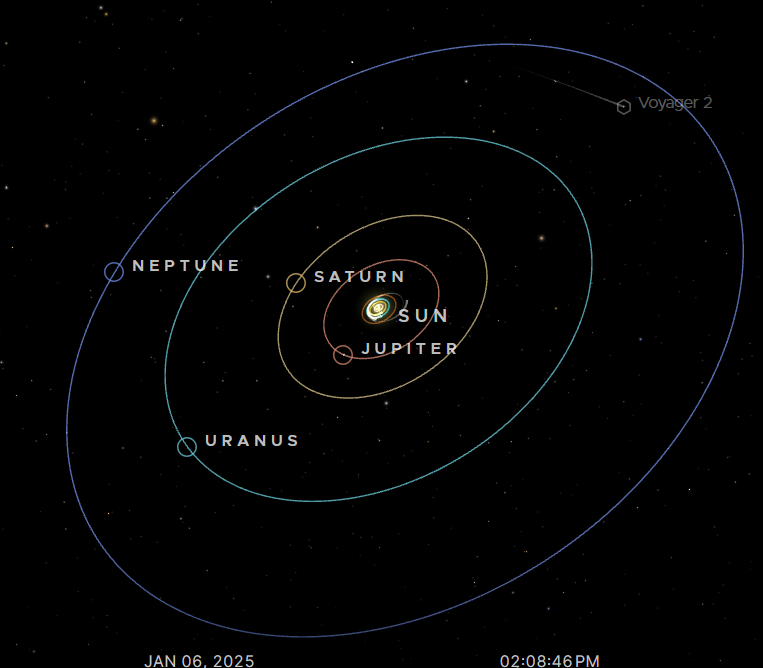
* **Simulation d’un système solaire**

Figure : Simulation officielle de la NASA : <https://eyes.nasa.gov/apps/solar-system/#/home>

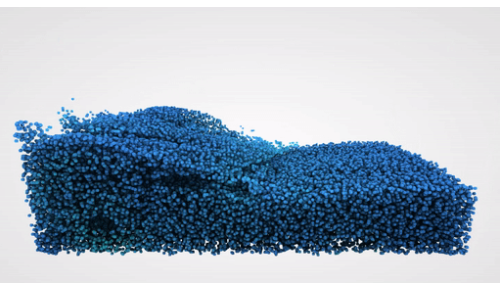
* **Simulation de fluides**

Figure 2: Expérience avec Google : <https://experiments.withgoogle.com/fluid-particles>

Bien que le choix final appartienne aux enseignants, il nous a été autorisé de communiquer notre préférence pour l’un ou l’autre projet et celle-ci a été prise en compte lors de la décision finale. A notre propre enthousiasme, elle a été entendue et avons alors pu commencer à travailler sur le projet qui nous tenait le plus à cœur, à savoir la simulation astrale.

# Objectif final de la simulation

Le but final de la simulation est d’avoir une caméra que nous puissions déplacer librement dans le système solaire afin d’observer les planètes, leur déplacement et leur rotation. Des shaders sur le Soleil seront également ajoutés afin d’avoir un meilleur éclairage ainsi qu’un meilleur rendu. Lors d’un clic sur une planète, une barre latérale est affichée avec diverses informations sur l’objet en question. Grâce au flèches directionnelles, nous pouvons nous déplacer rapidement entre les planètes et obtenir un zoom sur celle-ci ainsi que des informations les concernant.

# Stockage des données

Les données des planètes (rayon, masse, etc.) sont stockées au format JSON, et ce dans un fichier sécurisé par un algorithme de cryptage. (Cette sécurité n’était pas absolument nécessaire. Le choix était plus porté du côté pratique, car l’un d’entre nous avait déjà par le passé écrit des classes en C# qui permettaient de sérialiser des données en JSON et ensuite les sécuriser. Plutôt que de réécrire du code qui permet quelque chose de similaire mais sans la sécurité, nous avons préféré simplement reprendre cette partie.)

# Implémentation des astres

Pour ce qui est des planètes, à la base nous avions téléchargé des modèles 3D depuis le site [Sketchfab.com](https://sketchfab.com). Cependant au niveau de la gestion notamment des positions et des transformations de Matrix4x4, simplement mettre des textures sur des sphères directement proposées par Raylib était une meilleure idée. Chaque texture provient du même site cité ci-dessus.

Au niveau du Soleil, nous avions de base utilisé le même principe que les planètes, mais encore une fois pour une question de gestion, cette fois le Soleil est simplement un shader, expliqué plus bas.

# Fonctionnalités

## Déplacements

Grâce au touches « classiques » de déplacements **W**, **A**, **S** et **D**, nous pouvons nous déplacer librement dans le système.

La touche Espace permet à la caméra de monter et la F de descendre. Grâce aux flèches directionnelles, nous avons la possibilité de nous déplacer de planète en planète, obtenir une vue rapprochée sur celle-ci ainsi qu’une barre latérale avec quelques caractéristiques. Nous pouvons également faire un clic gauche sur une planète pour aussi obtenir le même résultat que précédemment.

# Aspect physique

## Valeurs utilisées et conversion

Pour rester le plus réaliste possible, nous avons choisi que chacune des valeurs utilisées, que ce soit la masse, le volume, le rayon, la distance par rapport au Soleil d’une planète, soient les vraies valeurs fournies ou par le site Wikipédia, ou par le site officiel de la NASA.

Par des soucis de taille, les valeurs de distances telles que la distance par rapport au Soleil ou encore le rayon de la planète sont divisées par 15'000'000 puis multipliées par 20 (la multiplication est un ajustement de dernière minute). Les valeurs telles que le poids, sont également réduites.

Puis pour l’affichage des données, les valeurs sont toutes reformatées par rapport à la réalité et données en notation scientifique si trop grandes (exemple ci-dessous pour Neptune) :

Figure : Reformatage des données pour Neptune

## Calculs et fonctions

Au niveau des calculs de physique, il fallait calculer la trajectoire des planètes, calculer leur rotation, leur vitesse, la distance par rapport au Soleil ainsi que la période de rotation (le temps que met une planète à faire un tour sur elle-même).

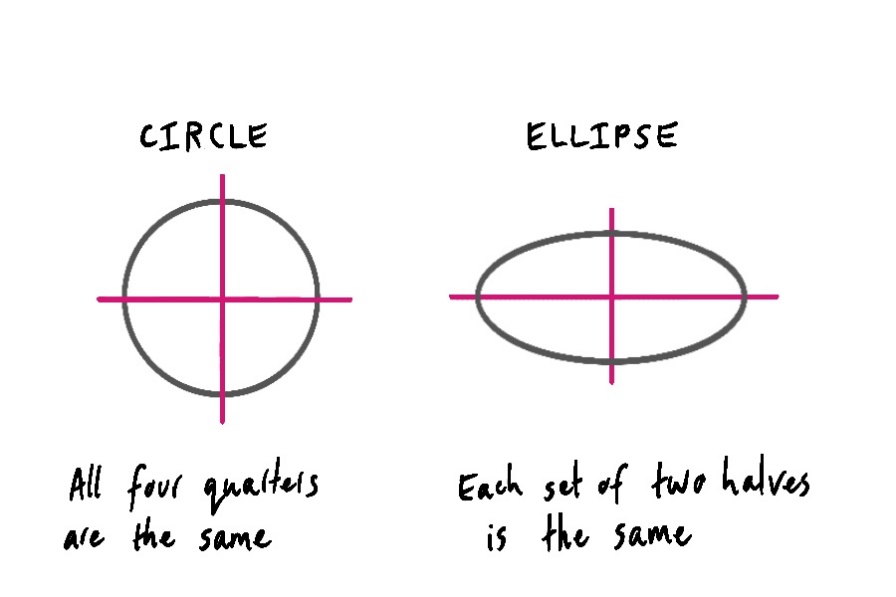
En réalité une planète ne tourne pas en rond autour du Soleil, elle tourne de manière elliptique.

Figure : Cercles et ellipses

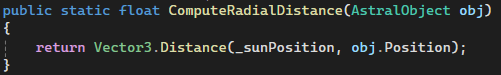
Au début, nous avions pensé et voulu faire le déplacement réel de chaque planète. Toutes les informations se trouvent relativement aisément sur le site de la NASA mais les mettre en place est une toute autre histoire. Ce sont des aspects de physique que nous n’avions jamais traité et les implémenter dans le code pour que cela fonctionne nous a malheureusement été infaisable malgré l’aide de l’IA, d’internet et le temps passé dessus. Nous avons donc pris la décision de faire tourner dans un premier temps les planètes en rond, puis s’il reste du temps, éventuellement repartir sur les ellipses.

Faire se déplacer les planètes en rond est quelque chose de relativement aisé. Un simple MCU (Mouvement Circulaire Uniforme) et c’était terminé. Il suffit d’avoir une période de révolution (p.ex pour la Terre : ~23h et 56min) et un rayon (distance du centre du Soleil au centre de la planète).

### Fonctions de calcul du rayon

Celle-ci calcule la distance entre les deux vecteurs de position des deux astres (**\_sunPosition** étant la position du Soleil et **obj** étant la planète).

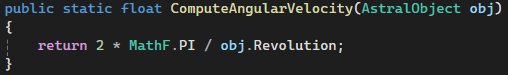
Figure : Fonction de calcul du rayon



### Fonction de calcul de la vitesse angulaire

Celle-ci calcule la vitesse angulaire de la planète grâce à cette équation où est la période de révolution :

Figure : Fonction de calcul de la vitesse angulaire



### Fonction de calcul de la rotation sur l’axe Z (roll)

Celle-ci met à jour la position de la planète sur l’axe Z. La multiplication par 0.0001 est là pour réduire la vitesse de la planète. Sans ça elle tourne beaucoup trop vite.

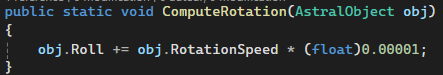


Figure : Fonction de calcul de la rotation

### Sauvegarde d’anciennes fonctions

Toutes les fonctions de calcul de trajectoires des planètes en ellipse sont gardées en commentaire dans l’éventualité où le temps restant est suffisant pour reprendre ce travail.

# Aspect graphique

## Post-Processing

L’attribut le plus visible de l’application est sans nul doute le soleil présent au centre du système. Il peut paraître simple dans son aspect, mais son rendu en arrière-plan est tout autre. En effet, pour réussir un effet de ce type, il est indispensable de passer par des shaders, et plus précisément ceux faisant partie de la catégorie « Post-Processing ». Comme vous le savez certainement, les shaders ont pour but de modifier l’aspect graphique lors du rendu de l’environnement 3D, et ce en prenant en compte toutes sortes d’informations liées aux objets de la scène. Là où les shaders de Post-Processing diffèrent, c’est qu’ils vont appliquer leurs modifications **après** le rendu de l’environnement 3D, en se basant sur ce qui est déjà présent.

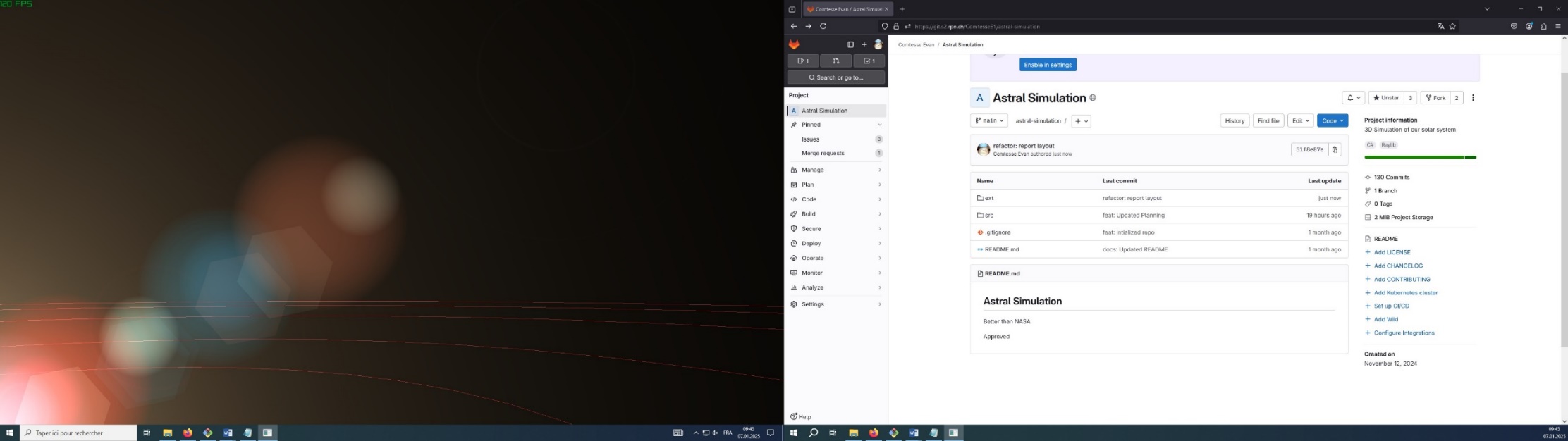
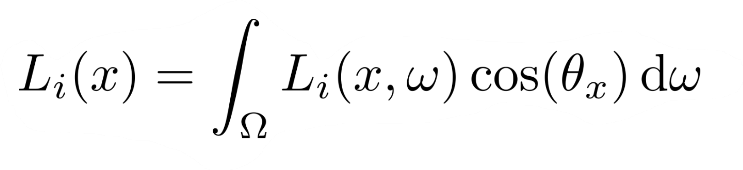
Nous utilisons ici cette technique pour créer le halo lumineux du soleil, mais également ce qu’on appelle communément des « Lens Flares », qui sont à l’origine une aberration optique générée par les objectifs de nos appareils d’observation. Elles ont ici pour but d’augmenter le réalisme de l’effet, pour faire croire à un réel soleil vu au travers d’un instrument optique.

Figure : Lens Flares de notre application, générées par le soleil en bas à gauche

## Éclairage

L’effet présenté au chapitre précédent permet en effet d’obtenir un soleil visuellement réaliste, mais ses interactions ne le sont pas encore. En toute logique, un astre tel que celui-ci produit de la lumière, qui se diffuse dans toutes les directions à partir de son origine. Ce n’est ici pas se qui passe si l’on n’utilise que l’effet de Post-Processing. Il faut donc procéder autrement pour que les planètes affichent une zone d’ombre et une zone éclairée par rapport à leur position respectivement au soleil. En utilisant des shaders plus « classiques », nous pouvons effectuer des calculs par rapport aux positions des astres pour définir les zones d’ombres de ceux-ci. Nous avons pour cela utilisé l’équation de rendu, qui est mondialement connue dans le monde de la programmation graphique :



Il s’agit maintenant de l’expliquer en détail et à l’aide de schémas. Le terme à gauche de l’équation correspond à la couleur finale qui est appliquée au pixel calculé par le fragment[[1]](#footnote-1) shader, exprimée sous la forme d’un vecteur à quatre dimensions. À droite, l’intégrale représente l’infinité de points depuis lesquels la lumière peut être diffusée, bien qu’uniquement un ou deux points soient calculés en informatique pour des raisons évidentes de performances.

Vient ensuite la couleur initiale de la lumière, exprimée ici *Li(x, w)*. On multiplie ensuite ce vecteur par le cosinus de **l’angle d’incidence**. Cet angle est défini par l’intersection entre le vecteur point-lumière et le vecteur normal au point calculé. Et en suivant les lois de l’algèbre linéaire, nous savons que le produit scalaire entre deux vecteurs équivaut au cosinus de l’angle entre ces deux. C’est donc grâce à cette relation que nous calculons cette partie de l’équation.

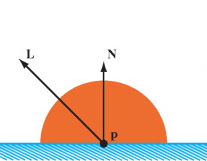
  
Il n’y a ici pas besoin de complexifier l’équation en ajoutant des **BRDF[[2]](#footnote-2)** car l’éclairage est très sommaire, mais il serait envisageable d’en rajouter si nous cherchions un éclairage plus réaliste qui intègre des surfaces métalliques ou autres matériaux particuliers.

Figure : Schématisation de l'équation de rendu graphique

# Difficultés rencontrées

Comme dit plus haut, le calcul de trajectoire des planètes en ellipse aurait été un véritable calvaire à faire au vu du nombre de paramètres à prendre en compte comme le périhélie et l’aphélie, les demi-axes, la longitude du périastre ou encore l’excentricité orbitale pour en citer quelques-uns. L’utilisation du Mouvement Circulaire Uniforme (MCU) était donc la meilleure solution.

# Protocole de test

|  |  |  |
| --- | --- | --- |
| Appuyer sur la touche W | Fait avancer la sonde | OK |
| Appuyer sur la touche A | Fait se déplacer la sonde vers la gauche | OK |
| Appuyer sur la touche S | Fait reculer la sonde | OK |
| Appuyer sur la touche D | Fait se déplacer la sonde vers la droite | OK |
| Appuyer sur la flèche directionnelle de droite une fois | Fait se déplacer la sonde vers une planète dans l’ordre d’éloignement des planètes par rapport au Soleil. Si arrivé à Pluton, revient à Mercure | OK |
| Appuyer sur la flèche directionnelle de gauche une fois | Fait se déplacer la sonde vers une planète dans l’ordre inverse d’éloignement des planètes par rapport au Soleil. Si arrivé à Mercure, revient à Pluton | OK |
| Effectuer un clic gauche sur une planète | Fait se déplacer la sonde vers la planète cliquée | OK |

# Conclusion

Ce projet de simulation nous a permis de mettre en œuvre un large éventail de compétences tant en programmation et calcul qu’en traitement graphique. La librairie que nous avons utilisée nous a permis une implémentation simple et efficace de ce que nous voulions, et ce grâce à l’utilisation de shaders pour le graphisme ou encore de calculs matriciels offerts par sa section « mathématiques » qui nous ont permis d’appliquer des transformations sur nos objets 3D.

Malgré certaines limites que nous avons été contraints d’appliquer, telles que l’adoption de trajectoires simplement circulaires pour les astres, nous avons réussi à produire un outil qui permet la visualisation du système solaire de façon interactive et suffisamment réelle. Le fait d’atteindre ces limites est néanmoins une bonne chose, car cela nous a poussé à approfondir nos connaissances sur certains concepts physiques et autres algorithmes.

En conclusion, ce projet fut une très bonne occasion d’apprendre par nous-même, et n’exclut pas non plus l’idée d’une amélioration une fois nos connaissances sur le sujet agrandies.

# Sources

Pour l’aide au code :

[www.chatgpt.com](http://www.chatgpt.com)  
[www.stackoverflow.com](http://www.stackoverflow.com)

<https://www.shadertoy.com/>

Pour la recherche d’informations sur les astres :

[www.wikipedia.org](http://www.wikipedia.org)  
[www.nasa.gov](http://www.nasa.gov)  
[www.chatgpt.com](http://www.chatgpt.com)

Sources des images du rapport :

* Figure 1 : <https://eyes.nasa.gov/apps/solar-system/#/home>
* Figure 2 : <https://experiments.withgoogle.com/fluid-particles>
* Figure 3 : Capture d’écran de notre programme
* Figure 4 : <https://www.thepencilroomonline.com/wp-content/uploads/2024/05/Untitled_Artwork-9.jpg>
* Figure 5 : Capture d’écran du code
* Figure 6 : Capture d’écran du code
* Figure 7 : Capture d’écran du code

1. Un fragment shader correspond au programme qui s’occupe de définir la couleur pour chaque pixel, et dont les données sont passées depuis le vertex shader. [↑](#footnote-ref-1)
2. Bidirectional-Reflectance-Distribution-Function [↑](#footnote-ref-2)