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

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

Table des matières

[Introduction 2](#_Toc187134234)

[Choix du projet 2](#_Toc187134235)

[Objectif final de la simulation 3](#_Toc187134236)

[Stockage des données 3](#_Toc187134237)

[Aspects physiques du programme 3](#_Toc187134238)

[Valeurs utilisées et conversion 3](#_Toc187134239)

[Calculs et fonctions 4](#_Toc187134240)

[Fonctions de calcul du rayon 4](#_Toc187134241)

[Fonction de calcul de la vitesse angulaire 4](#_Toc187134242)

[Fonction de calcul de la rotation sur l’axe Z (roll) 5](#_Toc187134243)

[Sauvegarde d’anciennes fonctions 5](#_Toc187134244)

# Introduction

Durant cet atelier, nous avions pour travail de choisir un projet à faire à deux. Le sujet étant relativement libre, je me suis mis avec Evan et nous avons choisi de continuer sur la lancée de la première saison en faisant un projet C# en utilisant la librairie graphique Raylib sur Visual Studio. Nous disposions pour ce projet d’une centaine de périodes. Nous avions peu de contraintes pour le projet, si ce n’est de proposer deux idées aux enseignants engagés avec nous dans cet atelier, M. Vadi ainsi que M. Schenk. Après discussion, ils en ont choisi un que nous devions réaliser. Nous sommes quatre groupes dans la classe. Deux sont évalués par M. Schenk, dont notre groupe, et deux par M. Vadi.

## Choix du projet

Nos projets proposés étaient les suivants :

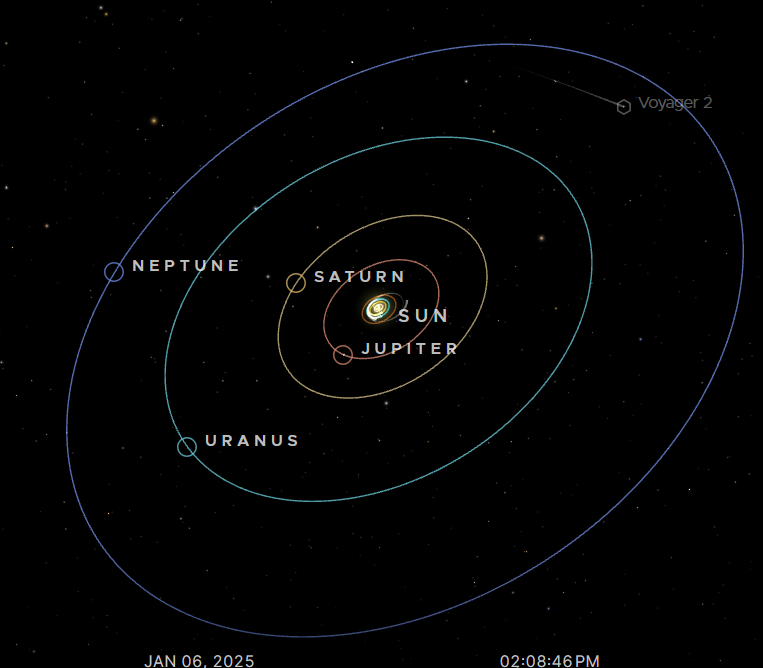
1. Simulation d’un système solaire

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

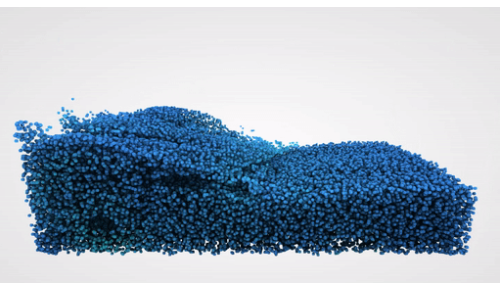
1. Simulation de fluides

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

Bien entendu, sur ces deux idées proposées, une seule a été retenue. Celle de la simulation du système solaire.

# 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 et un meilleur rendu. Lors d’un clic sur une planète, une sidebar est affichée avec diverses informations sur celle-ci. Grâce au flèches directionnelles, nous pouvons nous déplacer rapidement entre les planètes, obtenir un zoom sur celles-ci ainsi qu’à nouveau la sidebar.

# 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.

# 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. 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 notées 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).

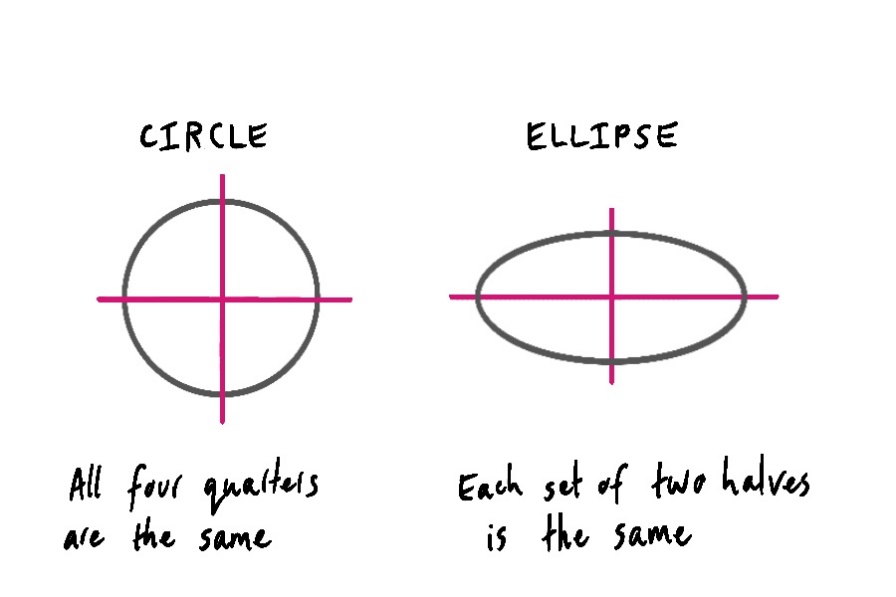
En réalité une planète ne tourne pas en rond autour du Soleil, elle tourne de manière elliptique.

Figure : Cercles et ellipses

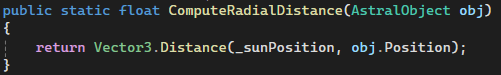
Au début, j’avais (Ethan) pensé et voulu faire le déplacement réel de chaque planète. Toutes les informations se trouvent relativement aisément sur le site de la NASA mais les mettre en place est une toute autre histoire. Ce sont des aspects de physique que je n’ai jamais traité et les implémenter dans le code pour que cela fonctionne m’a malheureusement été infaisable malgré l’aide de l’IA, d’internet et le temps passé dessus. J’ai 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 quelques chose d’extrêmement aisé. Un simple MCU (Mouvement Circulaire Uniforme) et c’était terminé. Il suffit d’avoir une période de révolution (p.ex pour la Terre c’est ~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).

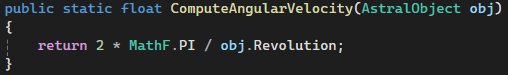
Figure : Fonction de calcul du rayon



### 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.

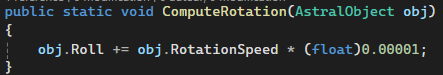


Figure : Fonction de calcul de la rotation

### 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

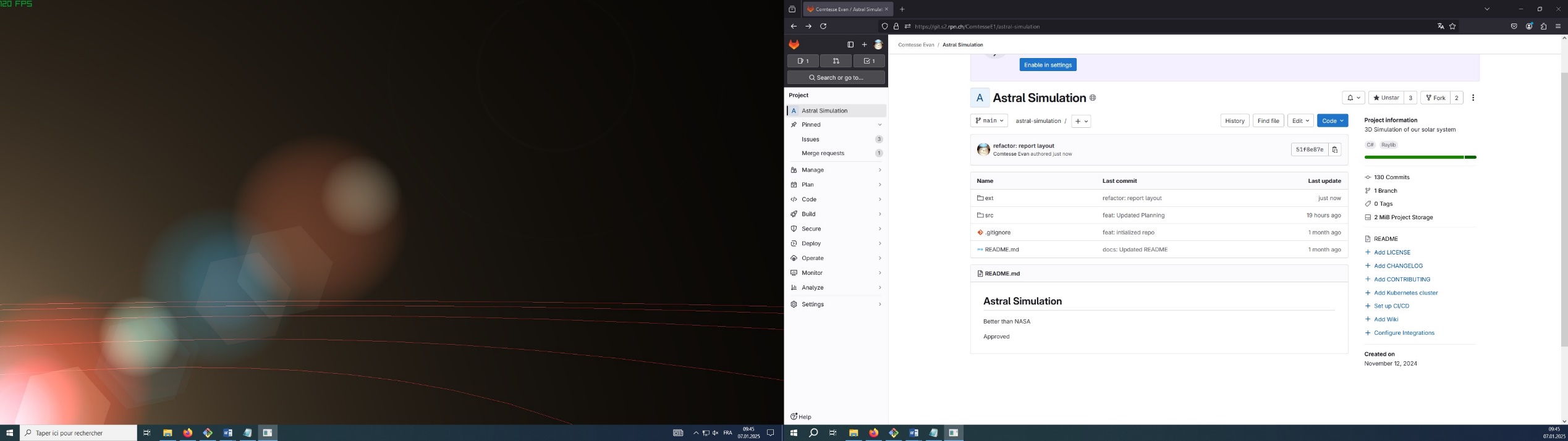
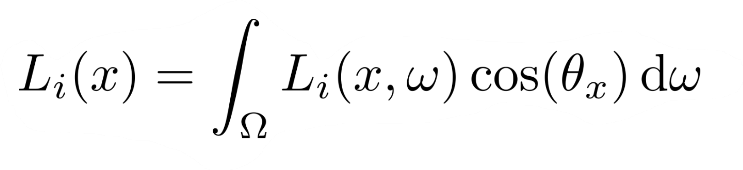
L’attribut le plus visible de l’application est sans nul doute le soleil présent au centre du système. Il peut paraître simple dans son aspect, mais son rendu en arrière-plan est tout autre. En effet, pour réussir un effet de ce type, il est indispensable de passer par des shaders, et plus précisément 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.

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

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.

## 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 :



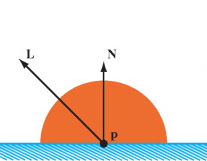
Il s’agit maintenant de l’expliquer en détail et à l’aide de schémas. Le terme à gauche de l’équation correspond à la couleur finale qui est appliquée au pixel calculé par le fragment[[1]](#footnote-1) shader, exprimée sous la forme d’un vecteur à quatre dimensions. À droite, l’intégrale représente l’infinité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

Figure : Schématisation de l'équation de rendu graphique

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.

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)