Master Informatique, Synthèse d'Images et Conception Graphique

Introduction à la Synthèse d'Images Réalistes - Projet

Julie RICOU

Enseignant :

Maxime MARIA

Mai 2023

Table des matières

Introduction

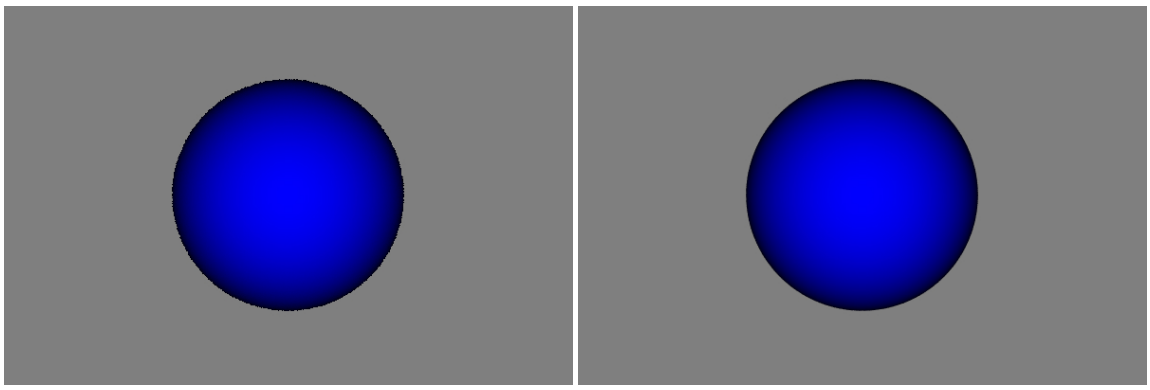
Durant cette UE, nous avons vu comment créer un moteur 3D en nous concentrant notamment sur les méthodes d'éclairage, les détections d'intersections entre rayons et objets, ainsi que différents matériaux. Dans un premier temps, j'expliquerai le travail que nous avons dû réaliser tout au long de ces travaux pratiques avant de vous présenter plus en détails le travail que j'ai réalisé pour mon projet personnel.

Vous trouverez mon projet téléchargeable sur le dépôt Git suivant : https://git.unilim.fr/ricou3/isir.git. Vous pourrez essayer chaque TP en modifiant les variables "tpToInit" (numéro du TP) et "sceneToInit" (expliqué à chaque TP) dans le fichier `main.cpp`. Dans ce fichier, il est également possible de changer le nombre d'échantillons par pixel pour l'antialiasing grâce à la variable "pixelsSamples" et le nombre de rayons d'ombrage pour réduire le bruit lors de l'utilisation de lumières surfaciques grâce à la variable "lightSamples".

Travaux pratiques

2.1 TP1

Dans ce TP, nous avons vu comment afficher notre premier objet en créant notamment une caméra en vue perspective positionnable et orientable qui permet de générer un rayon depuis le centre de chaque pixel afin de calculer la couleur de ce pixel. Nous avons ensuite créé notre première fonction `intersect` afin de gérer l'intersection entre un rayon et une sphère. Pour finir, nous avons mis en place un système d'antialiasing afin de réduire l'effet de crénelage causé par le fait que nous lancions uniquement un seul rayon par pixel en lançant à la place plusieurs rayons par pixel puis en faisant la moyenne des résultats de ces rayons.



(a) Sphère sans antialiasing (b) Sphère avec un antialiasing de 32 rayons

FIGURE 2.1 – Sphère avec et sans antialiasing

Paramètres "sceneToInit" :

— 1 : Caméra aux paramètres : position = (0, 0, -2), lookAt = (0, 0, 79)
— 2 : Caméra aux paramètres : position = (1, 0, 0), lookAt = (1, 0, 1)
— 3 : Caméra aux paramètres : position = (0, 1, 0), lookAt = (0, 1, 1)
— 4 : Caméra aux paramètres : position = (4, -1, 0), lookAt = (-1, -1, 2)

2.2 TP2

Dans ce TP, nous devions ajouter un objet plan de la même manière que ce que nous avions fait pour la sphère. L'objectif principal de ce TP était la prise en compte des lumières, nous avons donc créé une classe de lumière ponctuelle. Le premier intégrateur que nous utilisions ne prenait pas en compte les sources lumineuses, nous avons donc dû en créer un nouveau. Ainsi, dans cet intégrateur-là, lors d'une intersection avec un objet, la contribution de toutes les sources de lumière est prise en compte dans l'éclairage final et des rayons d'ombrage sont lancés afin de savoir si un point est éclairé. Afin de vérifier si un point est éclairé, nous avons dû mettre en place la méthode `intersectAny` afin de vérifier s'il y a une intersection entre le rayon d'ombrage et l'objet. Cependant, contrairement à la méthode `intersect`, on ne va pas chercher l'intersection la plus proche, mais simplement s'il y a une intersection.

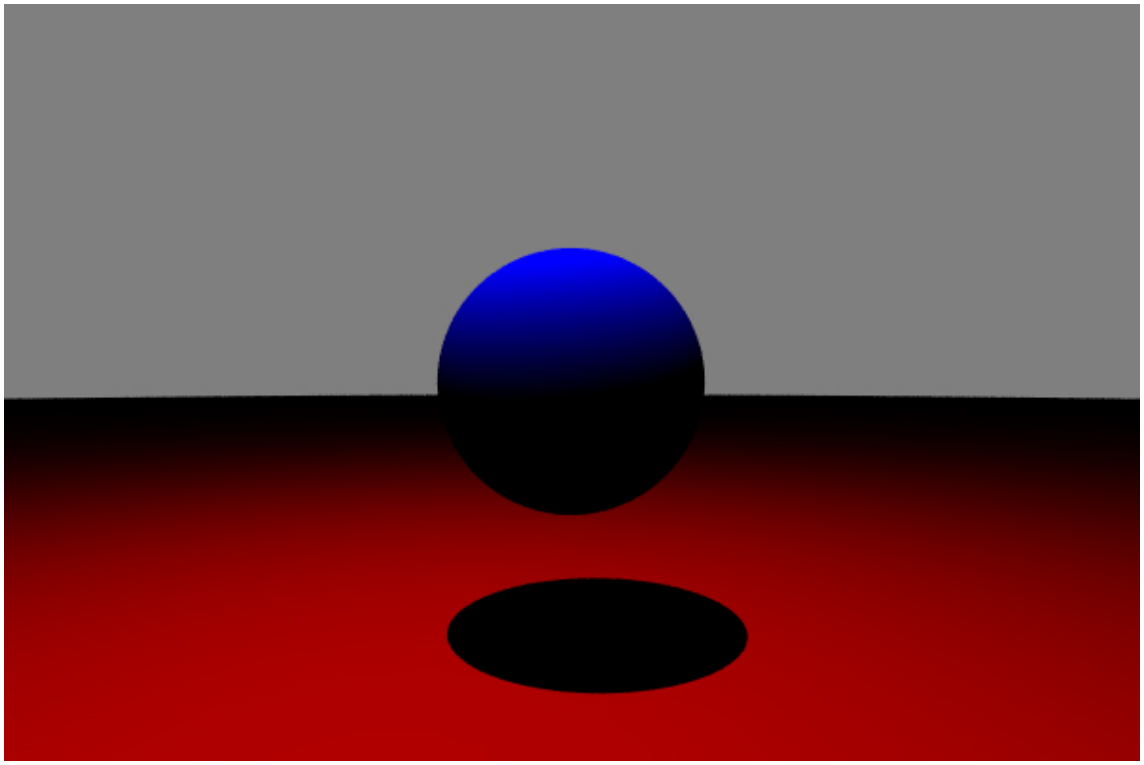
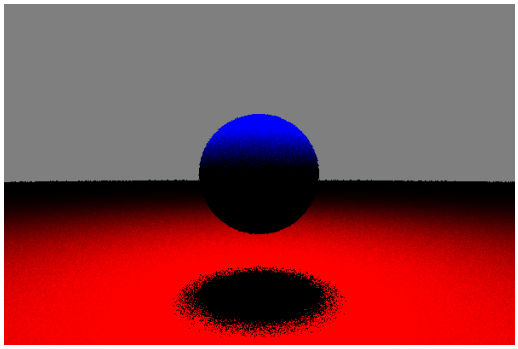


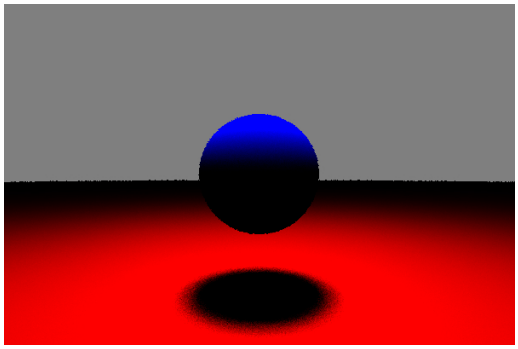
FIGURE 2.2 – Rendu final du TP 2

2.3 TP3

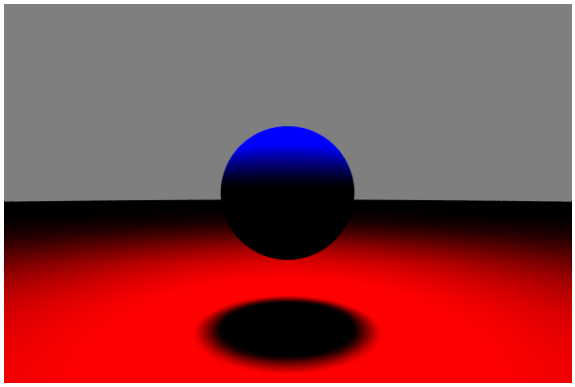
Dans ce TP, nous avons tout d'abord parallélisé la boucle de rendu grâce à l'instruction `#pragma omp parallel for`. L'objectif de ce TP était principalement l'ajout de lumières surfaciques afin de créer des ombres douces. Pour ce faire, nous avons créer une quad light, ainsi à partir de cette quad light, nous avons générer un certain nombre de rayons positionnés aléatoirement sur le quad. Plus on génère de rayons, plus les ombres seront douces et non bruitées.



(a) Ombre bruitée



(b) Ombre douce avec 32 rayons d'ombrage sans antialiasing



(c) Ombre douce avec 32 rayons d'ombrage et 32 rayons d'antialiasing

FIGURE 2.3 – Ombre bruitée et ombre douce

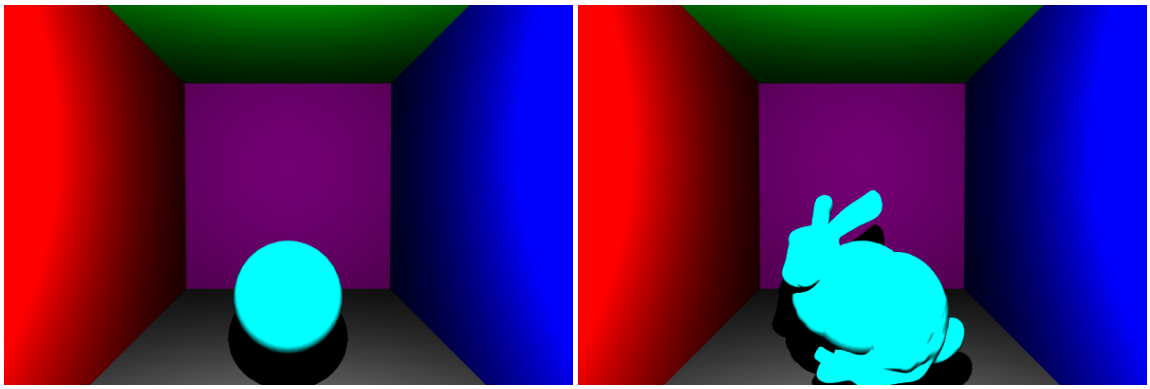
2.4 TP4

L'objectif de ce TP était d'ajouter de nouveaux objets 3d plus complexes au projet. Nous avons notamment géré l'intersection grâce à la méthode de *Möller et Trumbore* qui permet de trouver l'intersection entre un rayon et un triangle en 3 dimensions. Mon algorithme d'intersection est une version adapté du papier *Fast Minimum Storage Ray Triangle Intersection* [1]. Nous avons vite remarqué que le rendu pouvait être très long pour les objets comportant beaucoup de vertices. Le deuxième objectif de ce TP était donc d'optimiser la vitesse du calcul des intersections entre les rayons et les objets.

La première méthode que nous avons utilisée est la méthode de boîtes englobantes alignées aux axes (AABB). Cette méthode permet de créer des volumes englobants les modèles 3d afin de d'abord vérifier l'intersection avec ces volumes avant de tester l'intersection avec les triangles du modèle 3d. Algorithme d'AABB adapté du papier *An Efficient and Robust Ray–Box Intersection Algorithm* [2].

La deuxième méthode que nous avons utilisée et la méthode de hiérarchie de volumes englobants (BVH). Cette méthode permet de créer un arbre de volumes englobants, ainsi les triangles du modèle 3d seront divisés en plusieurs volumes et lors de l'intersection avec un volume, on descend dans l'arbre jusqu'à atteindre une feuille dans laquelle en test l'intersection avec les triangles la composant.

Ainsi avec ces méthodes, le rendu du modèle de Bunny sans antialiasing passe de 446ms pour un rendu sans optimisation à 115ms pour un rendu avec la méthode AABB et 0.5ms avec la méthode BVH.



(a) UV Sphère (b) Bunny

FIGURE 2.4 – Rendu final du TP 4

Paramètres "sceneToInit" :

— 1 : UV Sphère
— 2 : Bunny
— 3 : Bunny avec AABB
— 4 : Bunny avec BVH

2.5 TP5

L'objectif de ce TP était d'ajouter des BRDF à notre moteur. La BRDF permet notamment de décrire la lumière diffusée ou reflétée par un objet. La première BRDF que nous avons ajoutée est celle du modèle de Lambert. Le modèle de Lambert permet d'obtenir des objets totalement diffus, c'est-à-dire qui ne reflètent pas la lumière.

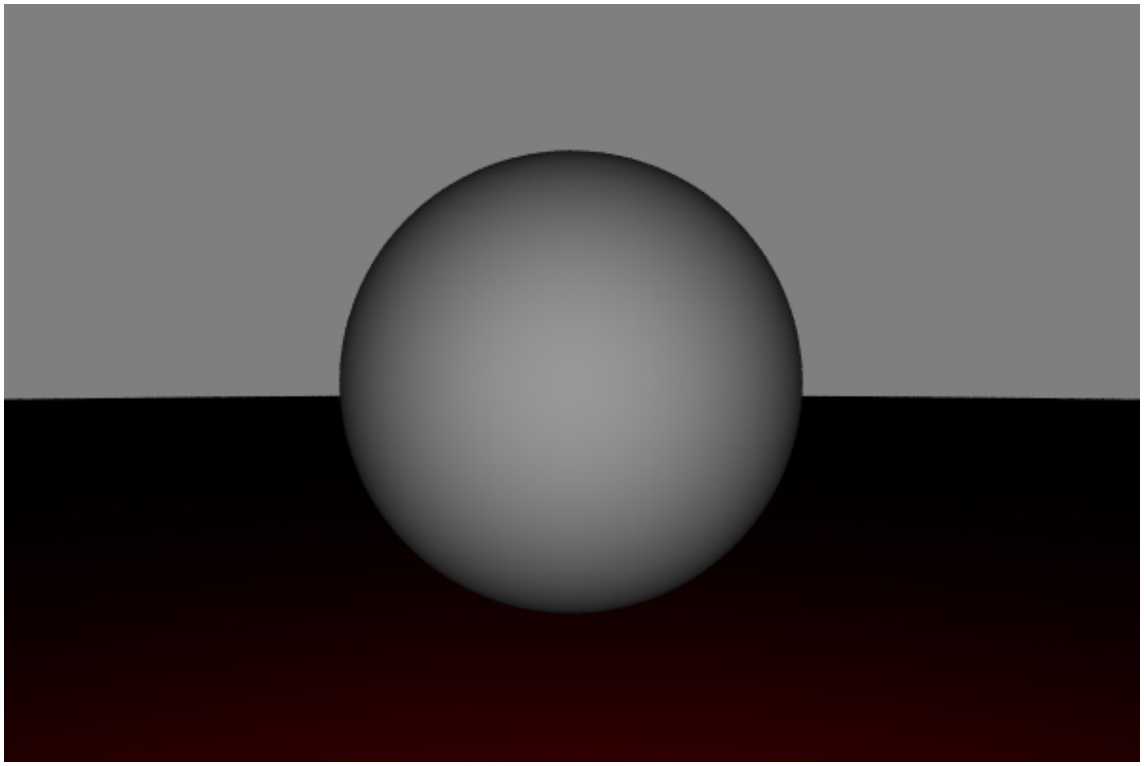
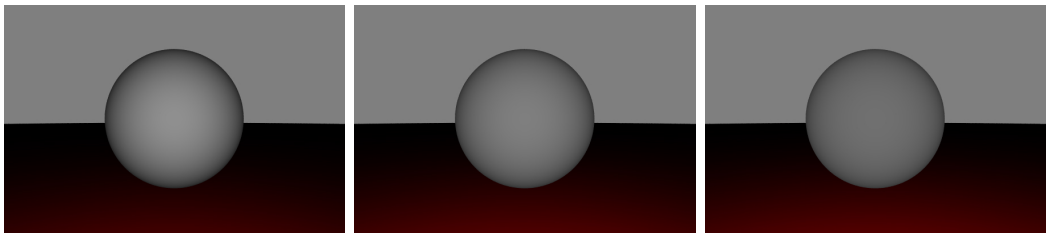


FIGURE 2.5 – Modèle de Lambert

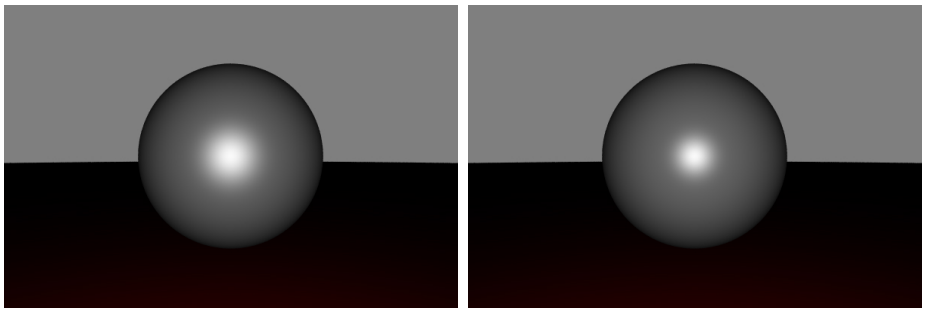
Nous avons ensuite ajouté le modèle d'Oren-Nayar. Ce modèle permet d'également prendre en compte la rugosité d'un objet en plus de son paramètre diffus. Cette méthode utilise la méthode de micro-facettes, ainsi le modèle simule le fait qu'une face ne soit jamais totalement lisse en simulant de toutes petites faces non-plates sur la surface de l'objet.



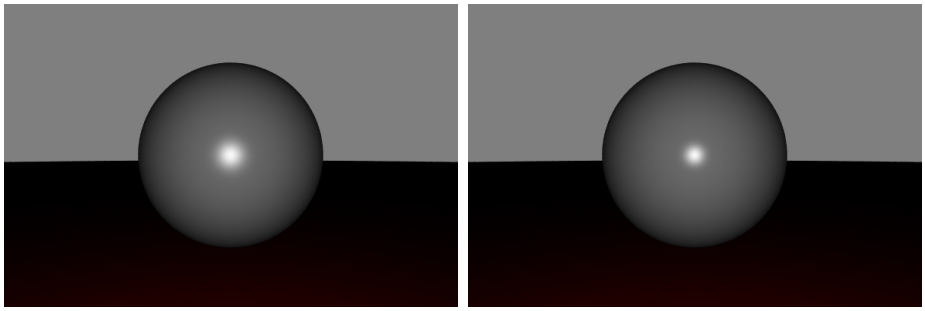
(a) alpha = 0.2 (b) alpha = 0.4 (c) alpha = 0.6

FIGURE 2.6 – Modèle d'Oren-Nayar

Le troisième modèle que nous avons ajouté au moteur est le modèle de Phong. Ce modèle permet de représenter le reflet spéculaire d'un objet. Bien que loin d'être physiquement réaliste, il permet de simuler facilement un reflet.



(a) s = 8 (b) s = 16



(c) s = 32 (d) s = 64

FIGURE 2.7 – Modèle de Phong

Le dernier modèle que nous avons mis en place pour ce TP est le modèle de Cook-Torrance. Ce modèle permet de créer la partie spéculaire des matériaux métalliques en utilisant également la méthode de micro-facettes décrite plus tôt.

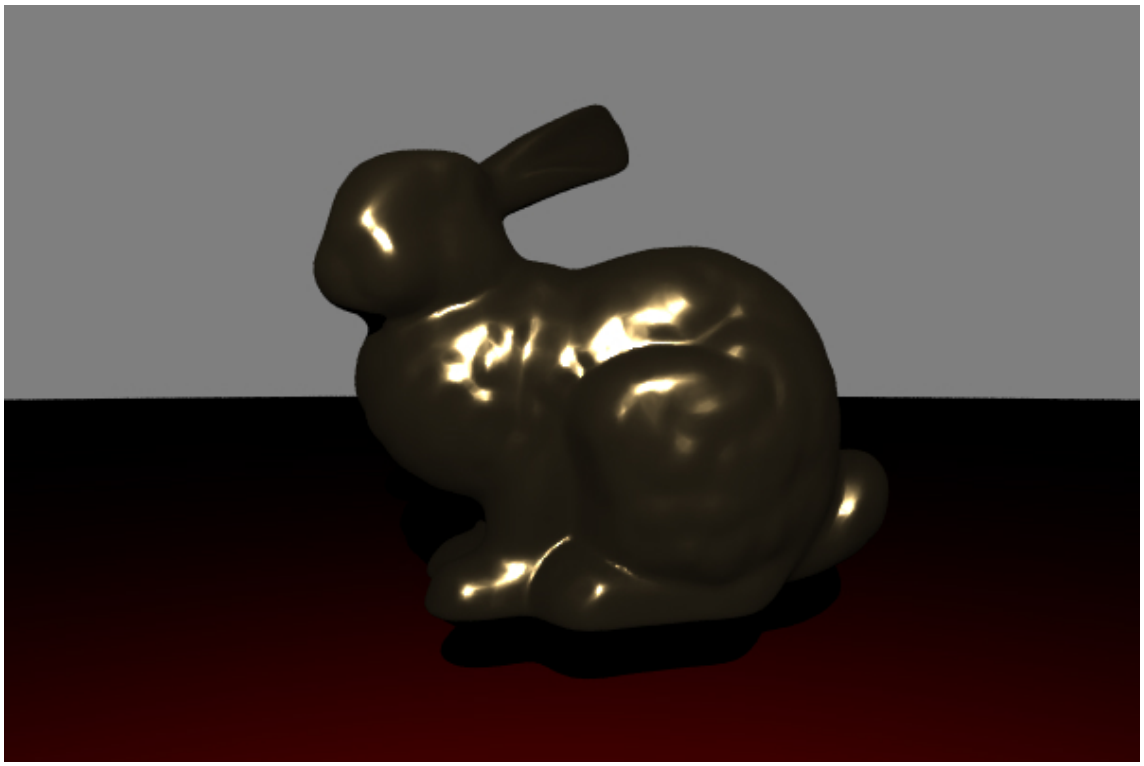


FIGURE 2.8 – Bunny avec le modèle de Cook-Torrance (metalness 0.7)

Paramètres "sceneToInit" :

— 1 : Lambert

— 2 : Oren-Nayar (paramètre `sigma` modifiable dans le dernier paramètre de l'objet)

— 3 : Phong (paramètre `s` modifiable dans le dernier paramètre de l'objet)

— 4 : Cook-Torrance (metalness 0)

— 5 : Cook-Torrance (metalness 0.5)

— 6 : Cook-Torrance (metalness 1)

— 7 : Cook-Torrance Bunny (metalness 0.7)

2.6 TP6

L'objectif de ce TP était d'ajouter au moteur les effets de réflexion et de réfraction. Pour ce faire, nous avons ajouté un nouvel intégrateur permettant de prendre en compte les rebonds des rayons lumineux afin de créer des surfaces réfléchissantes tel que des miroirs ou des surfaces transparentes.

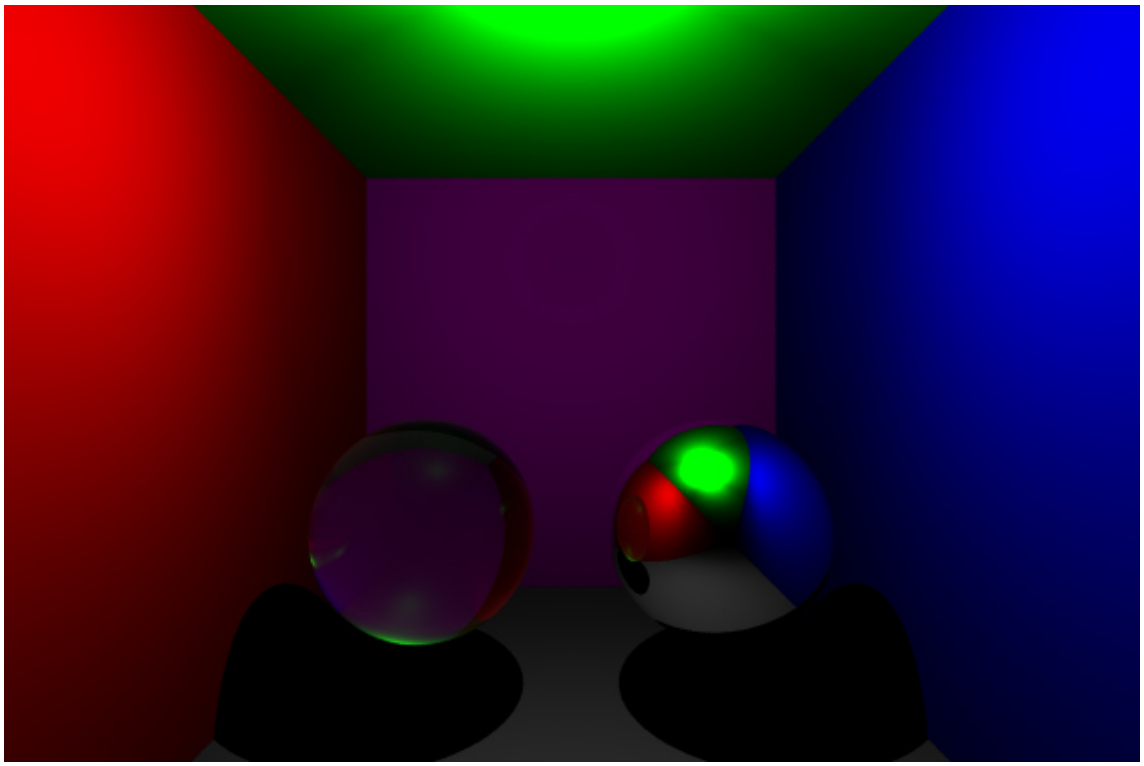


FIGURE 2.9 – Rendu final du TP 6

Paramètres "sceneToInit" :

— 1 : Scène de base avec lumière ponctuelle
— 2 : Scène de base avec lumière surfacique
— 3 : Une sphère métallique
— 4 : Deux sphères métalliques
— 5 : Deux sphères métalliques et le plan arrière métallique
— 6 : Une sphère transparente et une sphère métallique

2.7 TP7

Dans ce TP, nous avons mis en place un algorithme de *Sphere-tracing*. Cet algorithme permet de calculer l'intersection entre un rayon et une surface implicite. Les surfaces implicites étant des surfaces définies par une fonction contrairement aux surfaces définies par un maillage.

Pour calculer ces intersections, nous utilisons des fonctions de distance signée (SDF). Ces fonctions permettent de retourner la distance entre un point et le point le plus proche sur l'objet (la distance étant négative si le point est à l'intérieur de l'objet). Ainsi, cette distance va être ajoutée à la distance depuis le point d'origine du rayon puis on va à nouveau utiliser la SDF afin de calculer la distance jusqu'au nouveau point et ainsi de suite jusqu'à ce que la distance jusqu'au point soit inférieure à la distance minimale souhaitée ce qui correspondra à notre point d'intersection.

Pour ce TP j'ai donc intégré, en plus de la sphère implicite, une sphère creuse coupée, un cylindre arrondi et un anneau.

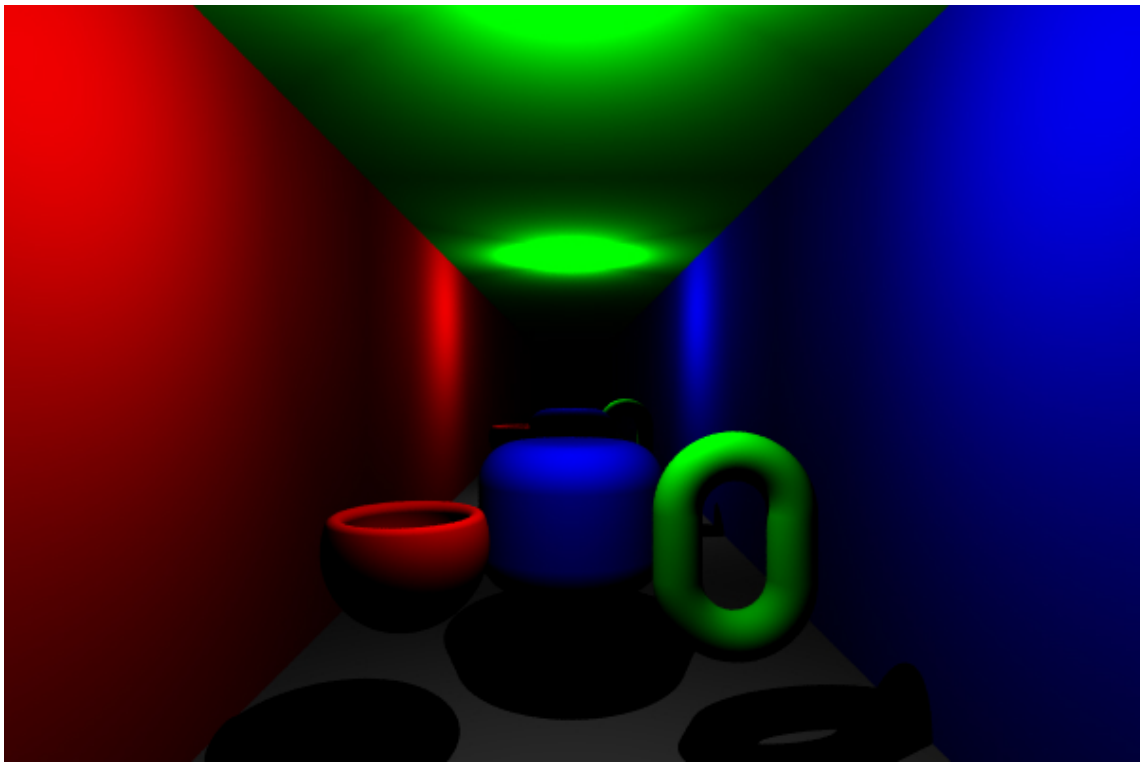


FIGURE 2.10 – Rendu final du TP 7

Paramètres "sceneToInit" :

— 1 : Sphère implicite
— 2 : Sphère creuse coupée implicite
— 3 : Cylindre arrondi implicite
— 4 : Anneau implicite
— 5 : Sphère creuse coupée, cylindre arrondi et anneau implicites

Ajouts réalisés

En plus des travaux pratiques, j'ai ajouté trois fonctionnalités.

3.1 Flou de profondeur

J'ai tout d'abord ajouté un flou de profondeur. Le flou de profondeur peut principalement être mis en place de deux façons : en sauvegardant la profondeur de chaque pixel avec un Z-buffer par exemple et en appliquant un flou à chaque pixel suivant sa profondeur ou en utilisant plusieurs caméras avec une position et une rotation légèrement différente puis en faisant la moyenne de tous les résultats.

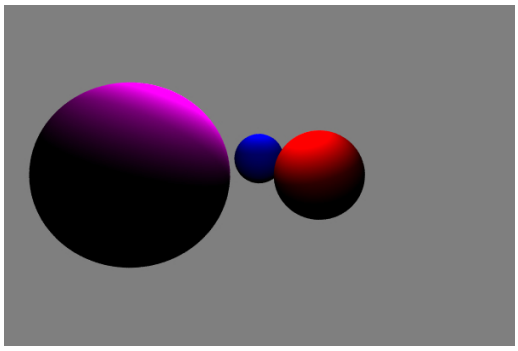
J'ai choisi d'utiliser la méthode d'addition de caméras. Je crée tout d'abord plusieurs caméras avec une position légèrement différente en x et y, mais avec la même direction qui correspondra au point de focus de notre flou de profondeur. Ainsi, on calcule le rendu pour chaque caméra (le rendu étant effectué en parallèle sur chaque caméra) puis on fait une moyenne de chaque rendu pour chaque pixel. Le rendu étant effectué plusieurs fois, bien qu'il soit calculé en parallèle, il peut devenir très long si la scène comporte beaucoup d'objets, de triangles ou si l'on veut un flou de profondeur très lisse puisque, plus il y a de caméras plus le rendu est lisse.

Durée du rendu :

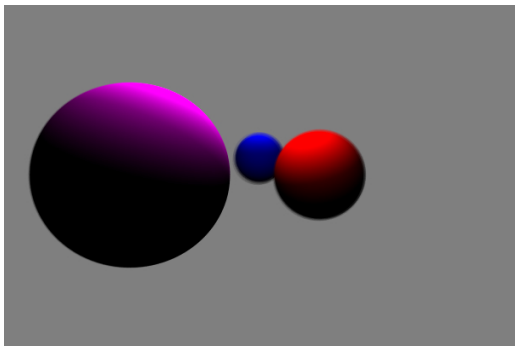
— sans flou de profondeur : 2ms

— avec 8 caméras : 13ms

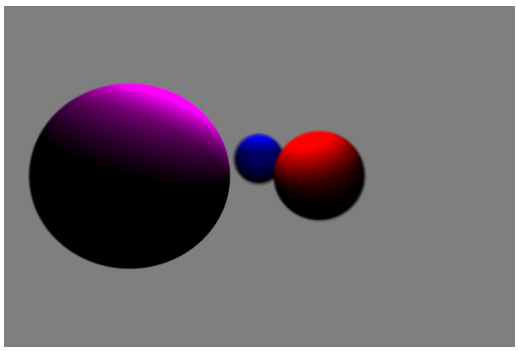
— avec 32 caméras : 49ms



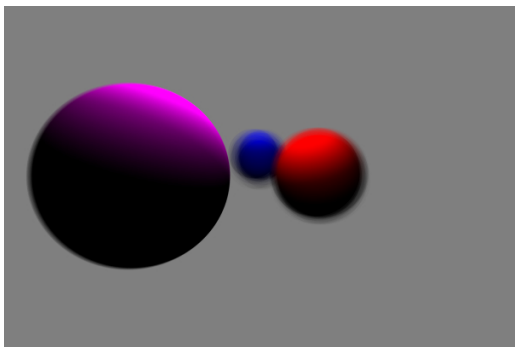
(a) Sans flou de profondeur



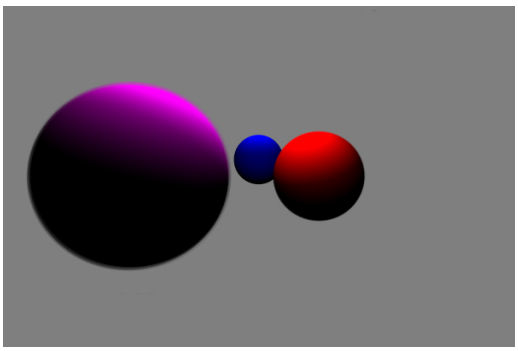
(b) Flou de profondeur avec 8 caméras (`aperture` 0.1, `focus` (0, 0, 3))



(c) Flou de profondeur avec 32 caméras (`aperture` 0.1, `focus` (0, 0, 3))



(d) Flou de profondeur avec 32 caméras (`aperture` 0.3, `focus` (0, 0, 3))



(e) Flou de profondeur avec 32 caméras (`aperture` 0.1, `focus` (0, 0, 16))

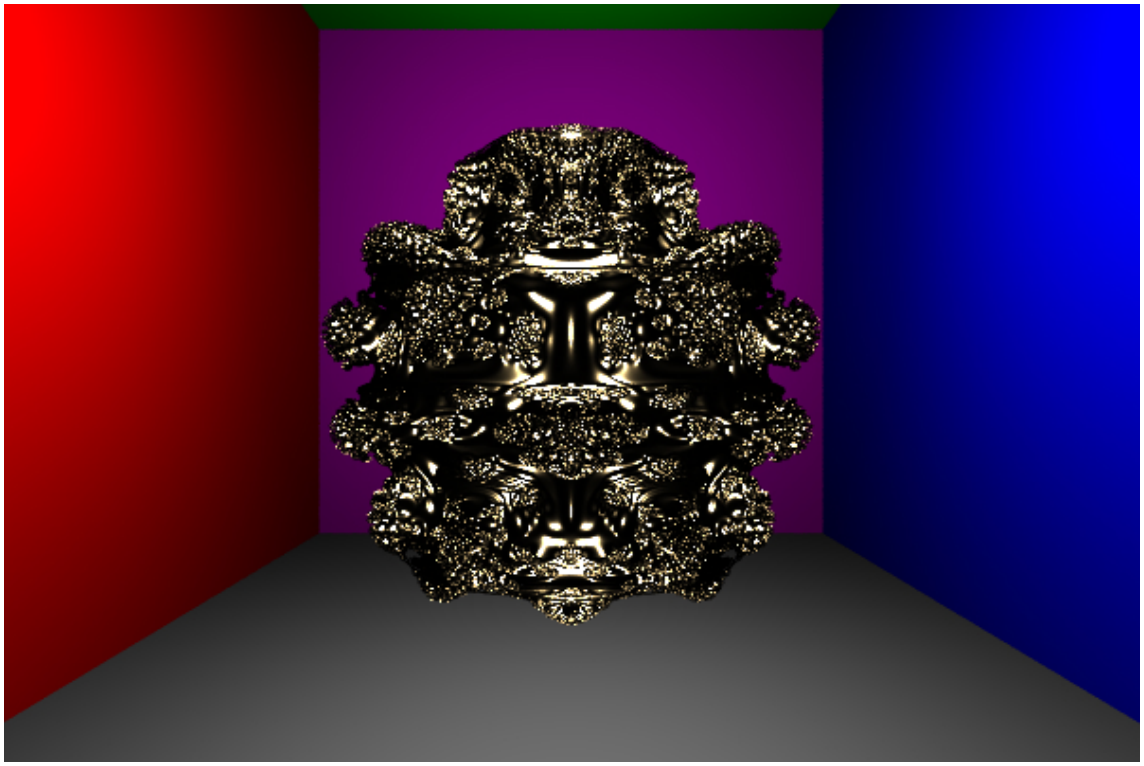
FIGURE 3.1 – Flou de profondeur

3.2 Mandelbulb

J'ai par la suite ajouté une version 3d de l'ensemble de Mandelbrot, le Mandelbulb. Les images de l'ensemble de Mandelbrot sont créées en parcourant les nombres complexes du plan. Pour chaque nombre complexe, on détermine si le résultat tend vers l'infini ou non lorsqu'on itère une opération mathématique. Les parties réelle et imaginaire de chaque nombre complexe sont considérées comme des coordonnées.

Le mandelbulb que j'ai implémenté est directement tiré du site d'Inigo Quilez :

Mandelbulb [1].



(a) Mandelbulb

(b) Mandelbulb zoomé

FIGURE 3.2 – Mandelbrot 3D (Mandelbulb) avec 3 itérations

1. Mandelbulb : https://iquilezles.org/articles/mandelbulb/

3.3 Fonction de déplacement

La dernière fonctionnalité que j'ai implémentée est une fonction de déplacement. Cette fonction permet d'ajouter une certaine valeur à chaque point de l'objet ce qui va déformer la forme de l'objet. Cette fonction est directement tirée du site d'Inigo Quilez : Fonction de déplacement [2]. Comme vous pouvez le remarquer sur l'image suivante, certains points des objets dont rendu en noir, et l'objet semble coupé par endroit ce qui n'est pas normal. Je pense qu'il s'agit d'un problème lors de l'intersection entre les rayons d'éclairage et d'ombrage et l'objet sdf, mais je n'ai pas réussi à résoudre le problème.

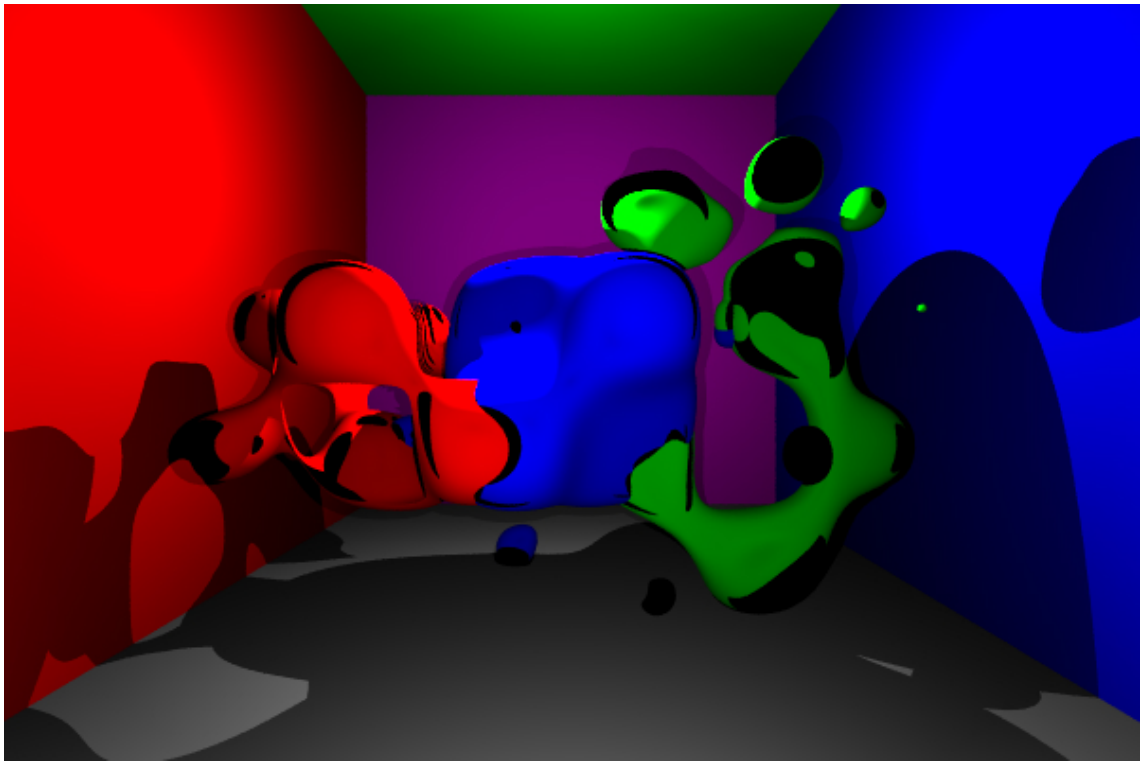


FIGURE 3.3 – Fonction de déplacement sur une hollow sphere, une sphere et un link avec comme fonction : sin(2.0f * x) * sin(2.0f * y) * sin(2.0f * z)

2. Fonction de déplacement : https://iquilezles.org/articles/distfunctions/

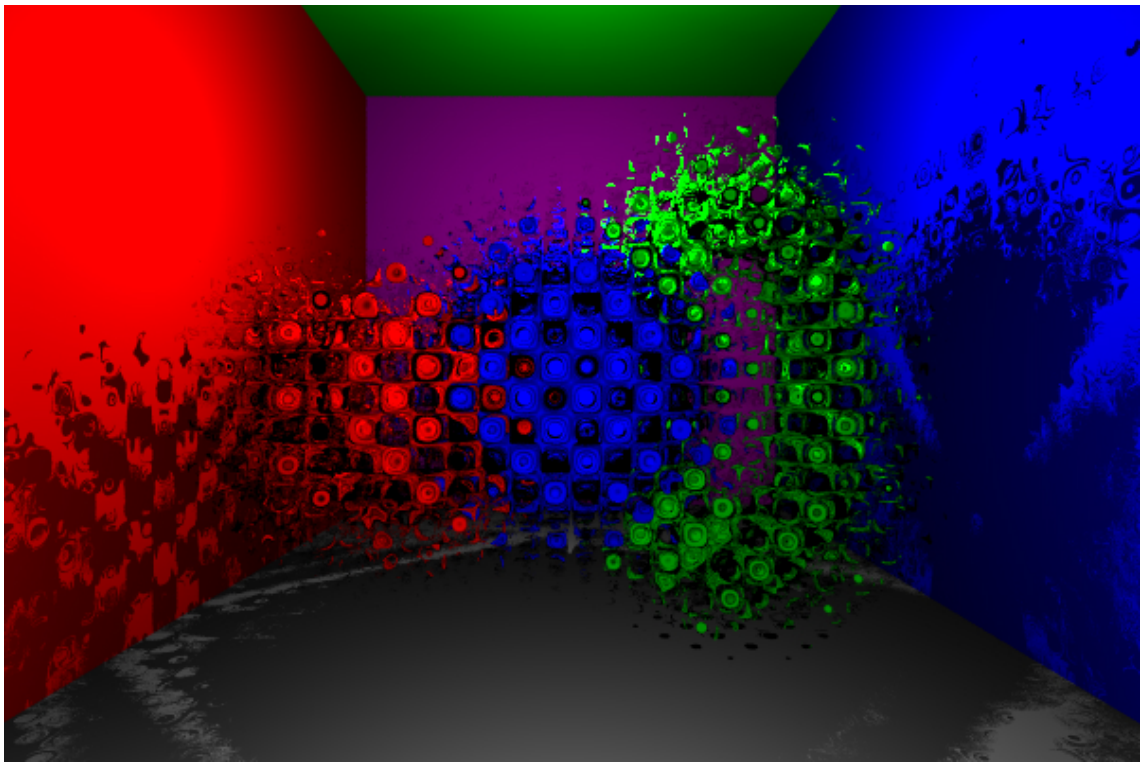


FIGURE 3.4 – Fonction de déplacement sur une hollow sphere, une sphere et un link avec comme fonction : sin(10.0f * x) * sin(10.0f * y) * sin(10.0f * z)

Paramètres "sceneToInit" :

— 1 : Flou de profondeur avec un point de focus en (0, 0, 2)

— 2 : Fonction de déplacement sur une hollow sphere, une sphere et un link

— 3 : Mandelbulb

— 4 : Mandelbulb zoomé

Conclusion

J'ai pu réaliser tous les travaux pratiques (hors exercices optionnels) et j'ai également pu ajouter quelques fonctionnalités en plus. J'aurais beaucoup aimé résoudre le problème avec la fonction de déplacement et j'aurais également souhaité ajouter la deuxième technique de calcul du flou de profondeur afin de pouvoir comparer les deux méthodes, mais je n'ai pas eu le temps d'ajouter cela.

Ce module m'a permis de bien comprendre de nombreuses choses sur l'éclairage notamment en me permettant de le mettre en pratique et de pouvoir visualiser l'impact des différents paramètres sur le rendu.

P.-S. : vous pouvez trouver de "belles" images du projet dans le dossier results/-HallOfShame

Bibliographie

[1] Tomas Moll et Ben Trumb. *Fast Minimum Storage Ray Triangle Intersection.* 2005.

[2] Amy Williams et al. *An Efficient and Robust Ray–Box Intersection Algorithm.* 2005.