Étudiant	Pierre THOELEN, Marco SPALLITTA ; Simon ROOSENS
Cours	M1-FOND0022-1 Fondamentaux Math-Physique
Travail	Rapport Examen – Système Solaire

Table des matières

Introduction 1
Lois de physique appliquées 2
Lois Newtonienne 2
Loi de gravitation Newtonienne 2
Vitesse orbitale circulaire (solution facile) 3
Vitesse orbitale elliptique (solution moins facile) 3
Vitesse angulaire astre 4
Caractéristiques des astres du système Solaire 5
Intégration du mode de jeu 6
Physique du vaisseau 6
Étapes 6
Implémentation 6
Aperçu 6
Système Solaire 7
Code 8
Vaisseau 9
Code 9
Problèmes rencontrés 9
Conclusion 10

Introduction

Dans le cadre du cours de Math & Physique, il nous a été demandé de réaliser un simulateur physique en intégrant les lois connues dans un moteur Unity 3D. Parmi la liste proposée, nous avons choisi le simulateur du système Solaire. Ce dernier est plutôt intéressant et permet de se pencher sur les différents lois et équations permettant cet équilibre fin par les forces de gravité et les vitesses orbitales.

La principale difficulté que nous imagions à ce stade sera de faire respecter au maximum l'échelle du système solaire (le déplacement des astres, le suivi par la caméra, …) tout en appliquant les lois universelles à partir des valeurs réelles et que ça puisse être visualisable et utilisable par le joueur. En effet, le Soleil est par exemple trop volumineux à côté des autres astres si on veut pouvoir les distinguer clairement.

Il existe de nombreux jeux ou simulateurs ludiques qui ont implémenté ce sujet, on peut les distinguer en 2 grandes catégories. La première comprend les jeux d'exploration spatiale (p.ex : Start Citizen) qui appliquent les lois newtoniennes (voir topic newtonian physics in star citizen) mais les thématiques principales sont l'aventure, le commerce spatial et les combats PvP et PvE. La seconde catégorie est quant à elle dédiée à la prise de conscience des différents éléments qui constituent notre système solaire (les lois physiques, les astres & les interactions entre elles). Le joueur prend le rôle d'un observateur et est limité à seulement visualiser l'évolution des orbites et les forces de gravitations au cours du temps. Un de ces simulateurs qui vaut la peine d'être testé est le Solar System Scope. Pour notre projet, nous allons tenté de combiner ces 2 catégories en proposant un simulateur réaliste combiné à un mode de jeu.

Lois de physique appliquées

Pour pouvoir créer un système physique réaliste, nous avons choisi d'utiliser les formules de Newton qui permettent de facilement simuler le système solaire sans implémenter de formule trop complexe.

Lois Newtonienne

Les lois de mouvement de Newton sont un ensemble de trois lois fondamentales qui décrivent la relation entre un corps et les forces agissant sur lui, formant la base de la mécanique classique. Nous pourrions utiliser la théorie de la relativité d'Einstein mais elle est, quant à elle, est une théorie de la gravitation qui décrit comment des objets massifs déforment le tissu de l'espace et du temps, conduisant au phénomène de gravité. Les principales différences entre les deux théories sont que les lois de Newton sont basées sur l'espace et le temps absolus, tandis que la théorie d'Einstein est basée sur l'espace et le temps relatifs, et que celle-ci est nécessaire pour décrire avec précision le comportement des objets se déplaçant à des vitesses très élevées ou dans des forts champs gravitationnels. N'ayant pas de besoin similaire, nous utiliserons donc les lois de Newton pour ce travail.

Loi de gravitation Newtonienne

La loi de gravitation de Newton énonce que tous les corps célestes exercent une force gravitationnelle sur les autres corps célestes. Cette force est proportionnelle à la masse de chaque corps et dépend de la distance qui les sépare. Plus les corps sont proches l'un de l'autre, plus la force gravitationnelle exercée est importante.

Voici l'équation mathématique de la loi de gravitation de Newton :

$$F = G * (m1 * m2) / r^2$$

où :

- F est la force gravitationnelle exercée entre les deux corps célestes
- G est la constante gravitationnelle (6.67 x 10^-11 N.m^2/kg^2)
- m1 et m2 sont les masses des deux corps célestes
- r est la distance entre les deux corps célestes

Cette équation peut être utilisée pour calculer la force gravitationnelle exercée entre deux corps célestes de masses connues et à une distance connue. Elle peut également être utilisée pour déterminer la distance entre les deux corps célestes en connaissant leur force gravitationnelle et leur masse respective.

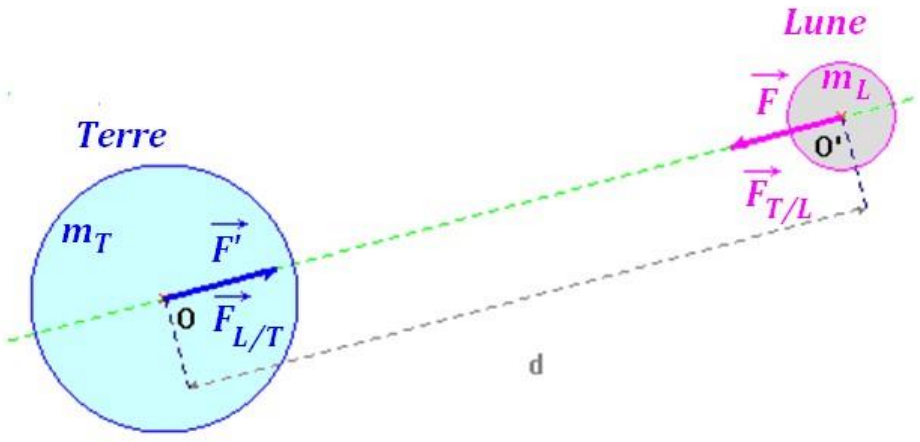

Figure 1: Principe de la loi de gravitation Newtonienne

Vitesse orbitale circulaire (solution facile)

L'équation permettant de calculer la vitesse orbitale circulaire d'un corps céleste dépend de la distance de ce corps céleste au corps central autour duquel il orbite, de la masse de ce corps central et de la constante gravitationnelle. La vitesse orbitale circulaire peut être exprimée en mètres par seconde ou en kilomètres par seconde.

Pour calculer la vitesse orbitale d'un astre, nous pouvons utiliser l'équation,

$$v = \sqrt{GM/r}$$

où :

- v est la vitesse
- G est la constante gravitationnelle
- M est la masse de l'objet autour duquel la Terre tourne (dans ce cas , le Soleil)
- r est la distance entre les deux objets (la distance de la Terre au Soleil).

En utilisant cette formule, nous pouvons calculer que la vitesse orbitale de la Terre autour du Soleil est d'environ 29,8 km/s. Cela signifie que la Terre se déplace à une vitesse d'environ 29,8 kilomètres par seconde en orbite autour du Soleil.

Ce calcul est basé sur l'hypothèse que l'orbite terrestre est un cercle parfait, ce qui n'est pas le cas. En réalité, l'orbite de la Terre est légèrement elliptique, ce qui signifie que la distance de la Terre au Soleil et sa vitesse orbitale ne sont pas constantes mais varient légèrement au cours de son orbite. Cependant, dans la plupart des cas pratiques, le calcul ci-dessus fournit une bonne approximation de la vitesse orbitale de la Terre.

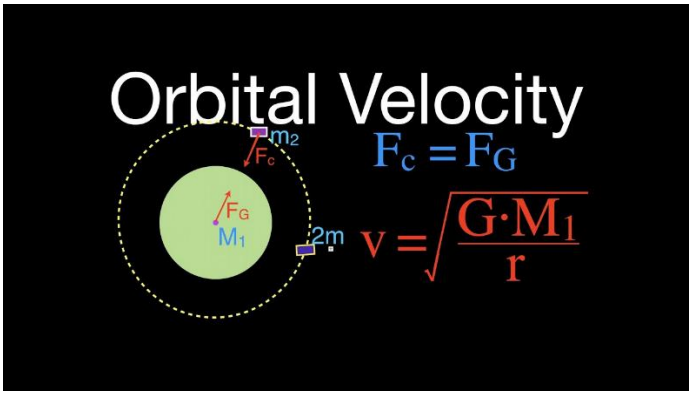

Figure 2: Principe de l'équation qui permet de calculer la vitesse orbitale.

Vitesse orbitale elliptique (solution moins facile)

La vitesse orbitale elliptique d'un corps céleste désigne la vitesse à laquelle ce corps se déplace autour d'un autre corps céleste, en suivant une orbite elliptique. Cette vitesse dépend de la distance entre les deux corps, de leur masse respective et de la force gravitationnelle exercée entre eux. Elle permet notamment de résoudre les approximations données par l'équation de la vitesse orbitale circulaire initialement prise en compte.

L'équation générale :

$$V = SQR(GM *((2/R)-(1/a)))$$

Où :

- v est la vitesse orbitale elliptique
- G est la constante gravitationnelle (6.67 x 10^-11 N.m^2/kg^2)
- M est la masse du corps central autour duquel l'autre corps céleste orbite
- r est la distance entre les deux corps célestes
- a est le demi-grand axe de l'ellipse décrivant l'orbite du corps céleste autour du corps central

Cette équation peut être utilisée pour calculer la vitesse orbitale elliptique à n'importe quel point de l'orbite, en utilisant la distance r entre les deux corps célestes en ce moment précis. Il est important de noter que cette équation ne prend pas en compte les effets relativistes ou d'autres phénomènes qui peuvent affecter la vitesse orbitale elliptique dans des situations extrêmes (par exemple, autour de trous noirs ou de corps célestes très massifs).

Vitesse angulaire astre

L'équation permettant de calculer la vitesse de rotation angulaire d'un corps céleste dépend de la période de rotation du corps céleste et de la distance à son axe de rotation. La vitesse de rotation angulaire peut être exprimée en degrés par seconde ou en radians par seconde.

Voici une équation générale pour calculer la vitesse de rotation angulaire en degrés par seconde :

$$\omega = 360 / P$$

où :

- ω est la vitesse de rotation angulaire en degrés par seconde
- P est la période de rotation du corps céleste en secondes

Voici une équation générale pour calculer la vitesse de rotation angulaire en radians par seconde :

$$\omega = 2\pi / P$$

où :

- ω est la vitesse de rotation angulaire en radians par seconde
- P est la période de rotation du corps céleste en secondes

Comme pour l’équation de la vitesse orbitale elliptique, celle-ci ne tient pas compte des effets relativistes ou d'autres phénomènes qui peuvent affecter la vitesse de rotation angulaire dans des situations extrêmes. Elles s'appliquent uniquement à des corps célestes tels que les planètes et les lunes dans le système solaire.

Caractéristiques des astres du système Solaire

Afin de garantir que la simulation une simulation réaliste, nous avons compilé les caractéristiques de chaque astre du système Solaire. Il y a beaucoup plus de caractéristiques (inclinaison de l'astre, point d'apogée de l'orbite élliptique, …) mais les informations ci-dessous pourront déjà donner une simulation qui tient la route.

	Mass in Earth Mass	Diameter in km	Distance to Sun in mKm	Rotation Period (rounded) in Earth Ref	Orbital Period (rounded) in Earth Ref	Statut
Sun	333000	297.85 (divided by 4 for visualization)	0	25 days	88 days	AJOUTÉ
Mercure	0.055	4 879,4	57,90	58 days and 15 hours	176 days	AJOUTÉ
Vénus	0,815	12 104	108,20	116 days and 18 hours	225 days	AJOUTÉ
Earth	1	12 742	151,48	1 day	365 days	AJOUTÉ
Moon	0.123	3474.8	149,60	28 days	28 days (around Earth)	AJOUTÉ
Mars	0,107	6 779	227,90	1 day	687 days	AJOUTÉ
Jupiter	317,8	139 820	778,34	9 hours	4335 days	AJOUTÉ
Saturne	95,152	116 460	1426,70	10 hours	10758 days	AJOUTÉ
Uranus	14,536	50 724	2870,70	17 hours	30708 days	AJOUTÉ
Neptune	17,147	49 244	4498,40	16 hours	60225 days	AJOUTÉ

Intégration du mode de jeu

La ludification de ce simulateur de jeu fut assez simple à imaginer. Nous nous sommes assez rapidement dirigés vers le contrôle d'un vaisseau à travers le système Solaire et ayant pour objectif de visiter les différents astres. Le joueur expérimente donc les quelques contraintes d'un voyage dans l'espace, notamment l'accélération et la décélération du vaisseau. Ce mode de jeu permet également de se rendre compte des distances et des tailles des différents astres au sein du système Solaire.

Physique du vaisseau

Les lois de Newton s'appliquent à un vaisseau spatial de la même manière qu'à n'importe quel autre objet. La gravité, par exemple, exercera une force sur le vaisseau spatial en fonction de sa masse et de celle des objets autour de lui. Les lois de la mécanique cinétique s'appliqueraient également, ce qui signifie que le vaisseau spatial aurait une vitesse et une direction constantes à moins qu'une force ne soit exercée sur lui. Si le vaisseau spatial entrait en collision avec un autre objet, les lois de la dynamique s'appliqueraient pour déterminer comment les deux objets réagiraient.

Étapes

1. Appliquer une force de gravité sur le vaisseau spatial en utilisant la première loi de Newton. Vous pouvez utiliser la masse du vaisseau spatial et celle des objets autour de lui pour déterminer la force exercée sur lui. Cela permettra au vaisseau spatial de suivre une trajectoire en orbite autour des objets qui l'entourent.
2. Utiliser la deuxième loi de Newton pour modéliser l'accélération du vaisseau spatial en réponse aux forces exercées sur lui. Par exemple, si le vaisseau spatial utilise ses propulseurs pour accélérer, on peut utiliser la force des propulseurs et la masse du vaisseau spatial pour calculer son accélération.
3. Utiliser la troisième loi pour prendre en compte la réaction en chaîne. Chaque force appliquée au vaisseau spatial doit avoir une force égale et opposée exercée sur un autre objet. Par exemple, si le vaisseau spatial accélère vers l'avant en utilisant ses moteurs, cela signifie que les moteurs doivent également exercer une force égale et opposée sur le vaisseau spatial pour le faire avancer.

Implémentation

Le simulateur est divisé en 2 parties : un simulateur qui permet de visualiser les différents astres en temps réel et un mode de jeu qui permet le contrôle d'un vaisseau et d'effectuer des quêtes de visite d'astre.

Aperçu

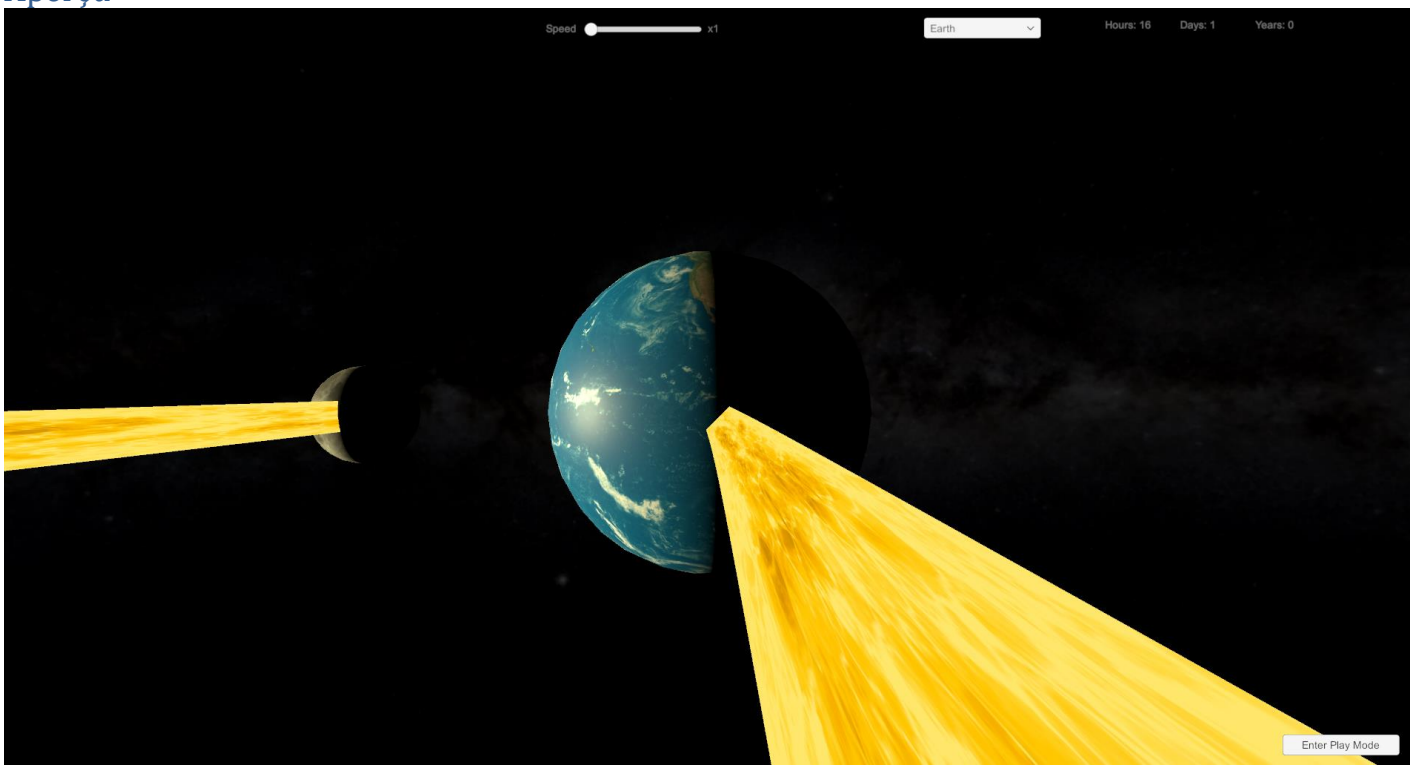

Figure 3: Aperçu du simulateur

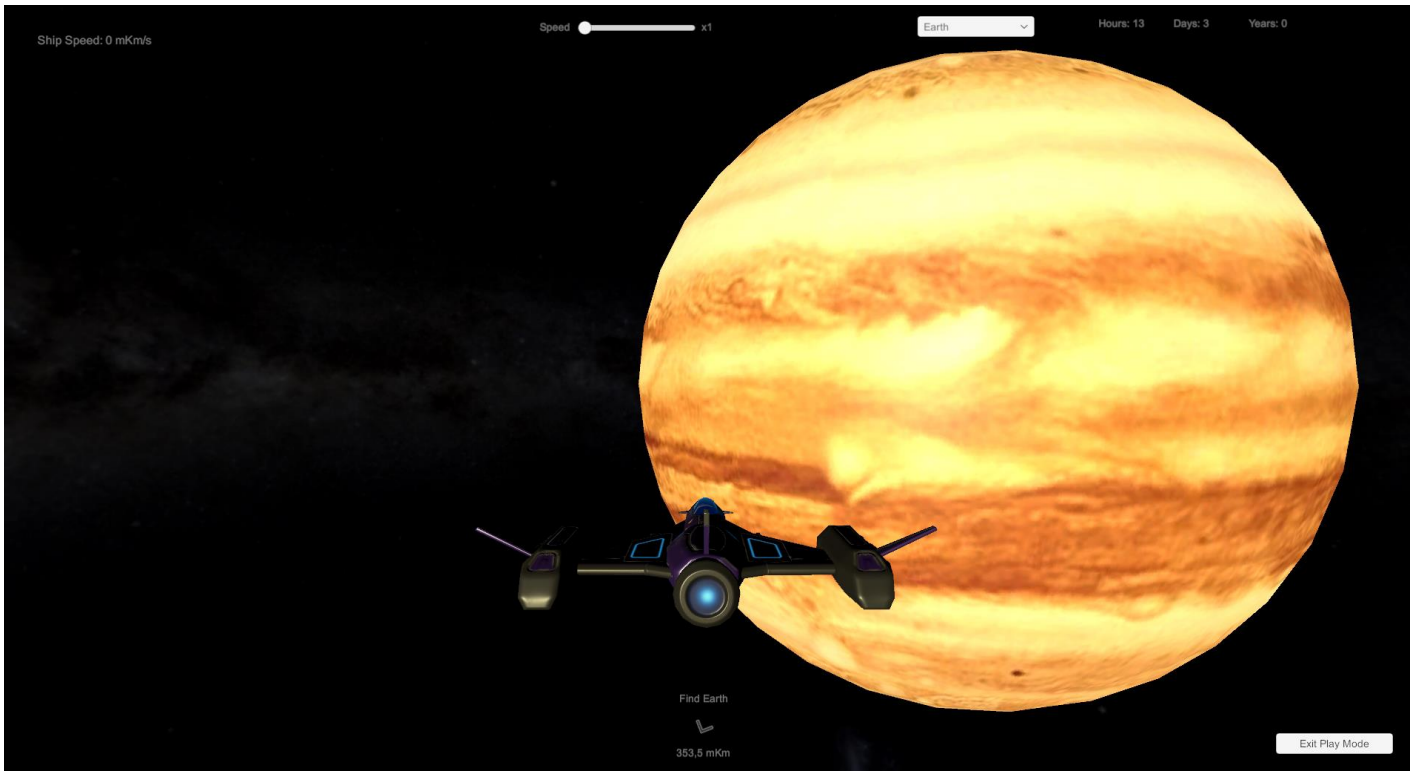

Figure 4 : Aperçu du simulateur avec le mode de jeu actif

Nous avons implémenté ces différents scripts :

- **SystemSolarManager** : Permet d'appliquer les forces de gravités, les rotations des astres, et les vitesses orbitales sur tous les objets ayant un Tag ***Bodies***. Cette solution ne nécessite pas de script individuel pour les différents astres, les seuls paramètres à appliquer manuellement aux astres sont leur masse et leur distance par rapport au Soleil.
- **GUIManager** : Permet d'interfacer les contrôles UI nécessaires à la simulation et au mode de jeu
- **CameraManager** : Permet de gérer la caméra pointée sur les astres à suivre ainsi que le vaisseau
- **SpaceShipMovement** : Permet de gérer les contrôles du vaisseau
- **ObjectivesManager** : Stocke et gère les objectifs du mode de jeu

Système Solaire

Pour implémenter les différentes lois identifiées au sein du système Solaire, nous avons,

1. Défini un objet **Rigidbody** pour chaque planète et pour le Soleil afin de gérer les forces et les collisions appliquées aux objets.
2. Calculé la force gravitationnelle exercée entre chaque planète et le Soleil en utilisant la loi de gravitation de Newton. La force gravitationnelle est donnée par l'équation F = G * (m1 * m2) / d^2, où G est la constante gravitationnelle, m1 et m2 sont les masses des deux objets, et d est la distance entre eux.
3. Appliqué la force gravitationnelle à chaque planète en utilisant la méthode AddForce de l'objet **Rigidbody** en s'assurant de prendre en compte la direction de la force en fonction de la position des objets dans l'espace.
4. Calculé la vitesse orbitale de chaque planète en utilisant l'équation de la vitesse orbitale v = sqrt(G * M / r), où v est la vitesse orbitale, G est la constante gravitationnelle, M est la masse du Soleil, et r est la distance de la planète au Soleil. Cette vitesse a été utilisée pour mettre à jour la position de la planète en utilisant l'objet **Transform**.
5. Ajouté la prise en compte du paramètre **Time.TimeScale**, permettant de modifier la vitesse d'exécution de la simulation
6. Ajouté un paramètre de **simulationFactor** pour prendre en compte les distances et masses avec une échelle adaptée à la visualisation

Code

```
void ApplyInitialOrbitSpeedToBodies()
{
    foreach (GameObject bodyA in bodies)
    {
        foreach (GameObject bodyB in bodies)
        {
            if (!bodyA.Equals(bodyB))
            {
                // On recupere la masse du corps B
                float mB = bodyB.GetComponent<Rigidbody>().mass;

                // On recupere la distance entre les 2 corps
                float d = Vector3.Distance(bodyA.transform.position, bodyB.transform.position);

                // On fait pointer le corps A en direction du corps B
                bodyA.transform.LookAt(bodyB.transform);

                if (!IsElepticalOrbit)
                {
                    /* On applique la vitesse orbitale initiale au corps A
                     * selon la formule,
                     *
                     *      v0 = Sqrt(G * m2)
                     *                -----
                     *                 d
                     */
                    bodyA.GetComponent<Rigidbody>().velocity += bodyA.transform.right * Mathf.Sqrt((G * mB * simulationFactor) / d);
                }
                else
                {
                    /* On applique la vitesse orbitale eliptique initiale au corps A
                     * selon la formule,
                     *
                     *      v0 = Sqrt(G * m2 * ( 2 -  1 ))
                     *                         --   --
                     *                         d    a
                     */
                    bodyA.GetComponent<Rigidbody>().velocity += bodyA.transform.right * Mathf.Sqrt((G * mB * simulationFactor) * ((2 / d) - (1 / (d * 1.5f))));
                }
            }
        }
    }
}
```

Figure 5: Fonction permettant d'appliquer la vitesse orbitale initiale.

```
void ApplyGravity(GameObject bodyA, GameObject bodyB)
{
    // On recupere les masses des 2 corps
    float mA = bodyA.GetComponent<Rigidbody>().mass;
    float mB = bodyB.GetComponent<Rigidbody>().mass;

    // On recupere la distance entre les 2 corps
    float d = Vector3.Distance(bodyA.transform.position, bodyB.transform.position);

    // On recupere le vecteur a utiliser pour appliquer la force (en direction de l'objet A)
    Vector3 dirVector = (bodyB.transform.position - bodyA.transform.position).normalized;

    /* On applique la loi universelle de la gravitation
     * en tant que force sur le corps A selon la formule,
     *
     *      F = G * m1*m2
     *              -----
     *               d^2
     */
    bodyA.GetComponent<Rigidbody>().AddForce(dirVector * (G * ((mA * mB) * simulationFactor) / (d * d)));
}
```

Figure 6: Fonction permettant d'appliquer les forces de gravitation sur les astres.

```
void ApplyAngularVelocity(GameObject body)
{
    /* On applique la rotation sur
    * le corps (vitesse angulaire) selon la formule,
    *
    *     W = 2 * PI
    *         ------
    *         P / 3600
    */
    body.transform.Rotate(Vector3.up * ((Mathf.PI * 2) / GetRotationPeriodInHours(body.name)) * Time.fixedDeltaTime * 50);

    if(body.name == "Earth")
    {
        IncrementEarthTime(Mathf.Abs(body.transform.eulerAngles.y));
    }
}
```

Figure 7: Fonction permettant d'appliquer les vitesses de rotation angulaire aux astres.

Vaisseau

- Pour implémenter la première loi de Newton, on doit tenir compte de l'inertie du vaisseau spatial. Cela signifie que plus le vaisseau spatial est lourd ou a une grande masse, plus il sera difficile à accélérer ou à déplacer. On peut gérer cela à travers les propriétés du **Rigidbody** attaché au vaisseau spatial, comme la masse (mass) ou la résistance à l'air (drag).
- La seconde loi de Newton stipule que la force totale appliquée à un objet est égale à sa masse multipliée par son accélération. On devrait donc tenir compte de la gravitation, des réactions des moteurs, des frottements et des forces de collision. On peut gérer cela à travers la méthode 'AddForce' de l'objet **Rigidbody**
- La troisième loi de Newton stipule que pour chaque action, il y a une réaction égale et opposée. En d'autres termes, nous devons nous assurer d'appliquer une force égale et opposée au vaisseau et ceci via le **Rigidbody** en utilisant également la méthode 'AddForce'

Code

```
// Update is called once per frame
void Update()
{
    if (GUIManager.GetPlayModeON())
    {
        // On applique une poussée positive ou négative au vaisseau ( 3ème loi de Newton)
        thrust=Input.GetAxis("Vertical") * Time.deltaTime;
        if (Input.GetKey(KeyCode.LeftShift))
        {
            thrust = thrust * 2;
        }
        speed += thrust; // La poussé modifie la vitesse ( 2ème loi de Newton)
        transform.Translate(0, 0, speed * 0.02f);
        /*Le vide spatial n'entraîne pas de frottement donc s' il n'y a pas de poussée
         le système est en inertie donc la vitesse ne change pas (1ère loi de Newton) */

        // systèmes de particules
        if(thrust>0)
        {
            thrust1.Play();
            if(Input.GetKey(KeyCode.LeftShift))
            {
                Boostthrust.Play();
            }
        }
        else if (thrust<0)
        {
            thrust2.Play();
            thrust3.Play();
            if (Input.GetKey(KeyCode.LeftShift))
            {
                BoostthrustL.Play();
                BoostthrustR.Play();
            }

        }

        // Si bouton droit pressé
        if (Input.GetMouseButton(1))
        {
            // On récupère les rotations X, Y de la souris
            rotationX += Input.GetAxis("Mouse Y") * (invertMouseY ? -1 : 1);
            rotationY += Input.GetAxis("Mouse X");

            rotationX = Mathf.Clamp(rotationX, -90, 90);

            // On effectue une rotation d'Euler sur le vaisseau
            transform.rotation = Quaternion.Euler(rotationX, rotationY, 0);
        }
    }
}
```

Figure 8: Fonction permettant d'appliquer les lois de Newton sur le vaisseau.

Problèmes rencontrés

- L'échelle du système pour que les orbites et accélérations restent cohérentes tout en apportant un visuel suffisant
    - Plus spécifiquement, la gestion du time scale pour accélérer/ralentir la simulation
    - Les forces de gravitation qui devenaient incohérentes après que les distances entre les astres aient été réduits (p.ex: la lune sortait de son orbite et subissant les forces de gravités du soleil de manière trop importante. Pour palier à ce problème, il a été nécessaire d'augmenter la masse de la Terre et adapter la constante universelle G)
- L'utilisation des masses et distances réelles des astres composants le système solaire a été compliqué à intégrer car chaque valeur est utilisé par les lois de Newton et doivent toutes respecter l'échelle appliquée

- L’architecture des composants qui permettent d’appliquer la gravitation ainsi que les orbites
    - La classe ***SolarSystemManager*** permet d'appliquer les différentes formules, notions temporelles et la définition des constantes mais elle a ses limites en termes de paramétrages. Il serait plus judicieux de gérer ces aspects avec plus de classes: Gestionnaire de gravité, Astre, Gestionnaire du système solaire.
- L’application d’orbites elliptique, plus complexe à première vue, plutôt que des orbites circulaires qui étaient initialement prévues
    - Il était intéressant de commencer par appliquer une orbite circulaire afin de valider une première itération de l’implémentation mais l’orbite elliptique étant plus réaliste, nous avons migré vers cette implémentation

Conclusion

Nous avons réalisé un simulateur d’un système Solaire en appliquant les différentes lois physiques, notamment les lois newtoniennes qui nous ont permis d’être au plus proche du comportement des différents astres et de leurs interactions entre eux.
Durant ce travail, il a été intéressant d’observer l’équilibre complexe et parfait donné par ce système Solaire qui nous permet aujourd’hui de vous écrire ces quelques lignes. Cette complexité fut ressentie durant l’implémentation où chaque paramètre a joué un rôle primordial pour le bon fonctionnement de ce système.
Cette première version perfectible permet de se donner une idée du résultat que pourrait donner la version finale moyennant une refonte de l’architecture pour avoir un simulateur plus modulaire et évolutif ainsi qu’une amélioration du gameplay où des fonctionnalités pourraient être ajoutées afin d’apporter une immersion plus importante du joueur. On pourrait également imaginer que les composants développés deviennent des API intégrables au sein d’un jeu vidéo plus vaste comme par exemple Star Citizen.
Pour finir, ce sujet fut intéressant et passionnant à mener car il a permis de rapidement visualiser notre système Solaire et ceci à l’aide des lois physiques qui régissent notre univers et sont finalement un élément de la raison de notre existence.