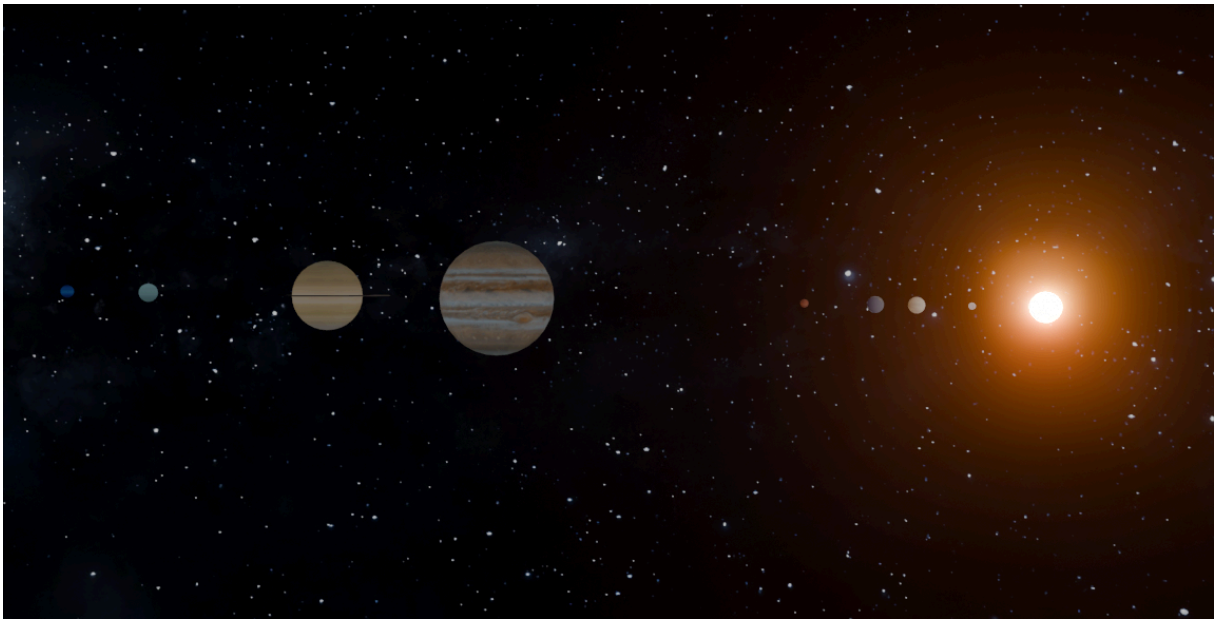
INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN

Projet scientifique encadré

STPI/P6/2024 – 08

Simulation du mouvement des planètes du système solaire



Étudiants :

Léa AURAND **Laurine FORTIN**

Yann-Mathis DELAHAYE **Nathan GRAFF**

Thomas DEMEULES **Flavien MADELON**

Enseignant-responsable du projet :

Alexis LECOMTE

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN

Cette page est laissée intentionnellement vierge.

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN

Date de remise du rapport : ***13/06/2024***

Référence du projet : ***STPI/P6/2024 – 08***

Intitulé du projet : ***Simulation du mouvement des planètes du système solaire***

Type de projet : ***Modélisation***

Objectifs du projet :

L'objectif du projet est de développer une simulation informatique sur le logiciel Blender précise et immersive du mouvement des planètes du système solaire. Cette simulation permettra aux utilisateurs de visualiser en temps réel les orbites des planètes et leur mouvement autour du Soleil. En utilisant des algorithmes, notre objectif est de reproduire le plus fidèlement possible les phénomènes astronomiques, tels que les positions des planètes au cours du temps. De plus, on intégrera plusieurs caméras dans la simulation permettant de suivre les différentes planètes. En résumé, ce projet vise à offrir une expérience immersive sur la dynamique complexe du système solaire.

Simulation - Astrophysique - Trajectoire - Problème à N corps

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN

Table des matières

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN

1. Introduction

Le but de ce projet est de simuler le plus précisément possible le mouvement des planètes du système solaire.
Nous avons dû faire face à plusieurs enjeux afin d'atteindre notre objectif :
Tout d'abord, il a fallu trouver un modèle théorique réaliste restant à portée de nos connaissances. Nos recherches nous ont permis d'identifier l'un des phénomènes physiques les plus difficiles à résoudre : le problème à N corps.
Ensuite, nous avons choisi de réaliser une simulation animée en 3D. Par conséquent, nous avons dû nous familiariser avec le logiciel d'animation Blender.
Enfin, afin d'avoir un regard critique sur notre simulation, nous nous sommes penchés sur l'erreur commise par cette dernière.

2. Histoire

Au cours de l'histoire, les Hommes ont toujours observé le ciel en quête de réponses. Historiquement, l'étude des astres et de notre planète est ainsi une préoccupation majeure au sein de la communauté scientifique. En effet, ces études ont commencé il y a bien longtemps et nous allons voir comment elles nous ont menés à ce que nous savons aujourd'hui :

- **-3000 :** Construction du site de Stonehenge (considéré par beaucoup comme le premier site d'observation) en rapport avec les cycles cosmiques.
- **-2000 :** Premiers calendriers luni-solaires en Égypte et en Mésopotamie.
- **-550 :** Découverte et dénomination des premières constellations par les Grecs. Durant cette même période, Pythagore suppose que la Terre est ronde.
- **-130 :** Hipparque détermine la position d'un millier d'étoiles (moins d'1° d'erreur) grâce à la trigonométrie.
- **140 :** Ptolémée propose un modèle où la Terre se trouve au centre de l'univers.
- **829 :** Le calife Al-Mamun fonde l'observatoire de Bagdad.
- **1543 :** Nicolas Copernic publie *De revolutionibus orbium coelestium* dans lequel il suggère que le Soleil se trouve au centre de l'Univers et non pas la Terre.
- **1609 :** Johannes Kepler présente ses deux premières lois sur le mouvement des planètes dans *Astronomia nova*. Il y évoque notamment l'ellipticité des orbites planétaires.
- **1671 :** Isaac Newton construit le premier télescope réflecteur ce qui permet d'observer le ciel plus précisément.
- **1718 :** Découverte du mouvement propre des étoiles.
- **1854 :** L'hypothèse de l'énergie gravitationnelle du Soleil permet de le dater à 25 millions d'années.
- **1917 :** Prédiction de l'existence des trous noirs (par Karl Schwarzschild en utilisant la théorie de la relativité générale)

On remarque donc que la communauté scientifique n'a cessé de développer son intérêt pour le monde céleste dans l'espoir de répondre à certaines de ses questions sur la nature de notre univers. Dans ce but, des appareils de plus en plus précis et de plus en plus

performants ont été développés pour découvrir de nouveaux éléments qui étaient jusque-là bien au-delà de notre portée. *[3][4]*

3. Méthodologie / Organisation du travail

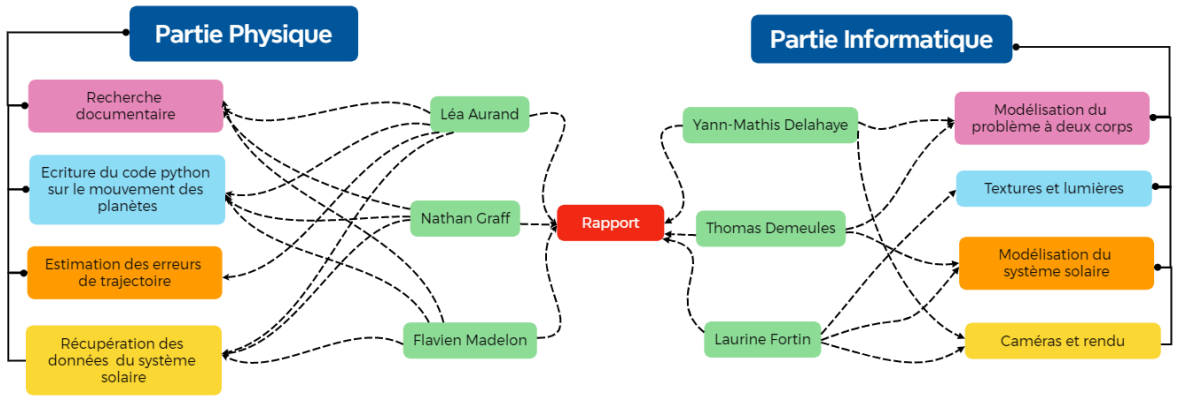
Dès le début du projet, nous nous sommes répartis en deux groupes de trois étudiants selon les compétences et les préférences de chacun : un groupe responsable de la partie physique et un autre de la partie informatique.

Le groupe physique s'est chargé de se renseigner sur les différents moyens de simuler le déplacement des planètes du système solaire. Il a fourni les codes python calculant point par point les mouvements des planètes ainsi que l'évaluation des erreurs commises.

Le groupe informatique quant à lui s'est chargé d'implémenter sur Blender les codes fournis par la partie physique et de créer les visuels de l'animation. Cela a nécessité de se familiariser avec les différentes fonctionnalités de Blender, et en particulier avec le codage en python intégré au logiciel.

Au début de chaque séance un point était effectué entre les deux groupes afin de réaliser ce qui avait été fait par chaque groupe et ce qu'il restait à faire.

Nous avons créé un drive où les différentes recherches et versions des codes pouvaient être partagées aux autres membres du groupe.



Organigramme de la répartition des tâches

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN

4. Problème à n corps

Le problème à n corps consiste à prédire le mouvement de N corps, dans notre cas les planètes du système solaire, tous soumis à la loi de gravitation.

Dans le cas où N=2, il est possible d'établir les équations décrivant la trajectoire des deux corps.

Avec la 2ème loi de Newton et la 3ème loi de Kepler, on arrive au résultat suivant :

$$r = \frac{p}{(1 + e\cos w)}$$

C'est une conique d'excentricité e et de paramètre p.

Pour N=3, les trajectoires des corps deviennent déjà chaotiques ce qui les rend sensibles aux valeurs initiales ; pour des valeurs initiales légèrement différentes on peut obtenir un résultat totalement différent. La difficulté du problème à N corps réside dans l'obligation de devoir approximer la résolution des équations différentielles.

Pour N supérieur à 3, on ne sait plus résoudre les équations différentielles du problème à N corps. De plus, pour chaque corps il faut calculer les N-1 forces exercées sur lui par les autres corps donc le problème a une complexité proportionnelle à N^2.

Avec les outils actuels, il est possible de connaître avec précision les données des planètes du système solaire à un instant précis qui interviennent dans les équations, c'est-à-dire la position, la masse et la vitesse. Cela va nous permettre de simuler les trajectoires des planètes à partir de données initiales exactes. [6] [7]

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN

5. Travail réalisé et résultats

5.1 Simulation point par point

Afin de réaliser cette simulation nous avons dans un premier temps voulu mettre en équation les différents mouvements des planètes. Après avoir pris connaissance du problème à N corps, nous avons pris conscience qu'il n'allait pas être possible de tracer les trajectoires des planètes avec les équations horaires de ces dernières.

Ces premières difficultés nous ont amenés à faire certaines hypothèses :

- Les trajectoires des astres sont considérées planes (pas de composante selon z)
- L'influence des lunes et autres corps stellaires est négligée
- Seules les forces d'interactions gravitationnelles sont modélisés

Ces hypothèses sont discutables étant donné que :

- les orbites des planètes ne sont pas parfaitement situées dans le même plan
- de nombreuses planètes du système solaire possèdent des lunes ce qui engendre des forces de marées par exemple
- d'autres forces sont à l'oeuvre, comme la force électromagnétique

5.1.1 Simulation Soleil, Terre, Mars

La découverte du document "python au lycée" de Arnaud Bodin nous a permis de découvrir la méthode de simulation point par point. [1]

Cette méthode consiste à considérer les planètes comme des particules et de simuler alternativement les forces gravitationnelles s'exerçant sur une planète puis son déplacement. Nous faisons cette hypothèse puisque nous étudions les interactions entre 2 corps séparés par une distance très grande par rapport aux diamètres des astres, isolés du reste du système solaire et possédant une masse. Ainsi, nous obtenons un ensemble de points simulant l'orbite des planètes autour du soleil. Dans un premier temps, nous avons réalisé le code python nécessaire au calcul des positions de la Terre, Soleil et Mars.

À ce stade, nous avons donc dû faire les hypothèses supplémentaires suivantes pour réaliser cette simulation :

- Les planètes sont considérées comme des particules (point possédant une masse)
- Les interactions et mouvements sont mis à jour chaque heure
- Les positions initiales des planètes sont arbitraires. Nous les choisissons sur l'axe x, ainsi y = 0, et nous prenons x tel qu'il soit égal à la distance entre le soleil et la planète lorsque celle-ci est à l'aphélie.
- La vitesse initiale est égale à la vitesse moyenne de la planète sur son orbite. La composante en x du vecteur vitesse est nulle puisque nous considérons que la planète est à l'aphélie ainsi, c'est la composante en y qui est égale à la vitesse moyenne.

INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN

Pour réaliser ce code, nous avons utilisé la programmation objet. Dans notre cas, ce type de programmation s'est traduit par la création d'une classe "Planete" (ci-contre) possédant pour chaque astre les informations suivantes : x, y (pour la position dans le plan), vx, vy (pour la vitesse dans le plan) et m (la masse).

```
class Planete():
    def __init__(self,x,y,vx,vy,m):
        self.x = x
        self.y = y
        self.vx = vx
        self.vy = vy
        self.m = m
```

```
def action_attraction(self,other):
    gx=(G*self.m*other.m*(other.x-self.x))/((math.sqrt((other.x-self.x)**2+(other.y-self.y)**2))**3)
    gy=(G*self.m*other.m*(other.y-self.y))/((math.sqrt((other.x-self.x)**2+(other.y-self.y)**2))**3)
    self.vx +=gx/self.m
    self.vy+=gy/self.m
```

Dans cette partie du code, nous avons simulé l'interaction gravitationnelle entre la planète *self* et la planète *other*. Pour cela, on projette le vecteur $\vec{g} = G\frac{m1*m2}{r^2}\vec{u}$ en coordonnées cartésiennes. Avec les notations de la classe "Planete" et en projetant $\vec{g}$ sur l'axe x, on obtient :

$$gx = G\frac{self.m*other.m}{(other.x-self.x)^2+(other.y-self.y)^2} * \frac{other.x-self.x}{\sqrt{(other.x-self.x)^2+(other.y-self.y)^2}}$$

On modifie ensuite la vitesse de l'astre grâce à la valeur de la force gravitationnelle puis sa position grâce à sa vitesse.

Pour calculer la vitesse, on utilise la méthode d'Euler qui sert à approximer la solution d'une équation différentielle d'ordre 1. Dans notre situation, cela donne la formule suivante : $v_x(t+dt) = v_x(t) + \frac{g_x(t)}{m}$. En effet, $\frac{g_x(t)}{m}$ est la dérivé de v_x par rapport au temps. On a donc : $\lim_{dt\to 0} \frac{v_x(t+dt)-v_x(t)}{dt} = \frac{g_x(t)}{m}$. Ainsi, d'après la méthode d'Euler, on obtient $\frac{v_x(t+dt)-v_x(t)}{dt} = \frac{g_x(t)}{m}$.

De même, pour calculer la position, nous utilisons la formule : $x(t+dt) = x(t) + v(t)$. De la même façon que pour la vitesse, la limite du taux d'accroissement nous donne $\lim_{dt\to 0} \frac{x(t+dt)-x(t)}{dt} = v(t)$ donc, d'après la méthode d'Euler, on a : $\frac{x(t+dt)-x(t)}{dt} = v(t)$.
Comme indiqué dans les hypothèses faites pour notre simulation, nous posons dt=1h. On obtient, de la même façon, les équations pour v_y et y en projetant $\vec{g}$ sur l'axe y.

```
for i in range(3) :
    p=planetes[i]
    for k in range(duree):
        for a in astres :
            if a!=p :
                p.action_attraction(a)
        p.mouvement()
        if k%20000==0:
            plt.plot(p.x,p.y, marker=".", color=couleur[i])
```

Enfin, nous créons une boucle ce qui nous permet, pour chaque planète, de calculer alternativement la somme des forces s'exerçant sur une planète et son déplacement.

On obtient ainsi un ensemble de points représentant la trajectoire des planètes au cours du temps.

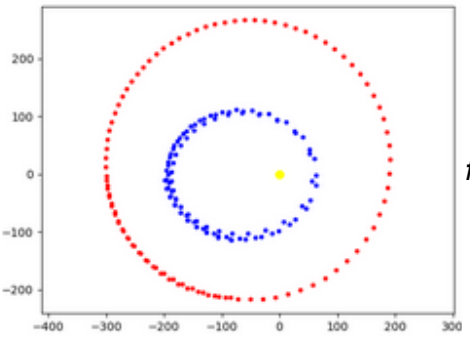


figure 1 : Positions du Soleil, de la Terre et de Mars au cours du temps

Sur la figure ci-dessus, on observe le Soleil en jaune, la trajectoire de la Terre en bleu et celle de Mars en rouge. La période de temps que nous avons simulée correspond à un peu plus d'une orbite de Mars et nous remarquons que la Terre en a réalisé plusieurs ce qui est cohérent avec la réalité puisque la période de révolution de la Terre autour du Soleil est plus courte que celle de Mars.

5.1.2 Généralisation à toutes les planètes

Avec le code précédent, nous avons pu généraliser la simulation à toutes les planètes du système solaire. Pour cela, nous avons ajouté les nouveaux astres dans la classe "Planete" et augmenté considérablement le temps de simulation.

Étant donné qu'une orbite de Neptune autour du Soleil prend une durée de 165 ans, nous ne pouvons pas simuler entièrement l'orbite des planètes les plus éloignées à cause des limites de nos ordinateurs. Du fait des très grandes distances entre les planètes, les orbites des planètes les plus proches du Soleil comme Mercure (gris) et Vénus (violet) sont très peu lisibles.

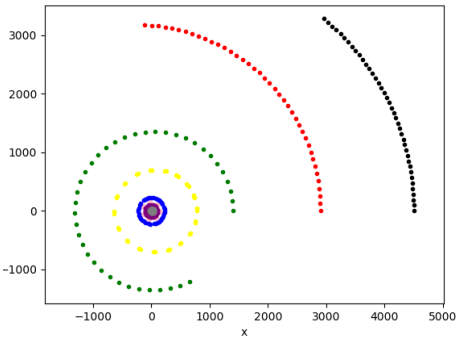


figure 2 : positions des planètes du système solaire

5.1.3 Conditions initiales

Notre simulation était jusque là approximative par rapport aux positions et vitesses réelles des planètes du système solaire. Nous avons ensuite découvert le site *Miriade* qui permet de récupérer la position et vitesse exacte des planètes à un instant donné.

Ainsi, certaines des hypothèses précédentes n'en sont plus :

- les positions initiales des planètes sont exactes
- les vitesses initiales des planètes sont exactes

En effet, nous pouvons désormais récupérer les positions et vitesses des planètes en temps réel ce qui nous permet d'avoir des trajectoires réalistes puisque le point de départ n'est plus arbitraire.

5.1.4 Requêtes à Miriade

La position initiale et la vitesse initiale de chaque planète sont initialisées grâce aux données réelles. Afin de récupérer ces données, nous effectuons des requêtes vers le site *Miriade.* Une requête est une demande faite à un serveur. On reçoit ensuite un objet réponse contenant la réponse du serveur.

Pour effectuer les requêtes, nous avons utilisé le code suivant :

```
import requests, json


def import_donnees(date) :
        url = "https://ssp.imcce.fr/webservices/miriade/api/ephemcc.php?-name=p:earth&-type="+
            "&-step=1d&-tscale=UTC&-observer=@sun&-theory=INPOP&-teph=1&-tcoor=2&-rplane=2&-oscelem=astorb&-mime=json&-output=--iso&-from=MiriadeDoc"
        r = requests.request("GET",url)
        rep = json.loads(r.text)
```

Le module "requests" permet d'effectuer la demande vers *Miriade* et le module "json" de décoder la réponse reçue puisque celle-ci est dans ce format.

L'URL utilisée permet de préciser la demande avec différents paramètres. Par exemple, pour faire la requête pour chaque planète, nous devons changer son nom dans l'URL ici désigné par le paramètre "name". [2]

5.1.5 Estimation des erreurs

Maintenant que nous avons une simulation complète avec des données initiales exactes, nous pouvons estimer l'écart de notre simulation vis à vis de la réalité. Pour vérifier les

trajectoires que nous calculons grâce à notre simulation, nous avons tracé, sur le même graphique, la trajectoire de Mercure calculée grâce à la simulation et la trajectoire réelle de la planète en relevant les valeurs des positions et vitesses sur le site Miriade :

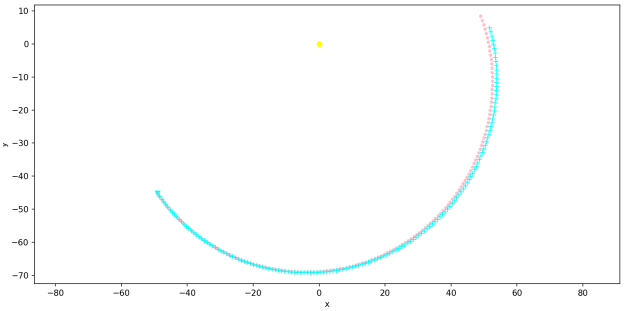


figure 3 : trajectoire réelle (bleue) et trajectoire calculée (rose)

On observe que plus on s'éloigne de la position initiale plus l'écart entre les 2 courbes est important. Cela est dû au fait que les erreurs s'accumulent lorsqu'on calcule les nouvelles positions car chacune est calculée par rapport à la précédente.

Pour autant, l'écart de distance par rapport à la distance de l'orbite à un temps exact reste faible. En effet, nous avons calculé le pourcentage d'erreur pour chaque point grâce à la formule : $\frac{|r_{calculé} - r_{réel}|}{r_{réel}} * 100$. Pour Mercure, elle est au maximum de l'ordre de 3% de la distance Soleil/Mercure pour une durée simulée de 960 heures.

5.2. Logiciel Blender

Le logiciel Blender est très complet et nous avons exploré plusieurs de ses fonctionnalités, telles que l'affichage de textures, la formation d'objets, l'éclairage, les caméras et l'animation, le tout en utilisant Python.

Blender possède plusieurs interfaces permettant de contrôler les différents aspects du projet :

- **Environnement 3D** : Permet de visualiser la scène et les objets qui la composent.
- **Collection** : Permet de voir la liste des objets présents dans la scène et de manipuler les liens entre ces objets.
- **Timeline** : Permet de gérer l'animation. L'animation dans Blender est décomposée en *frames*, avec une vitesse d'animation par défaut de 24 frames par seconde. Une fois l'animation lancée, le mouvement des objets est visible dans l'environnement 3D.

La dernière interface que nous avons utilisé est l'éditeur de texte permettant de rentrer du code Python dans Blender. La bibliothèque Blender est divisée en plusieurs modules, voici ceux que nous avons utilisé :

- **bpy** : Le module principal pour accéder à la plupart des fonctionnalités de Blender.
- **bpy.ops** : Contient les opérateurs, qui sont des fonctions permettant de réaliser des actions (par exemple, ajouter un objet, modifier une géométrie).
- **bpy.data** : Contient les données de Blender (scènes, objets, matériaux, textures, etc.).
- **bpy.context** : Fournit le contexte actuel (scène active, objet sélectionné, etc.).

Cependant, nous n'étions pas tous familiers avec la programmation en Python, et encore moins avec l'API Python permettant de contrôler Blender. Il a donc parfois été difficile d'implémenter les effets souhaités dans le logiciel. Heureusement, Blender offre la possibilité d'accéder à une fenêtre d'information qui affiche le code correspondant aux actions réalisées dans les interfaces actives du logiciel. Cette fonctionnalité s'est avérée extrêmement utile pour surmonter de nombreux obstacles.

5.2.1 Création objets

Nous avons convenu qu'une programmation orientée objet pour ce projet serait la plus optimisée. Il a fallu créer des objets pour les astres et les planètes.

On crée un objet *primitive_uv_sphere_add* pour chaque planète et pour le soleil et on l'associe à l'astre en lui donnant une référence avec l'instruction *bpy.context.active_object*. La sphère est créée à la position des coordonnées de la planète. On lisse ensuite la sphère pour obtenir un meilleur rendu car sans cela la sphère est composée de sections plates.

```
# crée une uv sphère
bpy.ops.mesh.primitive_uv_sphere_add(radius=1, enter_editmode=False, align='WORLD', location=(x, y, 0), scale = scale)

# lisse l'objet
bpy.ops.object.shade_smooth()

# donne une référence à l'objet crée
mesh_obj = bpy.context.active_object
```

Pour créer les anneaux de Saturne on crée un objet en forme de tore dont le centre est le centre de Saturne. Le *minor_radius* est le rayon d'une section du tore et le *major_radius* est la distance entre le centre du tore et le centre d'une section du tore. On a choisi un *minor_radius* petit afin d'avoir un tore de la forme des anneaux de Saturne.

```
#crée les anneaux
bpy.ops.mesh.primitive_torus_add(align='WORLD', location=(saturne.x, saturne.y, 0), rotation=(0, 0, 0),
                                 major_radius=1,  minor_radius=0.05, abso_major_rad=1, abso_minor_rad=0.75)
```

5.2.2 Éclairage et textures

Afin d'obtenir un beau rendu, il était nécessaire d'appliquer des textures et différentes propriétés aux objets. Nous avons pu récupérer les textures des planètes et des anneaux de Saturne sur le site Solar system scope [9] et les importer dans blender.

Sur Blender, appliquer des textures passe par la création d'une matière. Chacune de ces matières contient des nœuds associés à différentes propriétés, telles que la texture, la rugosité ou l'effet métallique.

```
# crée une nouvelle matière
material = bpy.data.materials.new(name=name)

#permet de créer une matière via les noeuds
material.use_nodes = True
```

Afin d'avoir un effet lumineux sur le soleil, il est nécessaire d'agir sur le nœud *Emission*.

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN

```
#connecte la texture à l'objet et change l'intensité de l'éclairage si soleil
if name == "Soleil":
    links.new(texture_node.outputs["Color"], principled_bsdf_node.inputs["Emission Color"])
    nodes["Principled BSDF"].inputs["Emission Strength"].default_value = 450
else:
    links.new(texture_node.outputs["Color"], principled_bsdf_node.inputs["Emission Color"])
    nodes["Principled BSDF"].inputs["Emission Strength"].default_value = 0.2
```

Dans cet extrait de code, les textures images sont associées au nœud *Emission Color*. Afin que les planètes soient visibles même dans les angles non éclairés par le soleil, il est nécessaire de leur faire émettre une lumière moindre. C'est ce qu'on peut observer dans la deuxième ligne de code, qui permet d'ajuster la force d'émission. Le soleil est à 450 tandis que les planètes sont à 0,2, permettant d'avoir un contraste suffisant.

Cependant, la question de l'éclairage du soleil n'a pas été réglée par cette manipulation. En effet, la lumière créée par un objet n'est pas reçue par les autres. Il a donc fallu créer deux lampes, positionnées autour du soleil.

```
#crée les lumieres aux environs du soleil
bpy.ops.object.light_add(type='POINT', radius=1, align='WORLD', location=(0, 0, soleil.r + 2), scale=(1, 1, 1))
bpy.context.object.data.shadow_soft_size = 100
bpy.context.object.data.energy = 10**5

bpy.ops.object.light_add(type='POINT', radius=1, align='WORLD', location=(0, 0, -soleil.r - 2), scale=(1, 1, 1))
bpy.context.object.data.shadow_soft_size = 100
bpy.context.object.data.energy = 10**5
```

Dans ce code,

-l'objet lampe est créé et ses coordonnées sont gérées grâce à *location*. Ici, uniquement sa coordonnée z est modifiée. Afin d'avoir de l'éclairage provenant du soleil depuis n'importe quelle position, nous avons placé deux lampes, l'une au-dessus du soleil, et l'autre en dessous.

- la taille des ombres est ajustée, ici nous prenons la valeur arbitraire de 100

- la puissance de la lumière est choisie. Ici, nous l'avons mis à 100 000 W, car après plusieurs tests, c'était la valeur optimale afin que le soleil éclaire suffisamment sans trop éblouir.

5.2.3 Mouvement des planètes

5.2.3.1 Système composé de la Terre et du Soleil

Lors de notre sélection de Blender pour ce projet, notre principal objectif était de trouver une manière efficace de coder et de visualiser les mouvements planétaires. Nous avons initialement envisagé d'utiliser des équations d'ellipses pour représenter les orbites des planètes, une approche adéquate pour le problème à deux corps. En effet, l'équation de l'ellipse décrivant la trajectoire d'une planète autour du Soleil peut être calculée numériquement dans ce cas précis.

Blender offre la possibilité de créer et de modifier des courbes, ce qui nous permet de représenter les orbites. Dans l'extrait de code suivant, une courbe de Bézier est créée dans l'environnement 3D et modifiée pour adopter les proportions d'une ellipse. Les constantes DEMI_GRAND_AXE et DEMI_PETIT_AXE correspondent aux dimensions réelles de l'orbite elliptique suivie par la Terre autour du Soleil.

```
bpy.ops.curve.primitive_bezier_circle_add(enter_editmode=False, align='WORLD',
location=(APHELIE-DEMI_GRAND_AXE, 0, 0), scale=(1, 1, 1))

trajectoire_terre = bpy.context.active_object
trajectoire_terre.name = "Trajectoire Terre"
trajectoire_terre.scale[0] = DEMI_GRAND_AXE
trajectoire_terre.scale[1] = DEMI_PETIT_AXE
```

Il est maintenant possible de sélectionner l'objet "Terre" et de lui appliquer la contrainte "Follow Path" en définissant l'ellipse précédemment créée comme objet cible. Ensuite, une animation de type "Follow Path" peut être ajoutée à la Terre. Ainsi, au fil de l'animation, la position de la Terre correspondra à un point de l'ellipse, simulant ainsi fidèlement la trajectoire orbitale de la Terre.

```
bpy.context.view_layer.objects.active = bpy.data.objects["Terre"]
terre = bpy.context.active_object
bpy.ops.object.constraint_add(type='FOLLOW_PATH')
terre.constraints["Follow Path"].target = trajectoire_terre
bpy.ops.constraint.followpath_path_animate(constraint="Follow Path", owner='OBJECT')
```

Grâce à cette méthode, nous pouvons modéliser avec précision les trajectoires des planètes et également visualiser leur mouvement de manière dynamique et interactive dans Blender.

5.2.3.2 Système Solaire complet

Cependant, cette méthode ne permet pas de modéliser les interactions entre plus de deux planètes, car les trajectoires suivies par les planètes ne sont pas directement calculables dans un système à plusieurs corps. Les trajectoires des planètes sont interdépendantes et doivent être calculées point par point pour tenir compte des influences gravitationnelles mutuelles.

Pour surmonter cette limitation, nous avons utilisé le calcul des positions point par point fourni par nos collègues de la partie physique. Blender dispose en effet d'un outil d'animation permettant d'insérer une *keyframe* à un moment précis de l'animation, enregistrant ainsi certains attributs d'un objet. En utilisant leur code Python pour calculer les positions des planètes point par point, nous avons pu enregistrer leurs positions à intervalles réguliers de *frames*.

Ensuite, Blender interpole automatiquement les données entre deux *keyframes*, ce qui nous a permis d'animer les planètes de manière fluide entre chaque position calculée. Voici le code utilisé pour implémenter cette méthode dans Blender :

```
for frame in range(DUREE_ANIMATION):
    #sélectionne la frame
    bpy.context.scene.frame_set(frame)
    for a in astres:
        if a != planete:
            planete.action_attraction(a)
    planete.mouvement()
    #associe la nouvelle position de la planète à l'objet lui correspondant
    obj.location[0] = planete.x
    obj.location[1] = planete.y
    #toutes les 50 frames, la nouvelle position de la planète est insérée dans l'animation
    if frame % 50 == 0:
        obj.keyframe_insert(data_path="location", index=0)
        obj.keyframe_insert(data_path="location", index=1)
```

La première ligne permet d'itérer sur les frames de l'animation, la fin de l'animation étant définie par la constante DUREE_ANIMATION en nombre de frames. Ensuite, nous devons itérer sur tous les astres, c'est-à-dire l'ensemble des planètes et le Soleil, afin de calculer leurs positions sur les axes x et y en utilisant les fonctions de la classe "Planete". Toutes les 50 frames, les positions des planètes dans l'environnement 3D de Blender sont mises à jour. Des keyframes sont insérées pour enregistrer les positions en x et y des planètes dans l'animation. Nous avons choisi l'intervalle de 50 frames car nous l'avons estimé le plus rentable en termes de temps de calcul et de fluidité de rendu dans l'environnement 3D.

5.2.3.2 Rotation des planètes

Nous avons également intégré les rotations des planètes du système solaire dans notre code Blender. L'idée était similaire à celle utilisée pour les mouvements des planètes : insérer des *keyframes* pour enregistrer l'angle de rotation de chaque planète autour de l'axe z. Cette partie du code a été ajoutée dans la même boucle que celle utilisée pour les mouvements des planètes, afin d'effectuer les calculs pour chaque astre simultanément.

Voici le partie de code correspondants à ces calculs :

```
obj.rotation_euler.z = (frame % planete.rot) * (6.28319 / planete.rot)
obj.keyframe_insert(data_path="rotation_euler", index=2)
```

La première ligne calcule l'angle de rotation de l'astre autour de l'axe z, en utilisant la vitesse de rotation de la planète, définie dans la classe "Planete" par l'attribut "planete.rot". Cette vitesse de rotation est exprimée en *frames* par tour. Nous avons défini la période de rotation de la Terre à 24 frames, puis ajusté proportionnellement les vitesses de rotation des autres astres en fonction de leurs périodes réelles de rotation. Ainsi, cette formule fait tourner l'objet autour de l'axe z à un rythme constant, complétant une rotation complète (360 degrés ou 2π radians) toutes les périodes de rotation de la planète. La deuxième instruction insère une keyframe pour enregistrer l'angle de rotation autour de l'axe z de l'objet.

5.2.4 Caméras dynamiques et affichage heure

Afin d'avoir un rendu intéressant et modifiable, nous avons créé des objets caméras associés à chaque planète. Chacune d'entre elles est tournée vers le soleil et se situe derrière sa planète associée.

```
#ajoute une contrainte pour que la caméra fasse face au soleil
track_constraint = camera.constraints.new('TRACK_TO')
track_constraint.target = bpy.data.objects['Soleil']
track_constraint.track_axis = 'TRACK_NEGATIVE_Z'
track_constraint.up_axis = 'UP_Y'
```

On utilise pour cela une contrainte nommée "Track to", qui permet de suivre une coordonnée d'un objet sélectionné grâce à la commande "track_axis". Ici, on veut que la caméra se tourne vers le soleil, c'est donc sa coordonnée z qui est fixée par ce code. La commande "up_axis" permet de décider l'orientation de la caméra.

Dû à la nature des mouvements des planètes, les caméras se déplacent de la même manière que celles ci.

De plus, nous avons associé à chacune des caméras un affichage de l'heure en simulation, afin d'avoir une échelle de temps permettant de se rendre compte de la vitesse des planètes. Cet affichage a été particulièrement compliqué à implémenter, car le système de *keyframe* ne fonctionne pas avec le contenu des textes.

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN

```
#appelle update à chaque changement de frame, permet de mettre à jour le contenu des textes
bpy.app.handlers.frame_change_post.append(update)
```

Cette instruction à ajouter dans le programme principal a donc permis de résoudre ce problème; au lieu d'insérer des *keyframes*, quand une animation est lue ou que l'utilisateur change de frame, Blender déclenche l'événement "frame_change_post". La fonction "update", ajoutée à ce gestionnaire avec "append" est alors exécutée. Elle permet de mettre à jour le texte, voici son implémentation :

```
def update(scene):

    frame = scene.frame_current
    if frame % FPS == 0:
        for planete in planetes:
            texte = bpy.data.objects['Heure_' + planete.name]
            heure, minutes, secondes = calcul_temps(frame)
            minuteur_en_chaine = temps_en_chaine(heure, minutes, secondes)
            texte.data.body = minuteur_en_chaine
```

Après avoir récupéré la frame courante, le minuteur est mis à jour à des intervalles réguliers de frames déterminés par la constante FPS. Cette constante définit le nombre de frames correspondant à une seconde dans le minuteur. Le temps en heures est calculé et converti en chaîne de caractères à l'aide des fonctions "calcul_temps" et "minuteur_en_chaine" (voir annexes). Pour chaque planète, le texte affiché dans la caméra est ensuite modifié en fonction du temps calculé.

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN

6. Conclusions et perspectives

Au cours de ce projet, nous avons dû explorer plusieurs méthodes pour mener à bien notre simulation, ce qui nous a parfois amené à rebrousser chemin. Il a également fallu choisir le logiciel sur lequel faire la simulation pour que celle-ci soit immersive et que la programmation soit accessible. Comme dans tout projet, nous avons rencontré un certain nombre de difficulté que nous avons listé ci-dessous :

- Appréhender la pertinence du concept de simulation point par point pour un phénomène complexe. Cela n'a pas été facile étant donné la difficulté du problème à N corps ce qui nous a donc poussé à faire différentes hypothèses.
- Implémenter les résultats physiques dans Blender
- Trouver de la documentation python pour Blender
- Coder en python, car n'ayant appris que le pascal à l'INSA, ce n'était pas une évidence pour les moins informaticiens d'entre nous.
- effectuer des requêtes internet automatisé sur miriade

Conclusion sur l'apport personnel de cet E.C. projet

Il est indéniable que ce projet nous a permis d'acquérir de nouvelles compétences dans différents domaines. Tout d'abord il a été nécessaire de travailler en équipe afin de réaliser les différents objectifs que nous nous sommes donnés. Ainsi nous avons pu améliorer notre gestion du temps et nos facultés de communication afin d'optimiser notre temps de travail et de faciliter notre compréhension des avancées faites par l'autre trinôme.

Ce projet a été formateur à bien des égards. Tout d'abord, nous avons pu développer nos compétences en Python et nous familiariser avec l'implémentation de code sur le logiciel Blender. Nous avons également appris à gérer tout ce qui est graphisme et caméra sur ce logiciel.

Notre projet nous a amenés à expérimenter la simulation point par point pour un phénomène complexe. Nous avons également appris à écrire une requête internet pour récupérer des données sur un site de manière automatique.

Perspectives pour la poursuite de ce projet

Maintenant que nous disposons d'une simulation cohérente et précise du mouvement des planètes, il aurait pu être intéressant de calculer les différents points de Lagrange du système solaire. Ces points, très spécifiques, sont des points où les interactions gravitationnelles se compensent ce qui en fait des lieux très prisés pour les grands télescopes spatiaux tel que le télescope James Webb. Nous aurions également pu simuler les interactions gravitationnelles de manière plus précise, c'est-à-dire prendre en compte les interactions entre les planètes et leurs satellites. Cela aurait complexifié la simulation mais elle aurait été plus réaliste.

7. Remerciements

Nous tenions à exprimer notre profonde gratitude à Alexis Lecomte pour l'accompagnement et le soutien qu'il nous a fournis tout au long de ce semestre. Sa disponibilité et ses précieux conseils ont été déterminants dans la réussite de notre projet

scientifique encadré. Grâce à lui, nous avons pu approfondir nos connaissances et mener à bien nos recherches avec confiance et rigueur.

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN

8. Bibliographie

[1] Arnaud Bodin, “Python au lycée”, publication indépendante, 24 mars 2019

[2] https://ssp.imcce.fr/webservices/miriade/api/ephemcc/ (valide à la date du 30/05/2024)

[3] Jérôme Perez , “Gravitation Classique”, ENSTA ParisTech, 2011

[4] https://www.superprof.fr/ressources/physique-chimie/physique-chimie-tous-niveaux/astronomie.html (valide à la date du 23/05/2024)

[5] https://astronomes.com/chronologie-astronomie/(valide à la date du 12/05/2024)

[6] https://fr.wikipedia.org/wiki/Probl%C3%A8me_%C3%A0_deux_corps (valide à la date du 23/05/2024)

[7] https://fr.wikipedia.org/wiki/Probl%C3%A8me_%C3%A0_N_corps (valide à la date du 23/05/2024)

[8] https://docs.blender.org/api/current/index.html (valide à la date du 07/06/2024)

[9] https://www.solarsystemscope.com/textures/

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN

9. Annexes

9.1 Listings des programmes réalisés

Programme point par point toutes les planètes

```
import math
import requests, json
from matplotlib import pyplot as plt
G = 6.67*10**(-38)*3600**2
UA = 149.597870700
duree=200000
class Planete():
    def __init__(self,x,y,vx,vy,m):
        self.x = x
        self.y = y
        self.vx = vx
        self.vy = vy
        self.m = m

    def action_attraction(self,other):
        gx=(G*self.m*other.m*(other.x-self.x))/((math.sqrt((other.x-self.x)**2+(other.y-self.y)**2))**3)
        gy=(G*self.m*other.m*(other.y-self.y))/((math.sqrt((other.x-self.x)**2+(other.y-self.y)**2))**3)
        self.vx +=gx/self.m
        self.vy+=gy/self.m

    def mouvement(self):
        self.x+=self.vx
        self.y+=self.vy

soleil = Planete(0,0,0,0,2*10**30)
terre = Planete(149,0,0,30*10**(-6)*60*60,6*10**24)
mars = Planete(228,0,0,24*10**(-6)*60*60,6*10**23)
saturne = Planete(1.4*10**3,0,0,9.6*10**(-6)*60*60,5*10**26)
uranus = Planete(2.9*10**3,0,0,7.1*10**(-6)*60*60,8.7*10**25)
neptune = Planete(4.5*10**3,0,0,5.3*10**(-6)*60*60,10**26)
jupiter = Planete(778,0,0,12.4*10**(-6)*60*60,1.9*10**27)
venus = Planete(108,0,0,34.8*10**(-6)*60*60,4.9*10**24)
mercure = Planete(57.9,0,0,38.9*10**(-6)*60*60,3.3*10**23)

astres = (soleil, terre, mars, venus, jupiter, saturne, neptune,uranus, mercure)
planetes = (terre, mars, uranus, jupiter, saturne, neptune, venus, mercure)
couleur = ("pink", "blue", "red", "yellow", "green", "black", "purple", "grey")
```

```
plt.plot(soleil.x,soleil.y, marker="o", color="yellow")
for i in range(8) :
    p=planetes[i]
    for k in range(duree):
        for a in astres :
            if a!=p :
                p.action_attraction(a)
        p.mouvement()
        if k%20000==0:
            plt.plot(p.x,p.y, marker=".", color=couleur[i])
plt.xlabel("x")
plt.ylabel("y")
plt.axis('equal')
plt.show()
plt.close()
```

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN

Extrait du code Blender permettant de simuler le mouvement de la Terre autour du Soleil

```python
def system_creation():

    #création Terre
    scale = (RAYON_TERRE,RAYON_TERRE,RAYON_TERRE)
    terre = add_sphere(scale)
    terre.name = "Terre"
    terre.data.materials.append(create_material("terre"))

    #création Soleil
    scale = (RAYON_SOLEIL,RAYON_SOLEIL,RAYON_SOLEIL)
    soleil = add_sphere(scale)
    soleil.name = "Soleil"
    soleil.data.materials.append(create_material("soleil"))

    return terre and soleil
```

```python
def earth_revolution():

    #creation elipse
    bpy.ops.curve.primitive_bezier_circle_add(enter_editmode=False, align='WORLD', location=(APHELIE-DEMI_GRAND_AXE, 0, 0), scale=(1, 1, 1))
    trajectoire_terre = bpy.context.active_object
    trajectoire_terre.name = "Trajectoire Terre"
    trajectoire_terre.scale[0] = DEMI_GRAND_AXE
    trajectoire_terre.scale[1] = DEMI_PETIT_AXE

    #sélectionne la Terre
    bpy.context.view_layer.objects.active = bpy.data.objects["Terre"]
    terre = bpy.context.active_object

    #lie la Terre à sa trajectoire
    bpy.ops.object.constraint_add(type='FOLLOW_PATH')
    terre.constraints["Follow Path"].target = trajectoire_terre

    #anime la Terre qui tourne autour du Soleil
    bpy.ops.constraint.followpath_path_animate(constraint="Follow Path", owner='OBJECT')

    return trajectoire_terre and terre
```

Extraits du code Blender permettant de simuler le mouvement de l'ensemble des astres du système solaire

```python
def trajectoire_planete(planete):

    #sélectionne la planète et crée l'animation pour l'objet
    obj = bpy.data.objects.get(planete.name)
    obj.animation_data_create()
    obj.animation_data.action = bpy.data.actions.new(name="Animation" + planete.name)

    #sélectionne la caméra associée à la planète et crée l'animation pour la caméra
    camera = bpy.data.objects.get('Camera_' + planete.name)
    camera.animation_data_create()
    camera.animation_data.action = bpy.data.actions.new(name="Animation_camera" + planete.name)

    #boucle sur une durée déterminée
    for frame in range(DUREE_ANIMATION):
        #sélectionne la frame
        bpy.context.scene.frame_set(frame)
        for a in astres:
            if a != planete:
                planete.action_attraction(a)
        planete.mouvement()

        #associe la nouvelle position de la planète à l'objet lui correspondant
        obj.location[0] = planete.x
        obj.location[1] = planete.y

        #fait faire une rotation à la planète
        obj.rotation_euler.z = (frame % planete.rot) * (6.28319 / planete.rot)
        obj.keyframe_insert(data_path="rotation_euler", index=2)

        #fait bouger la caméra
        trajectoire_camera(camera, planete, frame)

        #toutes les 50 frames, la nouvelle position de la planète est insérée dans l'animation
        if frame % 50 == 0:
            obj.keyframe_insert(data_path="location", index=0)
            obj.keyframe_insert(data_path="location", index=1)

    return obj
```

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN

```python
def trajectoires_planetes():
    for planete in planetes:
        trajectoire_planete(planete)


def main():

    #définie la durée de l'animation
    bpy.context.scene.frame_end = DUREE_ANIMATION

    #récupère les données de position sur miriades
    import_donnees()

    #nettoie la scène
    partially_clean_the_scene()

    #crée le fond
    create_background()

    #crée tout le système et ses caméras
    system_creation()
    cameras_planetes()

    #crée tout le mouvement des planètes et de leurs caméras
    trajectoires_planetes()

    #appelle update à chaque changement de frame, permet de mettre à jour le contenu des textes
    bpy.app.handlers.frame_change_post.append(update)


main()
```