PROJET INF443

Simulateur Système Solaire

6 juin 2023

Thibaut de Saivre
Antoine de Tarlé

ÉCOLE POLYTECHNIQUE
INSTITUT POLYTECHNIQUE DE PARIS
TYPOGRAPHIX

TABLE DES MATIÈRES

1 PRÉSENTATION GÉNÉRALE

L'objectif suivi pour ce projet était de réaliser une simulation semi réaliste du système solaire, et de donner la possibilité de l'explorer en contrôlant un vaisseau spatial, avec un bouclier activable/désactivable pour repousser les astéroïdes sur son chemin. Le contrôle du vaisseau et le retard de l'orientation de la caméra par rapport à celle du vaisseau sont inspirés du jeu mobile Galaxy On Fire 2.

Le code source du projet est organisé de la manière suivante :

```
Projet
├── assets ..................................................... textures et bump maps
├── cgp ..................................................................librairie CGP
├── shaders ..................................................... shaders personnalisés
└── src
    ├── background ..........................................................galaxie en background
    ├── celestial_bodies ..........................code source des astres simulés et affichés
    ├── navion ..................................................................vaisseaux spatiaux
    ├── simulation_handler ................................gestionnaire de simulation physique
    └── utils
        ├── camera ..........................................caméra et son modèle personnalisés
        ├── controls ..................................................contrôle du vaisseau au clavier
        ├── display .....................classes de bases de la hiérarchie des objets drawable
        ├── noise ....................................modèles de sphères bruitées (astéroïdes)
        ├── opengl ......................fonctions draw avancées OpenGL (instancing, UBO...)
        ├── physics .....................classe de base des objets physiques + timer statique
        ├── random .....................fonctions de générations de nombres et vecteurs random
        ├── shaders .....................classe statique de chargement et stockage de shaders
        ├── threads ................................utilitaire pour la simulation multithreaded
        └── tools ..........................................fonctions utilitaires géométriques
```

Figure 1 – Structure du code

En résumé, on y trouve en terme de fonctionnalités :

— **Simulation pseudo réaliste** des principaux corps du système solaire (Soleil, Mars, Terre, Jupiter, Saturne, Uranus, Neptune).
— **Simulation de ceintures d'astéroïdes** (ceinture des planètes telluriques, anneau de Saturne, grande ceinture de Kepler) en temps réel.
— **Vaisseau spatial contrôlable par l'utilisateur** en third-person-view avec une caméra customisée.
— **Détection des collisions entre le joueur et les astéroïdes** par l'intérmédiaire d'un bouclier activable/désactivable, ainsi qu'entre les astéroïdes et leur astre attracteur principal.
— **Détection des collisions entre un rayon laser et les astéroïdes** permettant leur destruction à distance.
— **Caméra personnalisée** qui suit les déplacements du vaisseau avec un retard programmé, donnant une impression de fluidité dans le mouvement.

Les mécanismes OpenGL / C++ mis en place pour réaliser ces fonctionnalités sont les suivants :

— **POO et héritage multiple** pour organiser les objets dessinables et simulés.
— **Shaders personnalisés** pour le Soleil, le vaisseau spatial, les planètes, les astéroïdes et la galaxie affichée en fond.
— **Caméra personnalisée** à base de vecteurs (direction et verticale locale) plutôt que roll, pitch, yaw pour un contrôle plus fin et intuitif de l'orientation.
— Modèles d'astéroïdes générés avec un bruit de perlin 3D, ainsi qu'une texture et une position dans la ceinture randomisés.
— **Fonctions draw personnalisées** pour utiliser des fonctionnalités plus avancées d'OpenGL (**instancing** pour les astéroïdes, **Uniform Buffer Objects** pour l'animation du bouclier entrant en contact avec des astéroïdes, Shadow mapping).
— **Bump mapping** pour la surface des astéroïdes.
— **Multithreading** et semi synchronisation de worker threads avec le thread principal pour gérer un très grand nombre d'astéroïdes ($> 100\ 000$!) en simulation temps réel.
— **Contrôle du vaisseau au clavier** avec un mécanisme d'accélération de translation et rotation pour une sensation fluide. Utilisation d'une map pour stocker les états des touches à chaque frame, afin de pouvoir gérer un grand nombre de touches pressées à chaque frame.

Ces points sont présentés dans les deux parties suivantes.

2
HIÉRARCHIE DE LA SIMULATION ET DE L'AFFICHAGE

Présentation de la hiérarchie des classes du projet. L'usage de la POO a été légèrement excessive dans ce projet, notamment à cause du manque de recul initial sur les structures à employer pour mener à bien les simulations et l'affichage.

2.1 Objects

Les planètes et les astéroïdes suivent une simulation physique pseudo-réaliste en temps réel. Les paramètres des différentes planètes représentées sont réalistes. Les distances sont mesurées en mètres, les axes de rotations et vitesses de rotations sur soi sont celles des planètes réelles. En ce qui concerne les orbites, on suppose qu'elles sont parfaitement circulaires. Chaque planète a sa position initiale sur l'axe (O, x) au lancement de la simulation, et se voit imprimer initialement une impulsion correspondant à $v_0 = \sqrt{\frac{MG}{r}}$ avec M la masse de l'attracteur en kg, G la constante de gravitation universelle, et r le rayon de l'orbite (on néglige la masse de la planète ou de l'astéroïde devant celle de l'astre attracteur).

En ce qui concerne le code, chaque objet simulé hérite de la classe **Object** (définie dans */utils/physics*), qui définit une position physique, une vitesse, une masse, ainsi que différentes fonctions utilitaires (comme pour calculer la force d'attraction gravitationnelle liée à un autre objet).

A chaque frame, les classes de gestion de simulation (dans */simulation_handler*) appliquent aux objets de type planète le traitement suivant :

1. Remise à 0 des forces appliquées sur chaque objet.
2. Calcul des forces d'attraction gravitationnelles entre chaque paire d'objets (pour la classe *SimulationHandler*. La classe *OptimizedSimulationHandler* ne calcule que la force d'attraction gravitationnelle des objects précisés comme *attracteurs* du système. Elle était destinée au traitement des ceintures d'astéroïdes, mais a été remplacé au profit d'autres méthodes de simulation).
3. Calcul de l'accélération pour cette frame.
4. Mise à jour de la vitesse et de la position.
5. Éventuellement, des tests de collision.

Les astéroïdes subissent un traitement légèrement différent, géré par la classe *AsteroidBelt*(dans */celestial_bodies/asteroid_belt*). Dans la mesure où une ceinture d'astéroïdes peut graviter autour d'un objet en mouvement dans le référentiel du Soleil (Saturne...), la stabilité de son orbite n'est pas garantie si on suit un modèle simpliste d'orbite circulaire avec une vitesse initiale calculée avec la formule précédente. Ainsi, on applique aux astéroïdes le prétraitement suivant :

1. Calcul du déplacement de l'astre attracteur depuis la dernière frame.
2. Mise à jour de la position de chaque astéroïde en lui ajoutant ce déplacement.
3. Suite de la simulation comme pour les planètes

Ce traitement a pour effet de supprimer les effets non galiléens liés à une révolution autour d'un astre en orbite autour du Soleil.

Une autre particularité des ceintures d'astéroïdes (dans un but visuel) est que les astéroïdes d'une ceinture suivent une orbite purement circulaire, malgré l'offset "vertical" (la verticale désignant l'axe de révolution) qu'ils peuvent avoir par rapport au plan de révolution. Cet offset est pris en compte durant la simulation pour éviter que les ceintures ne se contractent progressivement selon l'axe "vertical" à cause de positions initialement excentrées des astéroïdes.

Adaptations pour l'affichage :

- **Mise à l'échelle des distances :** il n'est pas envisageable d'utiliser des distances réalistes dans les appels OpenGL (de l'ordre de $10^{10}m$ à $10^{12}m$). Avant l'affichage, on applique une mise à l'échelle de toutes les distances en les multipliant par un facteur 10^{-10}.
- **Mise à l'échelle des rayons :** En ce qui concerne la taille des objets, on multiplie les rayons des planètes par 1500, et le rayon du Soleil "seulement" par 150, de manière à pouvoir apercevoir à l'écran chaque objet de manière satisfaisante.
- **Mise à l'échelle des vitesses d'orbite :** Enfin, les vitesses d'orbite sont par défaut multipliées par un facteur 20 afin que les planètes n'apparaissent pas immobiles. Ce facteur peut varier en ce qui concerne les ceintures d'astéroïdes plus ou moins distantes. Ils ont été adaptés pour donner un rendu à l'apparence plus dynamique et uniforme. (En réalité, c'est le pas de simulation qui est multiplié).
- **Pas temporel de simulation :** Le rapport temporel de simulation choisi est de **1s = 1 jour**. Les vitesses de rotation réelles des planètes n'ont pas été modifiées, ainsi la Terre effectue un tour sur elle même par seconde dans la simulation.

2.2 Drawables et base drawables

On présente ici la structure hiérarchique des classes permettant l'affichage d'objets.

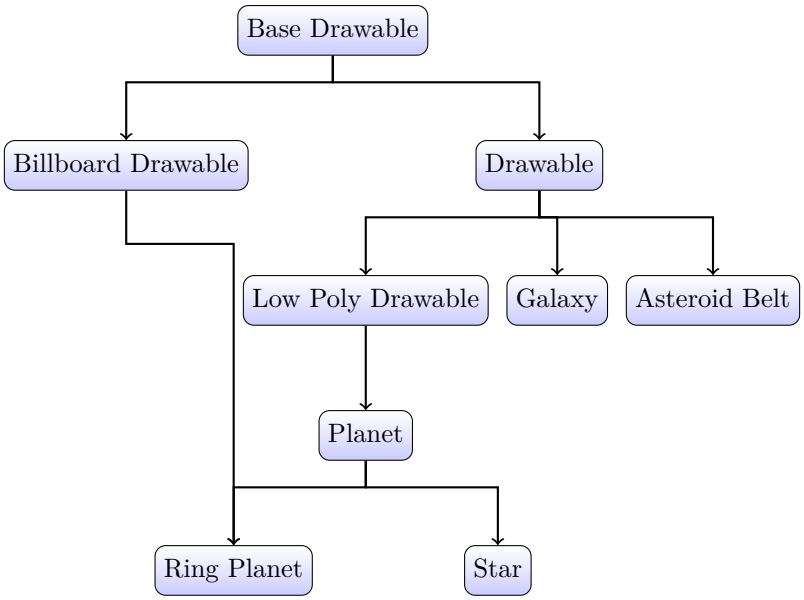

Figure 2 – Héritage des classes Drawable

Description des classes :

Base Drawable : Classe de base. Contient une fonction **initialize()** pour initialiser les *mesh_drawable*.

Drawable : Classe d'affichage des Drawables qui ne sont pas des billboards.
Contient une fonction **draw()** qui doit être appelée dans *scene.display_frame*.

Billboard Drawable : Classe d'affichage de billboards.
Contient une fonction **drawBillboards()** qui doit être appelée dans *display_semiTransparent*.

Low Poly Drawable : Classe d'affichage d'un drawable, dont la fonction **draw()** choisit en fonction de la distance de l'objet à la caméra et de son rayon si afficher l'objet réel avec **draw_real()**, ou un disque de même rayon et de couleur similaire avec **draw_low_poly()**.

Planet : *Drawable* initialisé avec un mesh sphérique, et qui hérite de *Object* pour la simulation physique.

Ring Planet : *Planet* avec un billboard transversal pour figurer une ceinture d'astéroïdes.
A été remplacé par l'usage d'un objet *Planet* avec ceinture d'astéroïdes (Saturne)

Star : Instance de *Planet* dont on peut modifier le shader. Utilisé pour le Soleil avec le shader **lava**.

Galaxy : *Drawable* servant à afficher une galaxie en fond.
La fonction **draw()** de *Galaxy* centre l'objet sur la caméra avant l'affichage, puis vide le *Depth Buffer* ensuite, de manière à éviter de masquer des objets lointains.
Elle doit toujours être draw en premier, à cause du vidage du *Depth Buffer*.

Asteroid Belt : Classe qui simule et affiche une ceinture d'astéroïdes autour d'un unique attracteur.
Les nombreux astéroïdes d'une ceinture sont affichés via de l'instancing, à partir d'un petit nombre de modèles de base.

Les héritages sont tous publics. Étant donné que certaines classes comme *Ring Planet* peuvent hériter de *Drawable* et *Billboard Drawable* en même temps, on fait hériter *Drawable* et *Billboard Drawable* **virtuellement** de *Base Drawable*, afin d'éviter les ambiguités de chemin d'héritage pour le compilateur.

Cas particuliers :
— L'affichage du vaisseau du joueur et de son bouclier sont gérés par la classe de contrôle au clavier.
— L'affichage des astéroïdes est géré directement par la ceinture d'astéroïdes, qui fait le lien entre les centaines de milliers d'astéroïdes et la douzaine de *mesh_drawables* utilisés comme base.

Afin d'afficher un grand nombre d'objets à l'écran sans saturer le GPU, tous les objets emploient donc des mesh différents selon la valeur du rapport $\frac{\text{distance à la caméra}}{\text{rayon de l'objet}}$. Cette conversion est gérée directement par la classe *Low Poly Drawable* dans le cas des planètes, qui en héritent directement. Dans le cas des astéroïdes dessinés par instancing, c'est la classe *Asteroid Belt* qui répartit les astéroïdes vers les mesh correspondants.

Les planètes disposent ainsi d'un mesh standard "HD" et d'un mesh disque de couleur égale à la couleur moyenne de la texture. Les astéroïdes possèdent 3 meshs différents :
— Un mesh HD
— Un mesh "low poly"
— Un mesh disque de couleur égale à la couleur moyenne de la texture.

On remarquera que dans le cas des astéroïdes, les paramètres ont été choisis de sorte qu'on voie visuellement la transition du mesh disque vers le mesh low poly en s'en rapprochant, pour mieux apprécier le fonctionnement de ce mécanisme. On peut ainsi afficher sans trop de soucis plus de 100 000 objets à l'écran ! L'étape suivante (mais qui n'a pas été réalisée) consisterait à choisir de ne pas afficher les objets se situant "top loin" de la caméra. En l'occurrence, l'affichage de tous les objets, lorsqu'on se situe au sein de la grande ceinture d'astéroïdes de Kepler (donc plus de meshes HD utilisés) nécessitait d'utiliser une carte graphique discrète pour conserver 60 fps (testé avec Nvidia RTX 3050 mobile vs AMD Radeon intégrée).

3
MODÈLES GÉOMÉTRIQUES, MÉTHODES 3D, ET TECHNOLOGIES EMPLOYÉES

3.1 Shaders et fonctions OpenGL personnalisées

Les shaders sont contenus dans le dossier *shaders/*. Chaque dossier contient un vertex shader et un fragment shader du même nom que le dossier.

Afin de s'assurer que les shaders sont chargés une unique fois pour être ensuite réutilisés partout dans le code, on utilise la classe statique **ShaderLoader** dans *utils/shaders/*. Elle permet le chargement et l'initialisation de shaders dans *scene.initialize()* (à réaliser avant l'initialisation des objets nécessitant les shaders en question !). Les shaders sont ensuite accessibles dans l'intégralité du code, la classe étant statique.

Remarque : par la suite, on a préféré dans d'autres cas utiliser des instances non statiques mais ***extern*** *de classes pour gérer le stockage et l'accès à des paramètres globaux dans l'intégralité du code.*

Shaders utilisés :

- **mesh :** shader fourni de base. Il est utilisé pour les planètes.
- **uniform :** permet **l'éclairage uniforme** d'une texture, des deux côtés, sans prendre en compte le lighting. Il est utilisé pour la texture de galaxie en fond d'écran, et l'était aussi pour le Soleil avant d'être remplacé par *lava*.
- **lava :** variante animées du shader *uniform*. Le vertex shader applique un **mouvement circulaire tangent à la surface** à chaque vertex (en fonction du temps et de la position du vertex). Le fragment shader applique une **saturation des couleurs selon une combinaison sinusoïdales de 2 bruits de perlin 3D** se déplaçant en fonction du temps. Le résultat est une texture magmatique animée, utilisée pour le Soleil.
- **instanced :** reçoit sous la forme de VAO et VBO une liste de positions (vec3), rotations (mat3) et facteurs d'échelle (float) pour réaliser de **l'instancing** d'astéroïdes. Le code OpenGL est défini dans la fonction *draw_instanced* de *utils/opengl/instancing*. Ce shader réalise aussi un **bump mapping** des normales depuis les bump maps associées aux textures des astéroïdes.
- **shield :** shader du billboard du bouclier anti-astéroïdes du joueur, qui n'utilise pas de texture. Il combine une couleur transparente bleue avec une **animation sinusoïdale** de surbrillance suivant la direction du vaisseau et des **animations d'onde de choc** lors de collisions avec des astéroïdes. Pour ce dernier, on passe à chaque frame au shader une structure uniforme contenant la position du choc et le temps depuis la collision. Pour envoyer une telle structure, il a fallu utiliser des **UBO (Uniform Buffer Objects)**, qui ne semblent pas être pris en charge dans CGP. La gestion de l'envoi de ces données ainsi que les appels OpenGL sont réalisés dans *utils/opengl/shield_ubo*.

Autres :

- **bumpy :** était destiné au **bump mapping**, il a servi pour expérimenter, et a été remplacé par instanced.
- **aura :** était destiné à réaliser un **effet d'aura** autour des planètes. Est plus ou moins achevé, mais pas utilisé.

Remarque : certains shaders lancent le warning cgp "try to send uniform variable to a shader that doesn't use it". Même après avoir marqué ces uniforms comme expected=false, ce warning apparaît toujours.

3.2 Ceintures d'astéroïdes et Multithreading

Une fois le bottleneck de la communication CPU - GPU résolu par l'instancing, et le bottleneck du GPU résolu par l'utilisation intelligente de meshs "low poly", il a fallu régler le bottleneck du CPU pour passer à un très grand nombre d'astéroïdes simulés en temps réel.

OpenGL étant *single-threaded* (contrairement à d'autres API comme Vulkan), du multithreading a été mis en place au niveau des calculs les plus lourds uniquement : la simulation des astéroïdes, la gestion de leurs collisions avec le bouclier du joueur, et la détermination du mesh à employer.

Fonctionnement de base :
La classe **AsteroidBelt** emploie pour la simulation une instance de **AsteroidThreadPool** (voir *celestial_bodies/asteroid_belt*). Selon le nombre total d'astéroïdes à simuler, *AsteroidThreadPool* lance un certain nombre de threads, chacun chargés d'une portion du vecteur des astéroïdes. Le nombre exact est défini dans la contante *ASTEROIDS_PER_THREAD*. Au démarrage et à l'arrêt du programme, les threads lancés et arrêtés sont affichés dans la console.

Synchronisation :
Afin de synchroniser les cycles de simulation avec le framerate du thread principal, on utilise la classe **ThreadSync** définie dans *utils/threads*. Elle permet de faire en sorte que chaque thread se mette en pause après un cycle, et de les relancer à la fin de chaque calcul de frame.

Remarque : elle permet aussi en théorie de synchroniser le thread principal avec les threads de calcul en attendant qu'ils aient terminé leurs calculs pour afficher la frame. En pratique, la mise en place de cette solution a provoqué des lags importants, voire des deadlocks. On ne garde donc qu'une synchronisation partielle : les threads de calculs ne calculeront jamais trop de cycles, mais s'il on du retard le thread principal ne les attendra pas. On obtient alors un rendu fluide où seuls les astéroïdes laggent, indiquant qu'il faut baisser le nombre d'astéroïdes par thread (s'il reste des coeurs supplémentaires disponibles sur la machine).

À titre indicatif, pour entre 100 000 et 200 000 astéroïdes, le programme total tourne en 4K 60 FPS avec fluidité sur une carte NVIDIA RTX 3050 mobile et un processeur AMD Ryzen 6800 HS, en n'utilisant que 12% à 20% de la puissance CPU (soit 2 à 3 coeurs à 100% d'utilisation).

Autres utilisations :
Les fichiers de *utils/threads/* définissent aussi d'autres utilitaires pour partager des données globales (notamment pour le GUI et les donnés de collision joueur-astéroïde). On utilise notamment des variables **atomic** pour les float et bool. La compatibilité avec **C++17** a été explicitement ajoutée pour Linux dans CMakeLists.txt (elle est supportée par défaut par Visual Studio sur Windows) afin de disposer de **shared_mutex** dans la classe **ReadWriteLock** dans l'optique de partager avec efficacité la position, la vitesse et la taille du vaisseau du joueur aux threads d'astéroïdes à chaque frame dans l'optique de calculer des collisions. On utilise aussi la classe **ThreadSafeDequeue** pour gérer efficacement les données d'animation de collision astéroïde / bouclier.

3.3 Caméra

Le déplacement du joueur est accompagné par une caméra personnalisée. L'objectif était de pouvoir suivre aisément la position et l'orientation du vaisseau. Afin de faciliter la prise en main de la caméra, on utilise un **modèle de caméra entièrement personnalisé**, qui utilise pour son orientation un **vecteur normalisé pointant vers la direction de regard**, et un **vecteur normalisé perpendiculaire au précédent représentant la verticale locale**. Le code est dans *utils/camera/*.

Positionnement de la caméra par rapport au vaisseau
- La caméra suit la position du vaisseau à **distance constante** (cette distance peut être changée depuis la GUI).
- Le vaisseau se situe systématiquement **au centre de l'écran**.
- L'orientation de la caméra ne suit pas immédiatement celle du joueur, donnant une impression de fluidité du mouvement (et permettant d'avoir une vue variable du vaisseau contrôlé).

Concrètement, les orientations de la caméra des **12 frames précédentes** sont stockées dans un buffer. À chaque frame, l'orientation de la caméra est calculée comme un mélange à 70%/30% de l'orientation de la 12ème frame et de l'orientation actuelle du vaisseau via un jeu de rotations et orthonormalisation. Le code est dans *utils/controls/player_object*, notamment la fonction *step()*.

Remarques : cette animation est frame-dependent, elle a été calibrée pour 60 FPS. Je n'ai pas trouvé le moyen d'obtenir un effet similaire en utilisant uniquement le timer global. Les paramètres ont été ajustés pour montrer une grande flexibilité de la rotation de la caméra par rapport au vaisseau, tout en étant suffisamment limitants pour éviter qu'à vitesse de rotation maximale le vaisseau ne tourne plus vite que la caméra.

3.4 Contrôle au clavier

Le vaisseau se contrôle au clavier.

Contrôles :
- **Barre espace :** accélérer. Le vaisseau met 3 secondes à accélérer à sa vitesse maximale, et 3 secondes à s'arrêter depuis sa vitesse maximale. Pour maintenir la vitesse maximale, garder la barre espace pressée.
- **Flèches :** tourner à gauche, à droite, monter, descendre. La vitesse de rotation maximale est atteinte au bout de 0.25 secondes.
- **Q - S (azerty) / A - S (qwerty) :** rouler dans le sens anti-horaire / horaire. La vitesse de rotation maximale est atteinte au bout de 0.12 secondes, et elle est deux fois plus rapide que la vitesse de rotation sur les autres axes.
- **A (azerty) / Q (qwerty) :** activer / désactiver l'affichage du vaisseau. L'option est aussi disponible depuis la GUI.
- **Z (azerty) / W (qwerty) :** activer / désactiver le bouclier anti-astéroïdes. Lorsqu'il est activé, le vaisseau peut entrer en collision avec les astéroïdes et les pousser.

- **E (azerty) / E (qwerty) :** activer / désactiver le rayon laser frontal. Lorsqu'il est activé, le vaisseau peut détruire les astéroïdes en les touchant avec le rayon laser, à une distance maximale égale à la distance du Soleil à Saturne. Le rayon de collision du laser est 5 fois plus grand que le rayon visible.

Effets particuliers
- Le joueur ne peut pas passer à travers les planètes et le Soleil, il sera repoussé.
- La caméra qui suit le joueur ne peut pas non plus passer à l'intérieur d'une planète par inadvertance, grâce à un système de détection de collision avec la caméra.

— Les paramètres de collisions et de la GUI sont gérés via des instances **extern** globales de classes disposant de variables *thread-safe (atomic)*.
— La classe **Controls** peut gérer un nombre arbitrairement grand de touches pressées par frame, car elle stocke l'état des touches dans une map et est parcourue entièrement à chaque frame.

Le code se trouve dans *urils/controls/controls*.

Note : le contrôle du vaisseau et les collisions avec les planètes sont inspirées du jeu sur mobile ***Galaxy On Fire 2****.*

Amusez-vous bien à contrôler le vaisseau dans les champs d'astéroïdes !