Rendu projet info graphique CSC_43043

Rodrigue Tavernier et Louis-Alexis Péneloux

Juin 2025

Lien code(github): lien
Lien rendu complet(Renater, valide jusqu'au 6 juillet): lien
Lien rendu complet(Google Drive): lien
Lien rendu complet(Stratus): lien

1 Description générale

Pour ce projet nous avons voulu faire une simulation de vaisseau spatial(très largement inspiré de Star Wars). L'objectif était de pouvoir contrôler ce vaisseau dans un environnement hostile: astéroïdes et vaisseaux ennemis. Nous voulions aussi de l'intéraction entre ces éléments: le vaisseau contrôlé peut détruire les obstacles grâce à un système de tir mais aussi être détruit par son environnement.

2 Structure du projet

Nous avons repris la structure des TP. Chaque objet de la scène est associé à une classe (vaisseau contrôlé — X-Wing, vaisseau IA, astéroïdes, etc.). En particulier, tout vaisseau hérite de la classe `ship` qui définit des comportements applicables à tous les types de vaisseaux. Les astéroïdes ont leur propre classe et sont définis comme un ensemble d'objets dont les informations sont stockées dans la classe(positions, vitesses etc...). On utilise plusieurs shaders selon les besoins: instancing et illumination Phong pour des sources différentes.

3 Description des implémentations

3.1 Modélisation des vaisseaux

Nous utilisons Blender pour modéliser deux vaisseaux issus de l'univers Star Wars : un X-Wing et un TIE Fighter. Il a notamment fallu gérer l'orientation des normales pour pouvoir appliquer les textures correctement. Les modèles sont ensuite exportés aux formats *.obj* et *.mtl*, puis analysés dans le code à l'aide de la fonctionnalité **advanced_obj_loader** de CGP.

En pratique, plusieurs exports sont réalisés (un par partie du vaisseau) pour permettre l'animation de l'objet. Tous les *mesh_drawable* sont ensuite stockés dans un champ de la classe *ship*.

3.2 Commande du vaisseau principal

Les commandes du vaisseau principal sont implémentées dans la fonction **idle_frame** de la classe *ship* (touches A–Z–E–Q–S–D).

Chaque commande directionnelle (haut, bas, gauche, droite, roulis gauche, roulis droit) agit sur un vecteur *angular_acceleration*, qui est ensuite intégré à la vitesse angulaire de l'objet à chaque frame. La position est ensuite mise à jour en fonction de cette vitesse.

Le fait d'agir sur l'accélération (et non directement sur la vitesse) permet une variation plus fluide de l'orientation (C^1 plutôt que C^0).

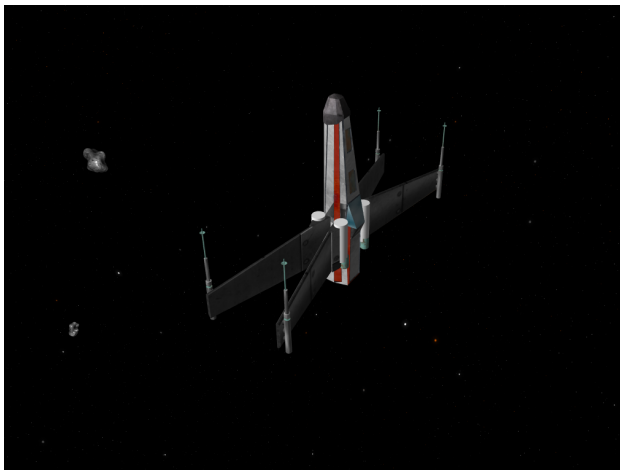

Figure 1: Modèle du X-Wing (gentil)

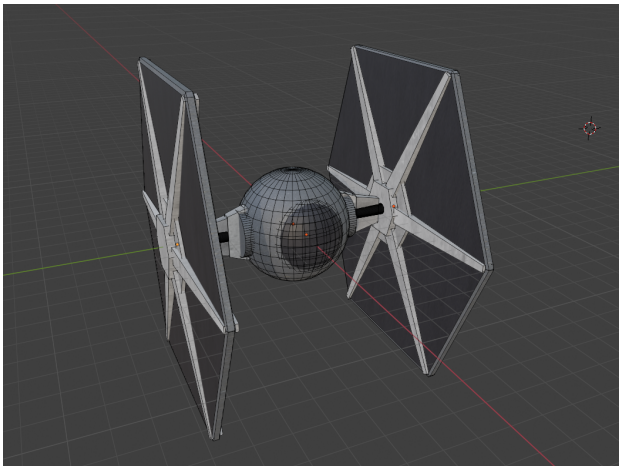

Figure 2: Modèle du TIE Fighter (méchant)

3.3 Gestion de la caméra

Nous utilisons une classe **camera_combat_mode** pour gérer la caméra.
Elle hérite de **camera_controller_first_person**. Pour un vaisseau donné (objet de type **ship**, pas nécessairement le joueur), la caméra se place en position arrière, avec un léger décalage dynamique basé sur l'accélération angulaire.

3.4 Animations

Une hiérarchie est utilisée pour représenter les différentes parties du vaisseau. Cela permet d'insérer des animations : pour le X-Wing, les ailes se replient lors du passage en mode accélération (touche espace), et les canons reculent légèrement au moment des tirs laser.

3.5 Demi-tour rapide

Le joueur peut effectuer un demi-tour rapide à l'aide d'une animation spéciale (touche O). Pour cela, nous avons défini quatre positions et orientations cibles de la caméra, entre lesquelles nous interpolons : SLERP (spherical linear interpolation) pour les orientations et polynômes de Hermite pour les positions.

3.6 Gestion des IA

Deux types d'IA sont intégrés à la scène :

- Une IA agressive qui suit une cible (un autre vaisseau ou le joueur), en pointant son vecteur d'accélération vers cette cible. Un angle maximal est imposé entre la direction de l'accélération et celle de la vitesse, pour éviter des virages trop brusques.
- Une IA passive, qui choisit aléatoirement une direction et une durée avant de changer de trajectoire.

Si une IA s'éloigne trop du joueur, elle est téléportée dans son dos. Cela justifie l'intérêt du demi-tour rapide : avec peu de vaisseaux à l'écran, le joueur en trouvera toujours devant lui après un retournement.

3.7 Animation de destruction

Pour animer la destruction des vaisseaux, nous utilisons la structure de données retournée par la fonction **advanced_obj_loader**, qui fournit un vecteur de *mesh_drawable* (un par partie avec texture associée). Pour simuler une explosion, les différents éléments sont dispersés dans des directions et rotations aléatoires.

3.8 Éclairage des réacteurs

Nous avons également implémenté un affichage spécifique pour les réacteurs arrières du vaisseau principal, en fonction des entrées clavier.

Un mesh convexe, correspondant exclusivement à l'intérieur du réacteur, est défini et associé à un shader dédié. Ce shader applique un modèle de Phong à partir d'une source lumineuse virtuelle, placée en amont du réacteur. La couleur de la lumière dépend des commandes directionnelles ou de la vitesse du vaisseau.

3.9 Modélisation des astéroïdes

Pour générer les astéroides, nous utilisons des primitives d'ellipsoïdes sur lesquelles ont ajoute un bruit de perlin. Les ellipsoïdes sont initialisés avec un scaling aléatoire selon chaque axe pour créer différentes formes. Afin de ne pas surcharger la mémoire, seulement un faible nombre d'astéroïdes sont effectivement générés. Pour en afficher une grande quantité, on pioche parmi les quelques mesh, puis on l'affiche à l'endroit voulu. Ensuite nous voulions éviter que les astéroïdes partent trop loin du vaisseau, car sinon ils ne sont pas utiles et occupent de la place. Pour éviter que cela n'arrive, on les contraint à rester dans un cube de côté `2*bound` centré sur le vaisseau. Si un astéroïde sort de ce cube, il est déplacé vers une des faces, de sorte à ce que sa trajectoire reste dans le cube. Les astéroïdes étant initialisés avec une vitesse constante(aléatoire), on peut projeter la nouvelle position en regardant l'intersection de la droite de la trajectoire avec les faces du cube. Le vaisseau se déplaçant, il faut mettre à jour le centre à chaque frame.

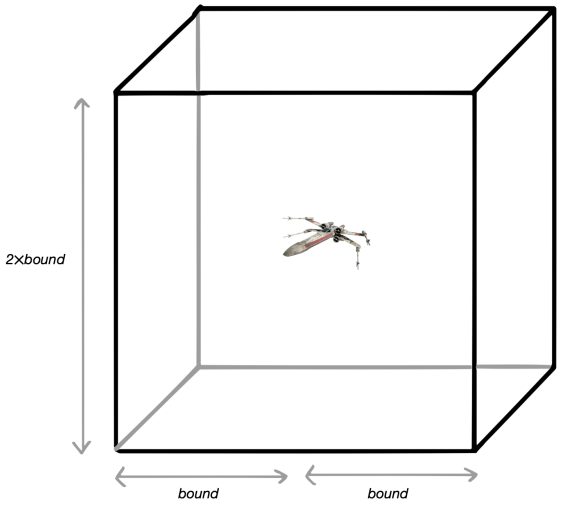

Figure 3: Domaine des astéroïdes

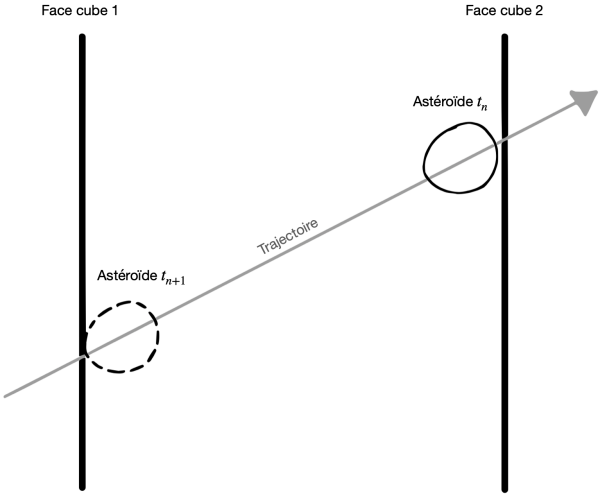

Figure 4: Trajectoire des astéroïdes

3.10 Destruction des astéroïdes

Un astéroïde est détruit s'il se fait toucher par un laser. Pour détecter la collision, définit un rayon aux lasers et aux astéroides. Si la distance entre les objets est inférieure à la somme de rayons, il y a collision. Les astéroïdes étant construits à partir d'ellipsoïdes, nous avons choisit un rayon simple: $\frac{scale.x+scale.y+scale.z}{3}$ une moyenne du scaling des 3 axes de l'ellipsoïde. Cette valeur peut être modifiée en fonction des paramètres de perlin appliqués.

Quand un astéroïde est détruit, on affiche des petits astéroïdes ainsi qu'un nuage de fumée composé de billboards.

3.10.1 Débris

Ils suivent les même principes que les astéroïdes. Seulement on en associe un nombre aléatoire pour chaque astéroïde. Les débris sont dirigés vers l'extérieur de l'astéroïde, avec vitesses et rotations aléatoires, pour donner une impression d'explosion.

3.10.2 Fumée

Les nuages de fumée sont un ensemble de billboards qui font face à la caméra. Il s'éloignent et se dissipent(fading) en modifiant la valeur alpha des fragments dans le shader, de sorte à ce qu'ils deviennent de plus en plus transparents jusqu'à disparaître. Comme il y en a beaucoup, nous avons décider d'utiliser de l'instancing ici. La difficulté a été de modifier les points dans le vertex shader de sorte à ce que les billboards regardent bien la caméra. Le nombre de nuages affichés étant variable, on utilise la méthode `update_supplementary_data_on_gpu()` de cgp.

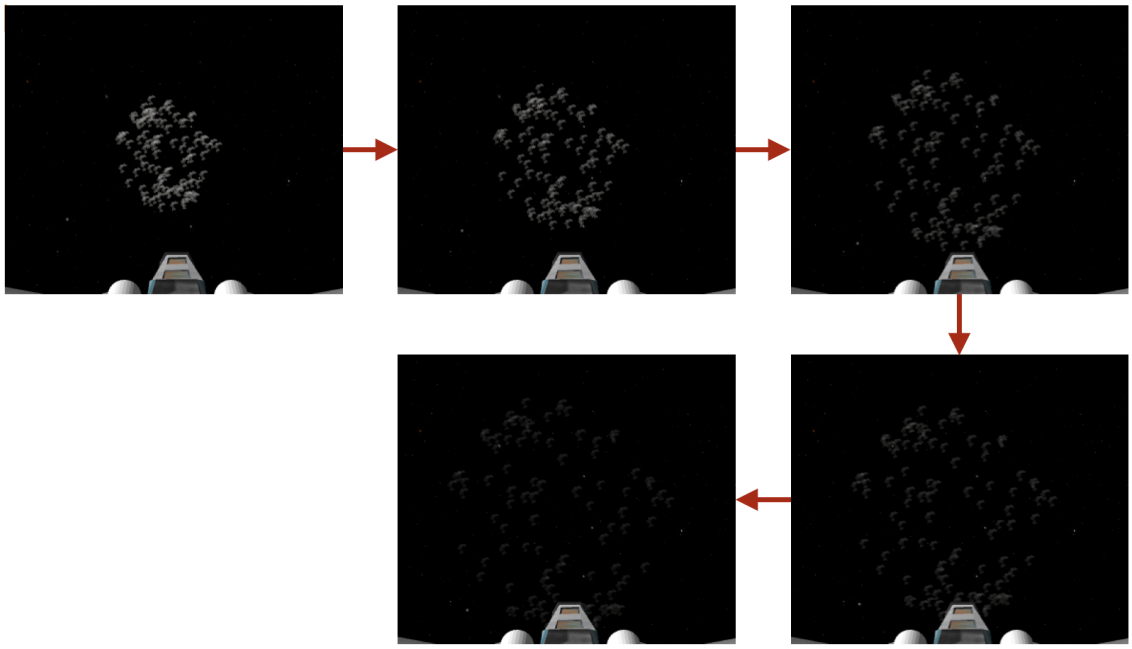

Figure 5: Dissipation de la poussière d'astéroïde

3.11 Lasers et illumination

Dans ce projet nous avons d'une part voulu ajouter la possibilité de tirer des lasers, mais qu'en plus, ces derniers soient des sources de lumière. Ce choix permet d'obtenir des "flash" quand on tire, et d'estimer la distance d'un laser à un objet. Contrairement à la lumière globale qui n'a pas de dissipation avec la distance à la source, la lumière des lasers diminue avec la distance aux surfaces. Concrètement, dans le fragment shader des objets dont on souhaite qu'ils soient illuminés par les lasers, on fait une boucle sur chaque source qui calcule la lumière Phong. La couleur finale est la somme des couleurs individuelles. Ce calcul étant coûteux quand le nombre de sources augmente, nous définissons un nombre maximum de lasers produisant de la lumière.
Les lasers étant généralisés pour tout vaisseau(classe `ship`), il est facile de les paramétrer(couleur, nombre maximal, vitesse, cadence de tir, positions initiales, etc...). Dans le cas du vaisseau contrôlé par l'utilisateur les lasers dépendent de la hiérarchie des mesh du vaisseau. Pour s'assurer qu'ils suivent bien les canons, on ajoute des éléments à la hiérarchie qui ont pour parents les canons.

3.12 Interface utilisateur(GUI)

Dans le GUI nous ajoutons la possibilité de changer les paramètres de perlin des astéroïdes et des débris. Ca permet notamment d'obtenir des astéroïdes plus ou moins grands ou lisses. Afin de ne pas surcharger de mises à jour, elles ne sont effectuées que si une valeur a été modifiée.

Figure 6: Simulation des lasers

4 En pratique

En pratique nous conseillons d'ajuster certains paramètres afin de rendre le jeu plus fluide. Nous avons testé le projet sur 4 architectures et les performances varient énormément. Les meilleures performances(fps) sont obtenues avec un macbook M1(même contre AMD Ryzen 5 7600 + gtx 4070 Ti). Par exemple, nous conseillons d'ajuster dans `scene.cpp`, le nombre d'astéroïdes, de vaisseaux IA, de billboards affichés ou la taille des maillages. En pratique, avec un M1 on arrive à atteindre 60fps de façon stable.

5 Idées non abouties

Nous avons essayé d'appliquer un flou gaussien pour donner un effet "néon" aux lasers.