RAPPORT DE POROJET INF443

Simulation éducatrice du Système Solaire en modèle simplifié

7 juin 2023

Daniel Abraham Elmaleh

TABLE DES MATIÈRES

1 Introduction **3**

2 Interface d'utilisation **3**

3 Approche et modélisation **4**

4 Réalisation **5**
4.1 Défis, solutions et Difficultés rencontrées 5

5 Perspectives **7**

6 Conclusion **7**

7 Annexe **8**

Figure 1 – Image de la simulation

1 INTRODUCTION

Le projet décrit dans ce rapport a été réalisé dans le cadre du cours d'informatique graphique. Il s'agit d'un programme informatique codé en C++, et mis en œuvre en utilisant la bibliothèque OpenGL. La bibliothèque d'interface CGP, fournie par notre professeur, a servi de support principal.

L'objectif du projet est de représenter graphiquement une simulation de notre système solaire à des fins éducatives et interactives. Cela a été réalisé en modélisant de manière simplifiée notre système solaire, incluant le mouvement des planètes dans l'espace autour du soleil. Il offre également une interaction avec l'utilisateur afin d'obtenir des informations caractéristiques des planètes et des données concernant leur mouvement simulé.

De plus, les rapports de grandeur ont été conservés pour donner une intuition sur les dimensions physiques. Cela permet d'introduire les grandeurs physiques de manière graphique, interactive et éducative. Il convient de noter que le modèle est simplifié et ne représente qu'une introduction très basique à la physique du système solaire. Par conséquent, il ne peut pas être considéré comme parfaitement fidèle à la réalité.

Cependant, ce projet est suffisamment générique pour être amélioré et pour représenter le système solaire de manière plus précise et fidèle à la réalité.

2 INTERFACE D'UTILISATION

L'utilisateur peut se déplacer dans le système solaire en utilisant la souris de la manière suivante : un clic gauche permet d'effectuer une rotation autour de l'axe (0,0,1). La rotation de l'axe par défaut (0,0,1) s'effectue avec les flèches, et la combinaison Ctrl + clic gauche permet d'effectuer une translation de la caméra dans l'espace pour observer autre chose que le point central.

Dans l'interface utilisateur, nous retrouvons différentes options : scale, paths, trajectories, earth night, look at "celestial body", no look. De plus, des informations sur chaque planète sont disponibles.

Scale : Correspond à une modification non uniforme de l'échelle de tous les objets sauf la Lune. L'objectif est de permettre une meilleure visualisation des éléments de la scène. Il est à noter que lorsque l'échelle est modifiée, les rapports de tailles et de distances ne sont plus en adéquation avec la réalité.

Paths : Permet l'affichage des trajectoires fixes et circulaires des planètes du système solaire.

Trajectories : Affiche la trajectoire des planètes en temps réel, illustrée par un tableau de 1000 points. Cette option est plus pertinente si le modèle devient plus élaboré.

Earth night : Affiche l'image de la Terre lorsqu'il fait nuit.

Look at 'celestialBody' : Permet de centraliser et de bloquer la caméra sur l'objet céleste spécifié et d'afficher les informations pertinantes sur l'objet sélectionné.

No look : Annule cette fonctionnalité. Et permet de se balader librement dans l'espace.

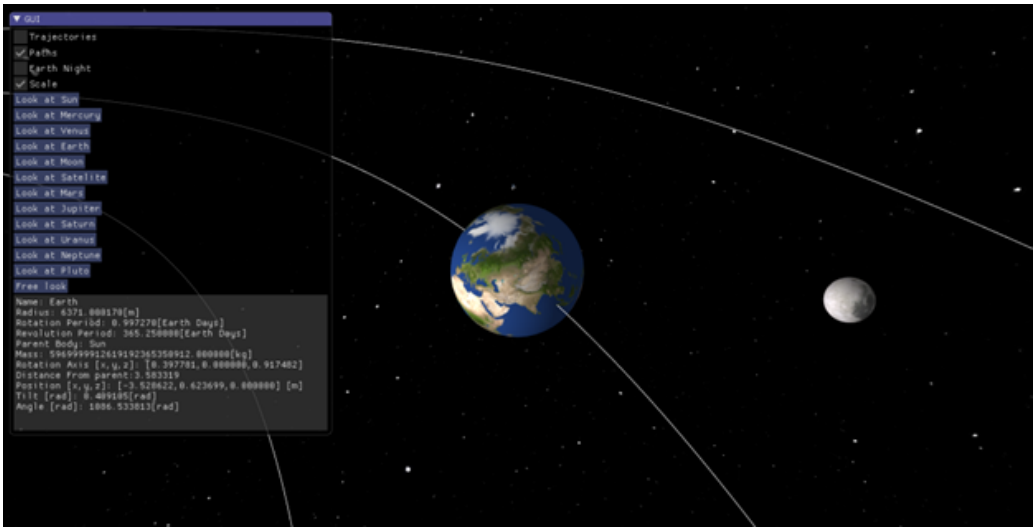

Figure 2 – Illustration de l'interface d'utilisation

3
APPROCHE ET MODÉLISATION

L'approche générale pour réaliser ce projet a consisté à m'appuyer sur du code déjà existant en tant qu'exemple, puis à procéder à des modifications graduelles par l'ajout de morceaux de code. L'élément novateur a été de créer une classe pour les objets célestes afin de faciliter la programmation du projet et de le rendre modulable et polyvalent. La création de tous les objets repose sur leur forme commune, la sphère. Le satellite, étant le seul objet non sphérique, a donc été créé séparément. Les règles de mouvement de manière hiérarchique ont également contribué à ce choix.

Le modèle du système solaire que j'ai choisi est une version très simplifiée dans laquelle nous approximons le mouvement des planètes par des cercles dont la période est équivalente à leur période réelle de révolution. Cette approximation est mise en œuvre à l'aide de l'équation trigonométrique du cercle centré sur l'origine (le Soleil).

$$x = dist * cos(\omega * t)$$
$$y = dist * sin(\omega * t)$$
$$Pos = (x, y, 0)$$

Les planètes sont inclinées selon leur réel degré d'inclinaison. Elles effectuent une rotation autour de leur axe N-S, avec une représentation relative équivalente à leur vitesse réelle de rotation. La position des planètes par rapport aux autres n'est pas réaliste, elles ont toutes été positionnées sur le même axe. Selon Kepler, les planètes ont une trajectoire plane, ce qui a été correctement implémenté, toutes les planètes se situent sur le même plan (z = 0). Elles commencent leur mouvement alignées le long de l'axe x de la simulation. Les distances des planètes sont normalisées relativement à la distance de Mercure au Soleil, ce dernier étant défini comme unité. Les rayons des planètes sont normalisés par rapport au rayon du Soleil.

Le modèle animé comprend les objets suivants : le Soleil, les planètes du système solaire incluant Pluto, notre Lune, et un satellite constitué de 5 parties, à savoir le corps du satellite (rectangle primitif), une tour (cylindre primitif) avec une antenne rotative (disque primitif) à son extrémité, deux ailes (rectangle primitif par interpolation de 2 points), et une tête (cône primitif). Tous les objets célestes, à l'exception du satellite, sont

modélisés sous la forme d'une sphère implémentée par une sphère primitive.

Dans le fichier constantes.hpp se trouvent toutes les données caractéristiques réelles des objets célestes de notre système solaire. Ces données ont été utilisées pour définir les mouvements, tailles et distances (avec des rapports de proportionnalité approximatifs).

4 RÉALISATION

Le choix a été fait d'exploiter principalement deux caractéristiques du C++ : la programmation orientée objet et la notion de pointeur avec utilisation d'allocation dynamique de mémoire. L'approche générale a consisté à créer une classe très générale représentant un objet céleste, afin de pouvoir ensuite créer la base de n'importe quel objet dans la simulation. Ceci est surtout possible grâce au fait que leurs attributs sont souvent très génériques, à savoir la vitesse et la période de rotation et de révolution, etc.

Dans ce projet, j'ai utilisé deux structures hiérarchiques différentes mais redondantes. La première en créant une classe pour caractériser les attributs et fonctions génériques d'un objet céleste, et la seconde en utilisant la structure hiérarchique proposée par la bibliothèque CGP pour tirer profit de l'animation par hiérarchie et faire appel à la fonction de conversion de coordonnées locales en globales. Ce choix m'a permis de faciliter la tâche d'animation ainsi que la structuration du code et la pertinence des informations.

Chaque objet céleste est dérivé d'un autre objet, à l'exception du Soleil. L'implémentation a contourné la création de classes dérivées en utilisant un attribut sous forme de pointeur qui désigne l'objet parent. De cette manière, j'ai pu conserver une certaine simplicité.

4.1 Défis, solutions et Difficultés rencontrées

La plupart des difficultés rencontrées ont été résolues grâce à une lecture détaillée de la bibliothèque cgp, qui m'a permis de comprendre comment utiliser certaines fonctions et structures. Le satellite a été conçu selon une hiérarchie d'éléments, une solution clé pour exploiter la généricité de mon code. Afin de faire du satellite un objet céleste, j'ai choisi de positionner une lune minuscule à la tête de sa hiérarchie. Cette méthode a permis de forcer le mouvement et les caractéristiques d'un objet céleste sans toutefois visualiser celui-ci.

Une amélioration que je n'ai pas réussi à implémenter était d'ajouter les trajectoires de la lune et du satellite. Avec une reprise des éléments de la fonction "local_to_global_coord", cela aurait pu être possible, mais ce n'était pas évident, car les éléments "path" et "trajectory" ne sont pas des "mesh drawable".

L'utilisation de pointeurs et d'allocation dynamique s'est avérée être une très bonne solution pour ce projet. En effet, la taille mémoire que tous les objets célestes occupaient dans la mémoire statique était trop importante et a entraîné des fuites de mémoire. La transition vers les pointeurs a été complexe car il a fallu convertir toutes les parties du code en syntaxe compatible avec les pointeurs. Grâce à la flexibilité du C++, cette transition a été relativement facile. Une difficulté rencontrée était la portée (locale ou globale) de ces pointeurs vers les objets célestes. C'est pourquoi nous avons eu recours à l'allocation dynamique, qui a résolu tous les problèmes rencontrés. Le seul ajout était de veiller à libérer cette allocation en fin de programme avec des "delete".

Illumination - Pour l'illumination, j'ai opté pour une modification de la position de la lumière pour qu'elle soit fixée au centre de la scène, plus précisément au centre du soleil, plutôt qu'à une position initiale dépendant

de la caméra. Ce choix permet d'illuminer la scène d'une manière qui semble réaliste, car les planètes sont éclairées depuis la direction du soleil. Cela permet de visualiser le jour et la nuit sur les planètes.

Animation - L'animation, ou la simulation animée de la scène, est une mise en œuvre du modèle simple proposé par ce projet. La simulation est réalisée grâce à des appels récurrents et fréquents dans la boucle d'animation, et par l'appel d'une fonction calculant la nouvelle position spatiale et la rotation angulaire de chaque objet céleste. La simulation est donc non interactive et prédictive, car elle implémente un modèle avec des mouvements prédéfinis. L'interaction avec le programme est dédiée à la visualisation et à la navigation dans la scène, afin d'acquérir des informations qualitatives et quantitatives grâce aux données affichées lors de la sélection d'une planète.

Textures - Les textures permettent de visualiser l'image d'une planète ou d'un objet de la scène. Les textures des planètes ont été trouvées sur un site internet (au lien : https ://www.solarsystemscope.com).

Shading/matériaux - Pour compenser le manque d'éclairage provenant de la caméra, et pour que les planètes soient visibles même dans l'obscurité, j'ai modifié leurs paramètres de Phong de la manière suivante : augmentation du paramètre ambiant à 0.5 (ce qui augmente la lumière venant de toutes les directions) afin de visualiser l'objet même dans l'obscurité. C'est une imprécision qui est là pour faciliter la visualisation. J'ai aussi augmenté le paramètre de diffusion à 0.8 pour atténuer l'effet de réflectivité de la sphère (la planète), une planète n'est pas brillante comme une bille de verre. De même, le paramètre spéculaire a été réglé à 0 pour supprimer l'effet de réflexion ponctuelle de la lumière.

Figure 3 - Grâce à l'utilisation des pointeurs, il a été plus facile de mémoriser chaque objet céleste dans un tableau de pointeurs. Cela a permis de retrouver l'objet concerné et sa position (qui est un attribut de l'objet) afin de centrer la caméra sur sa position en temps réel.

Figure 4 - Voici un exemple de création d'un objet céleste (ici, Jupiter), suivi d'ajustements spécifiques à l'aide de "setters" pour caractériser certains attributs non génériques tels que l'inclinaison, l'axe de rotation et la masse, qui sont utilisés pour l'affichage des caractéristiques de la planète.

Figure 5 - La mise en rotation de la platine du satellite, ainsi que l'inclinaison de cette dernière, ont été gérées. La gestion conditionnelle de l'entrée de l'utilisateur via l'interface graphique permet d'afficher et de dimensionner correctement le satellite.

Figure 6 - La partie centrale de la fonction display_frame parcourt le tableau de pointeurs vers les objets de l'animation. Elle appelle la fonction update_data qui met à jour les attributs de chaque objet selon les formules et caractéristiques correspondantes en fonction du temps. C'est le cœur de l'animation du projet. De plus, les affichages conditionnels en fonction des entrées de l'utilisateur sont pris en compte pour mettre à jour l'animation.

Figure 7 - La fonction principale qui met à jour l'objet céleste est une méthode de la classe CelestialBody. Cette fonction implémente le modèle simple des trajectoires circulaires choisi pour ce projet, ainsi que la rotation de l'objet lui-même. Il y a également une mise à jour du tableau des points de la trajectoire en fonction de la nouvelle position de l'objet.

Figure 8 - Les attributs de la classe CelestialBody.

5 PERSPECTIVES

Ce projet constitue une excellente introduction à la compréhension des dimensions et du mouvement élémentaire des planètes de notre système solaire. Néanmoins, il est tout à fait envisageable de le perfectionner pour mieux modéliser notre système solaire. La structure générique du code, l'approche orientée objet et le choix d'encapsulation et de modularité du code, tels qu'ils sont mis en œuvre dans ce projet, facilitent une adaptation relativement aisée à des modifications et des complexifications du modèle et de l'animation.

Par exemple, lors du calcul des nouvelles positions des planètes, il serait possible de générer des mouvements plus réalistes en effectuant des calculs d'interactions doubles ou multiples et en implémentant le modèle de Kepler.

Parmi d'autres améliorations possibles, on pourrait envisager de modéliser le terrain de chaque planète pour que ces dernières ne soient plus de simples sphères. On pourrait également ajouter les lunes des planètes et la ceinture d'astéroïdes afin d'obtenir une simulation plus réaliste du système. Il serait aussi possible d'améliorer les positions relatives des planètes dans l'espace en utilisant les informations disponibles sur le site de la NASA par exemple. De plus, les axes de rotation par rapport à la position du soleil n'ont pas été pris en compte, tout comme les axes de révolution de la lune et des différentes planètes par rapport à l'autre.

Les possibilités d'amélioration sont infinies, mais compte tenu de l'objectif éducatif très basique et dans le cadre d'un mini-projet de cours académique, j'ai jugé l'implémentation actuelle suffisante.

6 CONCLUSION

Au cours de ce mini-projet de programmation graphique, j'ai acquis des compétences et des notions clés sur l'intuition sous-jacente à l'informatique graphique et à la bibliothèque OpenGL en particulier.

Ce projet m'a permis d'aborder les concepts théoriques du cours sous un angle très différent, tout en soulignant la particularité de la programmation orientée objet pour modéliser des scènes graphiques. L'objectif était de trouver une compatibilité sémantique avec les mathématiques et la physique qui se cachent derrière, en plus de l'intuition humaine. Ce cours servira de base pour toute future interaction avec ce vaste sujet.

7 ANNEXE

Figure 3 – Caption for the first figure

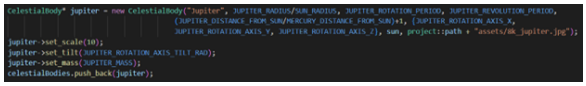

Figure 4 – Caption for the second figure

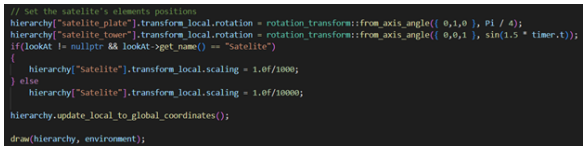

Figure 5 – Caption for the first figure

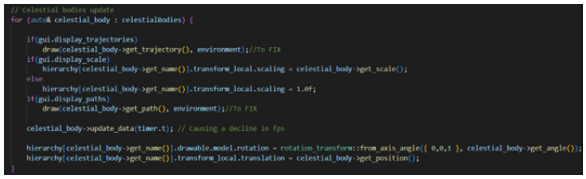

Figure 6 – Caption for the second figure

```cpp
void CelestialBody::update_data(float time)
{
    // Update the angle of rotation of the celestial body
    this->angle = rotation_angular_velocity * time;

    // Calculate the position of the celestial body based on its orbit
    if (distance_from_parent > 0.0f) {
        float orbit_angle = revolution_angular_velocity * time;
        float x = distance_from_parent * cos(orbit_angle);
        float y = distance_from_parent * sin(orbit_angle);
        this->position = { x, y, 0 };
    } else {
        this->position = { 0, 0, 0 };
    }

    // Update the trajectory of the celestial body
    if (this->get_path_traj())
        trajectory.add(this->position);
}
```

Figure 7 – Fonction de mise a jour pour annimation

```cpp
private:
        //Set once in constructor
        string name;
        float radius;
        float rotationPeriod;
        float revolutionPeriod;
        vec3 rotation_axis;
        CelestialBody* parentBody;
        string texture_path;
        float revolution_angular_velocity;
        float rotation_angular_velocity;
        //Varies
        float distance_from_parent;     //Set in constructor
        vec3 position;                  //Set in constructor
        float scale;
        trajectory_drawable trajectory = trajectory_drawable(500);
        curve_drawable path;
        float tilt;
        float angle;
        float mass;
        bool path_traj;
```

Figure 8 – Attributs de CelestialBody