Projet MGSI

Raymarching & Raytracing

Présentation

- Explication d’ensemble de l’application.
- Découpage des tâches.
- Précision point par point des classes principales de l’application.
- Problèmes rencontrés et solutions apportées.

L’application

Les différents choix d’implémentation de l’application :

- Raytracing ou Raymarching
- Image ou Shader
- Statique ou Interactif

Mise en place et découpage

Points clé pour la structure de base du projet :

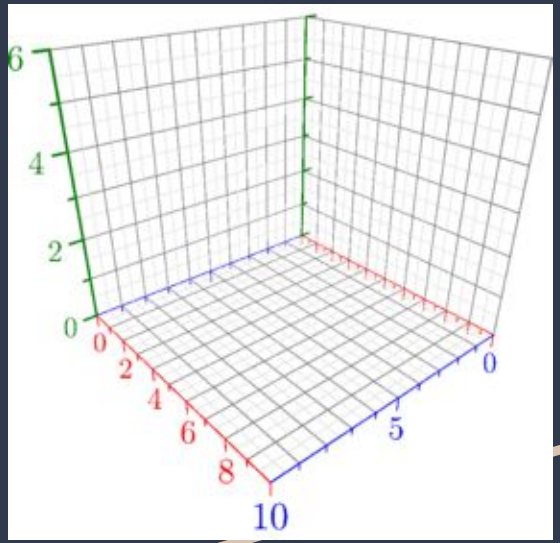
- Avoir des classes le plus indépendantes possible
- Permettre des mises à jour facile du code

Projet

La classe Projet s’occupe d’initialiser puis de lier toutes les classes entre elles.

- Elle gère l’initialisation de la Scene et Renderer
- Elle s’occupe de la boucle principale
- Elle détecte l'intervalle de temps passé entre les images

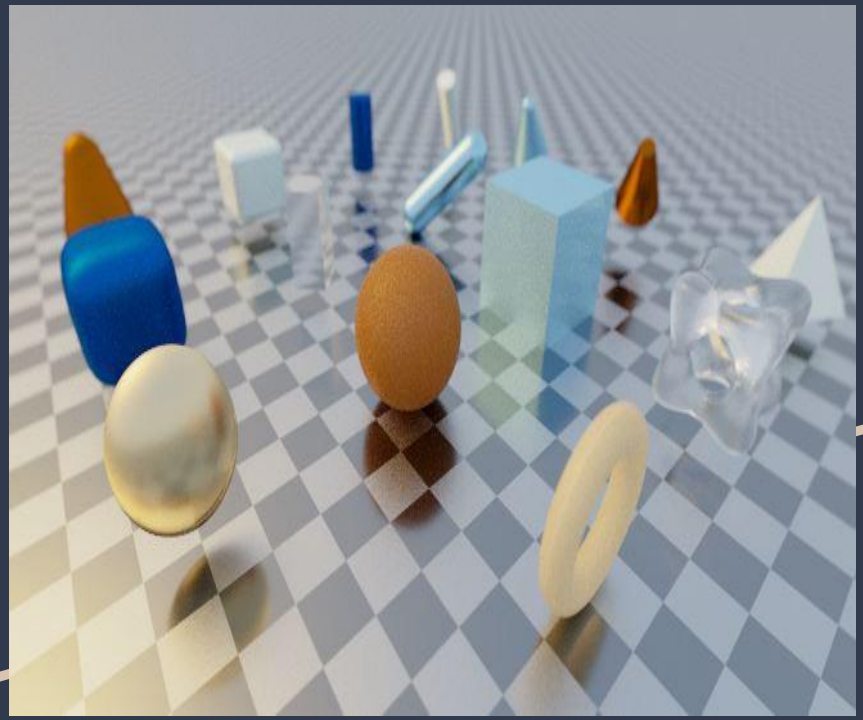
Scene



- La classe Scene s'occupe de stocker l'ensemble des objets et de les envoyer au Renderer

- Elle envoie les informations au Renderer dans un format encodé :
  1. Les objets sont transformés en liste de matrices et float
  2. La hiérarchie d'objet est transformée en liste de int pour les types
  3. La hiérarchie est liée aux listes de matrices et float d'objets

Objects



Par défaut un objet possède :

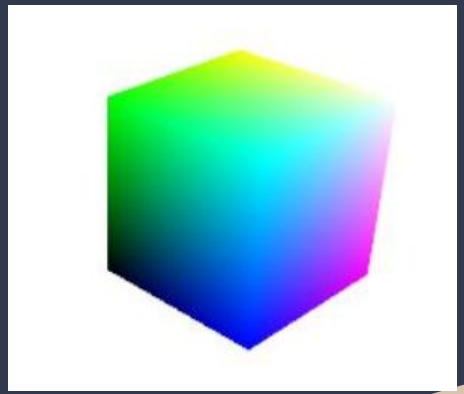
- Une position
- Une rotation
- Un vecteur pour la mise à échelle
- Un matériel

A cela peuvent s’ajouter d’autres informations selon le type de l’objet.

Il contient également une méthode virtuelle nous permettant de :

- Définir ses propriétés spécifiques suivant l’objet
- Retourner son type au format int, afin que le shader puisse l’identifier

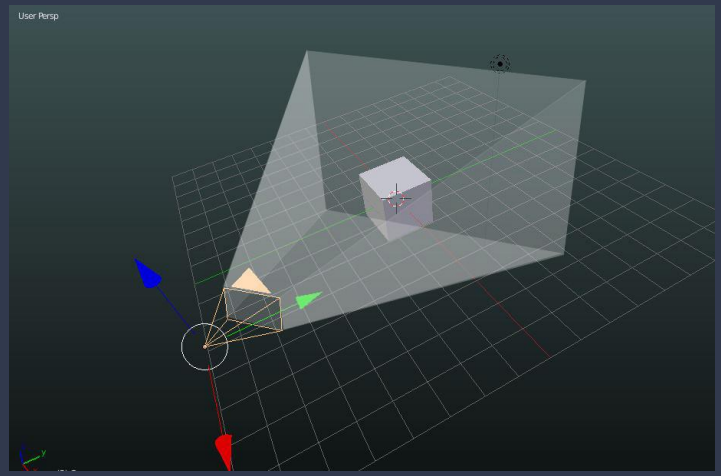
Renderer



La classe Renderer s'occupe de faire le lien entre la scène et les shaders, elle est découpée selon le modèle suivant :

- L'initialisation des shaders et uniforms
- L'affichage, lui même découpé en sous parties :
  - Affichage de la scène dans un framebuffer
  - Fusion de l'image actuelle et de l'ancienne
  - Affichage du de l'image finale

Caméra



La caméra a pour but de stocker les informations relatives au point de vue actuel.

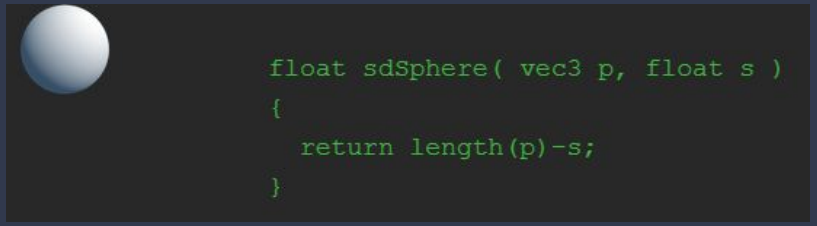
Elle joue un rôle essentiel pour l'affichage de l'application et nous permet de nous déplacer en temps réel dans la scène.

Elle est composée :

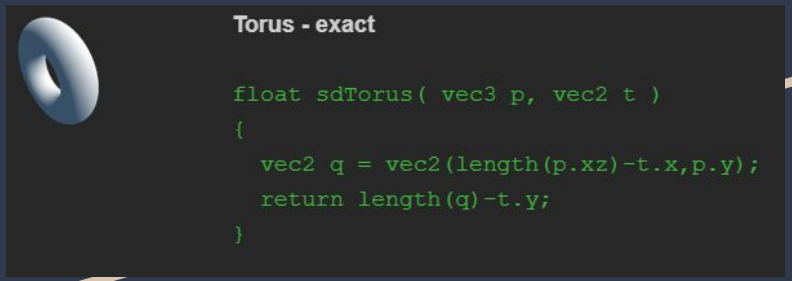
- D'un vecteur position pour la position de la caméra
- D'un vecteur de rotation pour permettre à la caméra de faire une rotation
- D'un vecteur en deux dimensions pour le champ de vision.

Shader

Exemple de fonction de distance :



```
float sdSphere( vec3 p, float s )
{
  return length(p)-s;
}
```



Torus - exact

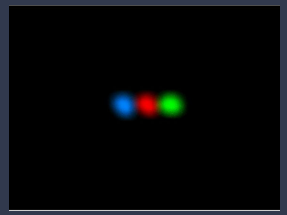
```
float sdTorus( vec3 p, vec2 t )
{
  vec2 q = vec2(length(p.xz)-t.x,p.y);
  return length(q)-t.y;
}
```

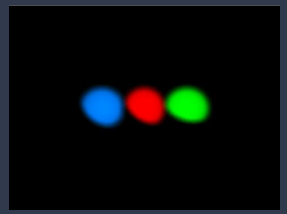
Un shader est un petit programme envoyé au GPU qui sert à modéliser des objets avec des input et des output provenant du renderer.

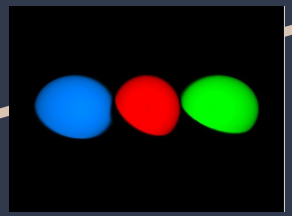
Pour modéliser les objets provenants de la scène, qu'on appelle primitives, on utilise des fonctions de distances qui permettent de représenter les objets autour d'un point, si un rayon partant de la caméra à une distance inférieure ou égale à une distance minimale, le rayon touche l'objet et on affiche alors la couleur correspondant à l'intersection.

Si des rayons partant d'un même pixel ne retournent pas la même valeur, le lissage du Renderer permet d'en faire une moyenne.

Profondeur de champ







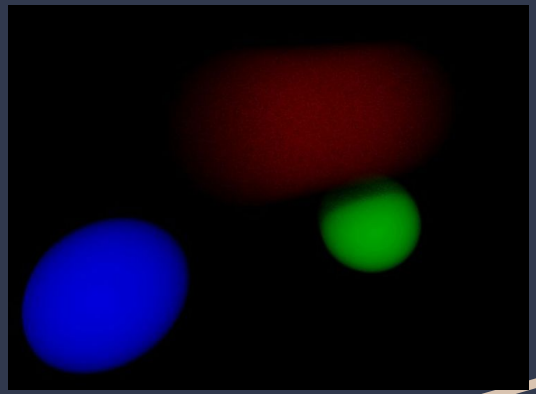
Exemple de rendu

La profondeur de champ est le facteur qui détermine la netteté relative des objets du scène depuis le point de vue de la caméra.

On définit une distance focale qui est la distance entre l'observateur et l'objet où l'image est nette. Plus la distance entre un objet et l'observateur se rapproche de cette distance focale, plus l'objet apparaît net.

On utilise la profondeur de champ afin de mettre des objets en valeur dans un point de vue, la vision se concentre naturellement sur la partie nette de l'image, cela aide aussi à donner une impression de profondeur/perspective sur l'image

Flou de mouvement

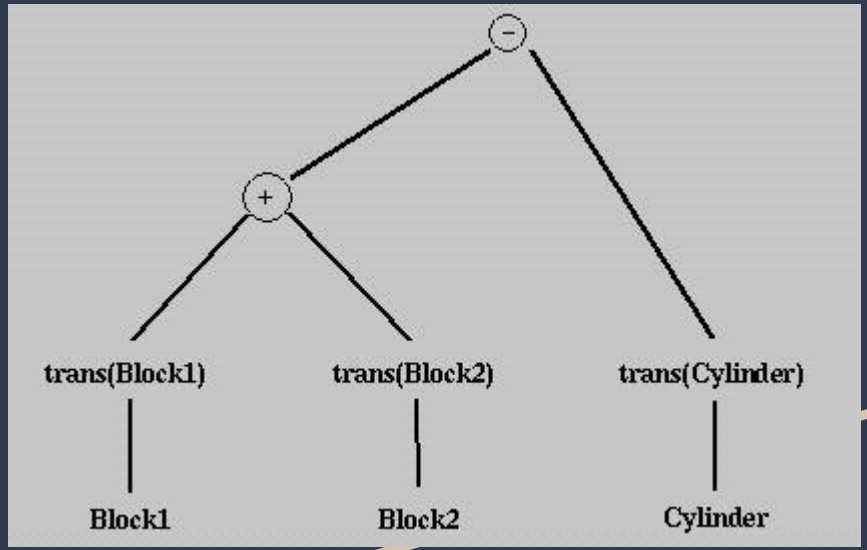


Le flou de mouvement s’inspire de la profondeur de champ pour créer un effet de mouvement relativement réaliste pour notre technique de rendu.

Nous utilisons une fonction qui génère un déplacement aléatoire de l’objet sur son axe de mouvement pour chaque rayon envoyé par pixel. De ce fait, les rayons ont une probabilité donnée de toucher l’objet qui dépend de sa vitesse ainsi que de sa direction relative à la caméra.

Cela nous crée, une fois reconstitué via le lissage temporel du Renderer, un flou de mouvement réaliste.

Arbre CSG et Shader



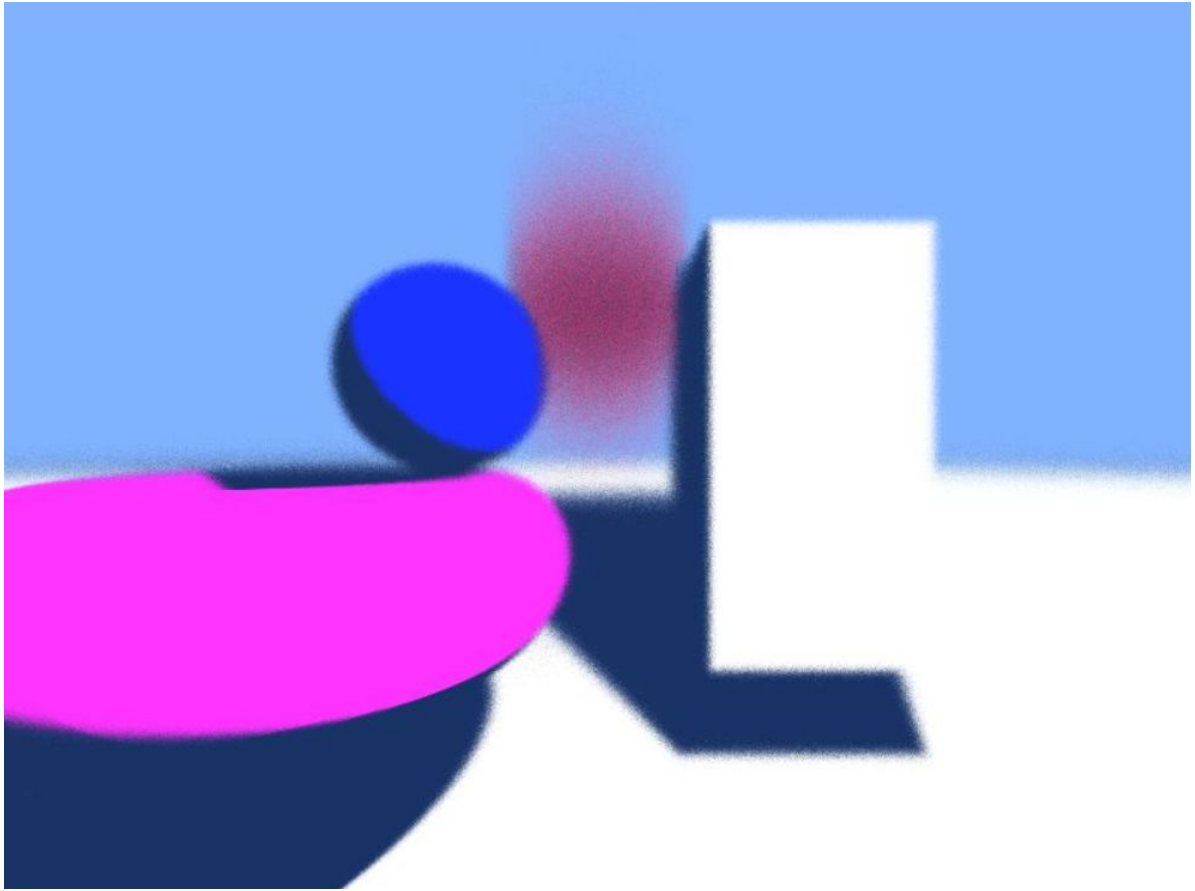
Afin d'envoyer les informations de la scène au shader, nous avons dû mettre en place les techniques suivante :

- Création de la classe CSG
- Envoie des informations de Scene à Renderer
- Envoie des données au Shader
- Retranscription du Shader

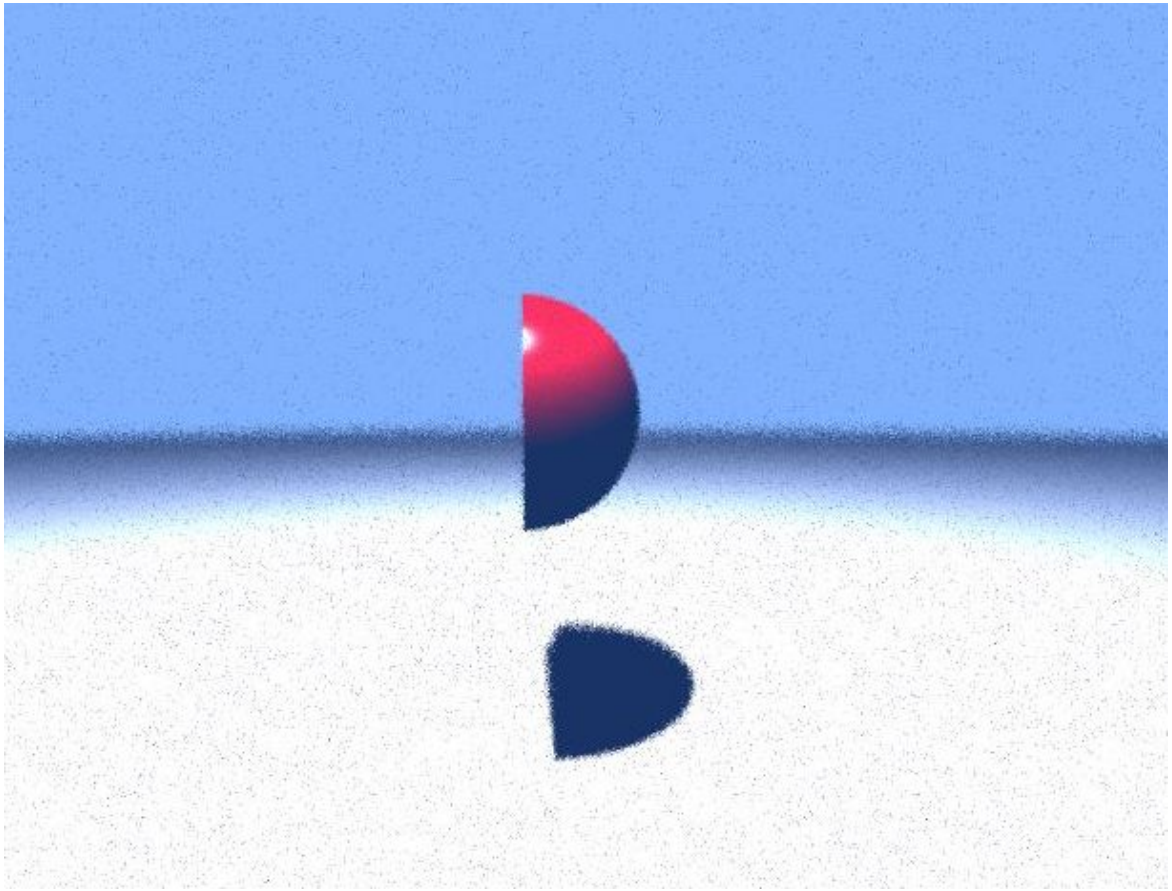
Problèmes rencontrés

Problèmes actuels :

- L'axe Y est inversé
- L'application demande énormément d'uniforms
- L'application en ray tracing n'a pas de réel gain de performances en mouvement
- En CSG, la différence est incalculable en raytracing sans lancer le rayon au moins deux fois



Rendu obtenu avec le shader ‘specular’



Rendu obtenu avec le shader ‘raytracing’ avec une intersection d’objet

Fin