ÉCOLE
CENTRALE LYON

Rapport de projet

MOS 2.2 - Informatique Graphique

Ray Tracing

Enseignant
M. Nicolas BONNEEL

Etudiant
Anass EL HOUD

Année scolaire
2020/2021

1. Introduction

Ce projet s'inscrit dans le cadre du cours MOS 2.2 Informatique Graphique à l'Ecole Centrale de Lyon avec Prof. Nicolas BONNEEL. Le but est de développer une scène 3D en se basant sur le principe de Ray tracing ou traceur de rayons.
La technique de Ray Tracing cherche à reconstituer le trajet inverse de la lumière, c'est-à-dire de la caméra vers les sources lumineuses. Pour ce faire, on place une image virtuelle dans la scène devant l'observateur (c'est cette image qui constitue le résultat final). Pour chacun des pixels qui composent l'image, on lance un rayon partant du point de vue (l'observateur) passant par le centre du pixel. La couleur du pixel traversé sera déterminée en suivant la trajectoire du rayon lancé vers la ou les sources lumineuses de notre scène.

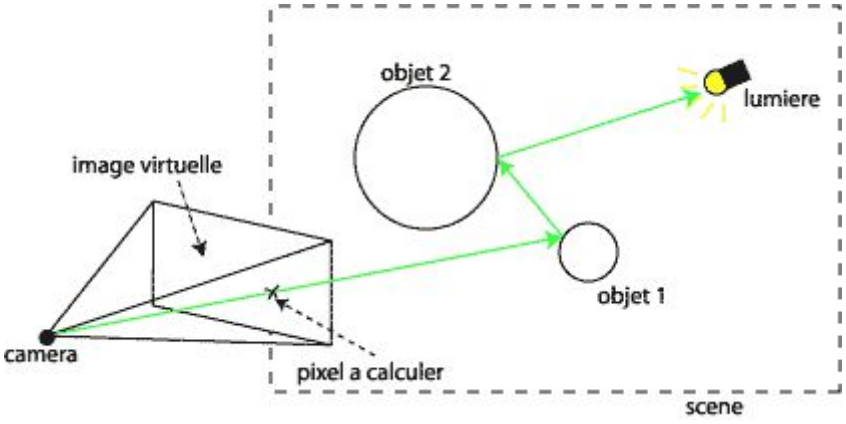


Image ref: Lanceur de rayons - Laure Heïgéas

Dans ce rapport, je vais présenter l'avancement de mon travail sur le projet de Ray Tracing du MOS 2.2 Informatique Graphique. Le code a été publié sur mon GitHub: https://github.com/anasselhoud/RayTracing-project

2. Idée globale

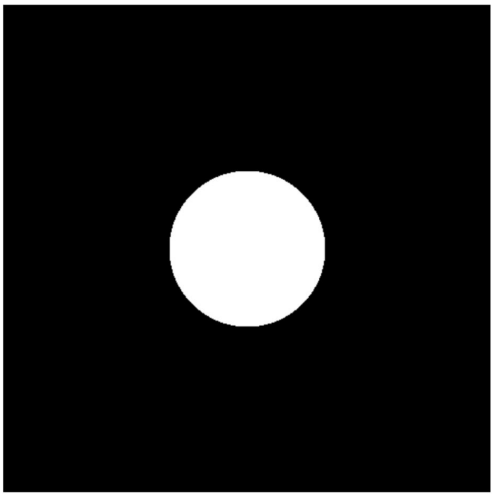
Je vais suivre le même enchaînement du cours. Je commencerai à coder les premières caractéristiques liées à l'intersection rayon-sphère, ombres portées et les surfaces spéculaires et transparentes jusqu'au ombres douces (cours 1 à 4).
Dans une deuxième partie, je commence la manipulation des maillages et les intersections avec les boîtes englobantes. C'est à partir de là qu'on devient de plus en plus proche de notre scène finale.

La troisième partie du projet est consacrée à la finition de la scène en ajoutant les textures et corrigeant les différents bugs restants.

3. Partie initiative (Cours 1 : Cours 4)

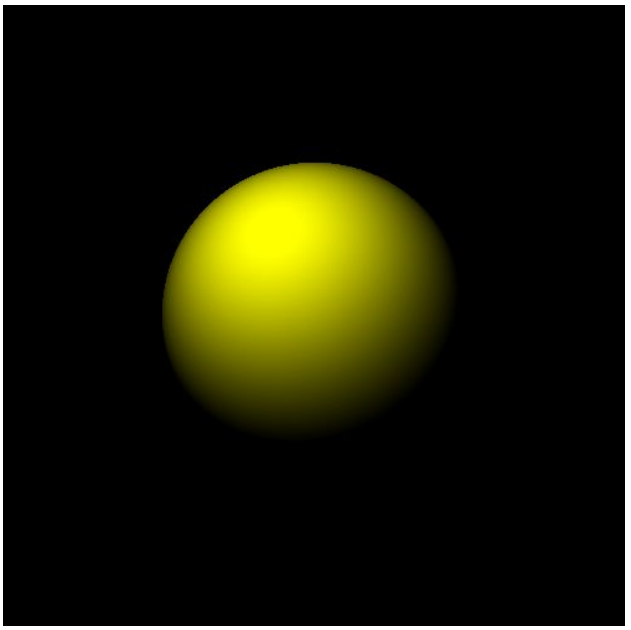
Toutes les théories scientifiques fascinantes ont commencé avec la forme la plus simple et évoluera avec le temps, c'est ainsi que notre projet sera développé.
Une scène sera représentée par une classe qui va être composée de plusieurs objets (sphères…). L'idée la plus naïve est d'envoyer un rayon vers un pixel de l'image, si le rayon intersecte un object dans son chemin, le pixel sera blanc.

En appliquant ce modèle simple pur une sphère, le code nous renvoie l'image suivante:

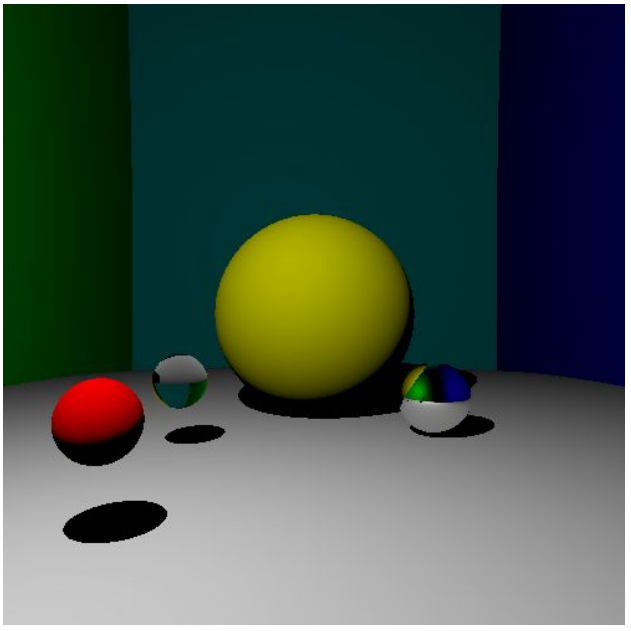


Cette image ne prend en compte que l'interaction directe entre le rayon et l'objet. Il n'y a aucune notion d'éclairage ou d'ombres qui rend l'image plus vive. C'est ce qu'on va implémenter par la suite.

On ajoute ainsi une source de lumière dans la scène en ne prenant que l'interaction rayon-cercle la plus proche de la caméra. Le résultat de cette implémentation:



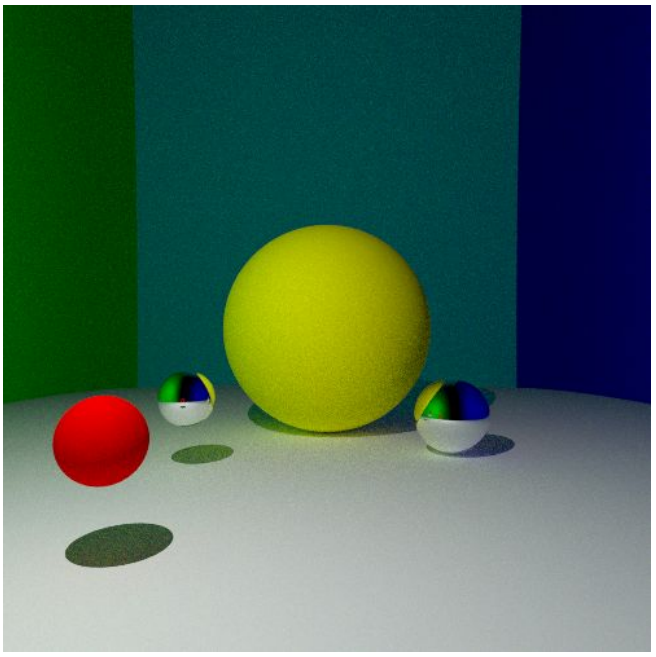
L'objectif du deuxième cours était d'implémenter les différentes interactions physiques comme la réflexion et la transparence ainsi que les ombres portées. En même temps, nous ajoutons un coefficient au calcul de la couleur du pixel appelé la correction Gamma qui influe globalement sur la qualité des couleurs de l'image comme l'on remarque dans l'image suivante:



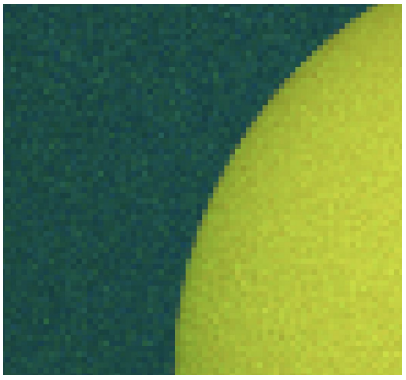
Pour générer l'image, on ne prend en compte que l'éclairage direct venant de la source lumineuse. En réalité, il existe également l'effet de l'éclairage indirect venant des

autres surfaces en plus des sources. C'est pourquoi nous ajoutons dans le troisième cours cet éclairage indirect qui rend l'image plus saturée en couleur par rapport au résultat précédent:

Résolution = 500 x 500 avec 50 rayons.



Même si l'image précédente semble meilleure, elle contient encore des erreurs à corriger. Nous précisons deux:



1- Crénelage: on observe que les contours des objets ne sont pas lisses en raison de l'envoi des rayons aux centres des pixels. La solution serait de faire un échantillonnage de rayons aléatoirement à l'intérieur de chaque pixel au lieu de les envoyer au centre.

2- Ombre: on remarque aussi que les ombre dans l'image précédente sont franches car il sont dues à des sources ponctuelles de lumières. Il serait plus réaliste si on intègre des sources étendues de lumière.

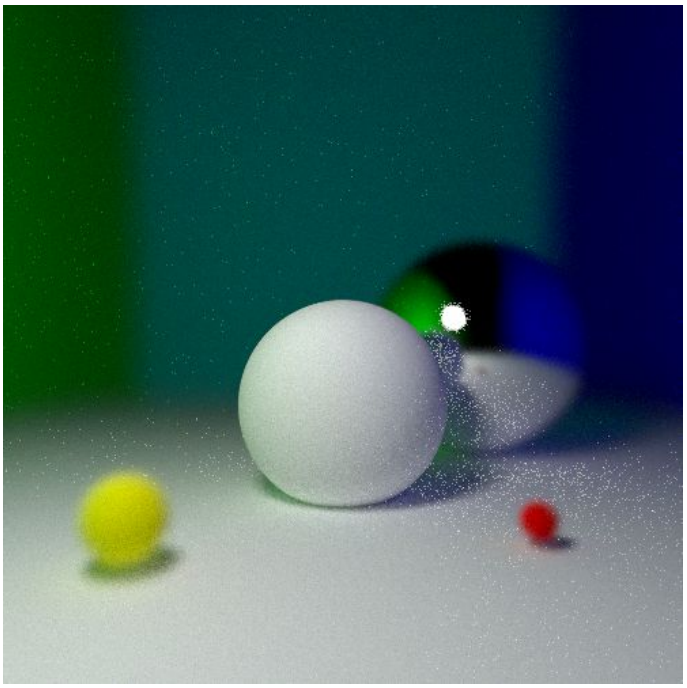
C'est la raison pour laquelle, nous essayerons d'affiner des ombres douces au lieu des ombres strictes qu'on avait auparavant et de corriger l'effet de crénelage. Cela donne plus de réalisme à notre scène comme l'on peut remarquer dans l'image suivante

générée par mon code:

Résolution = 500 x 500 avec 50 rayons.

Le nombre des rayons affecte largement la qualité de l'image. Un nombre plus bas de rayons implique plus de bruits dans l'image. En même temps, plus on augmente le nombre de rayons, plus l'exécution du programme devient plus lente.

Ensuite, nous ajoutons une nouvelle variable appelée distance de mise au point de l'appareil. L'idée vient du fait qu'on ajoute un élément (diaphragme) entre la lentille et l'image qui va bloquer la lumière sauf dans une partie. Ceci crée une sensation de profondeur de champ comme l'on peut voir dans l'image suivante:

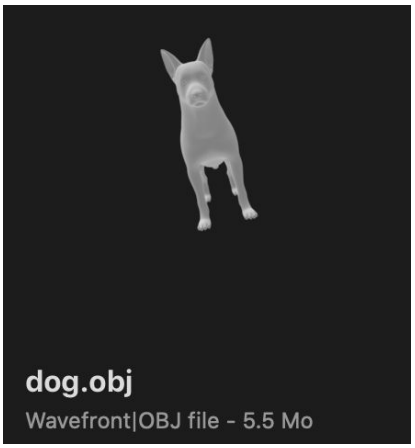


4. Maillages & Textures (Cours 5: Cours 6)

Dans cette partie, je commence à introduire les maillages dans mon code.

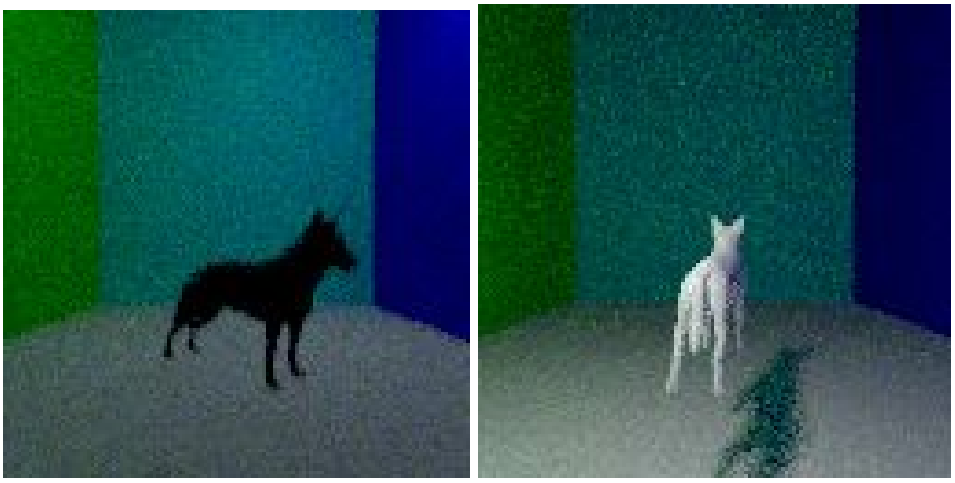
En effet, le maillage est un ensemble de plusieurs triangles, voire des milliers et des milliers. Le calcul normal d'intersection rayon-triangle prendra par conséquent un temps énorme. C'est la raison pour laquelle on va définir des méthodes plus efficaces. La première étant de considérer une suite de boîtes englobantes qui recouvrent la totalité du maillage. Nous n'allons prendre que les rayons qui intersectent la boîte. Si le rayon n'intersecte pas la boîte, il ne pourra pas, par conséquent, intersecter les triangles qui appartiennent à cette boîte. Cela permet de gagner du temps en évitant de calculer les interactions avec tous les triangles.

Le fichier OBJ qui contient les éléments du maillage est : "**dog.obj**" que j'ai trouvé en libre source sur le site https://free3d.com/:



Pour tester l'algorithme, le code est exécuté pour les paramètres suivants:

Résolution = 128 x 128 avec 10 rayons.



La différence entre les deux images est la position de la lumière ainsi que l'angle d'orientation du maillage (10° à droite et 120° à gauche). L'objectif était de simplement tester le comportement de l'algorithme vis à vis de ces changements de paramètres.

Tout semble bien! Allons plus loin maintenant en implémentant une autre méthode efficace pour le calcul des intersections. Il s'agit d'intégrer la recherche des triangles d'intersection par dichotomie. La recherche dichotomique est connue pour son efficacité et rapidité par rapport aux autres méthodes de recherche, surtout quand les éléments sont triés et distinguables. La rapidité de cette méthode par rapport à la méthode précédente nous autorise à augmenter le nombre de rayons et la résolution de l'image.

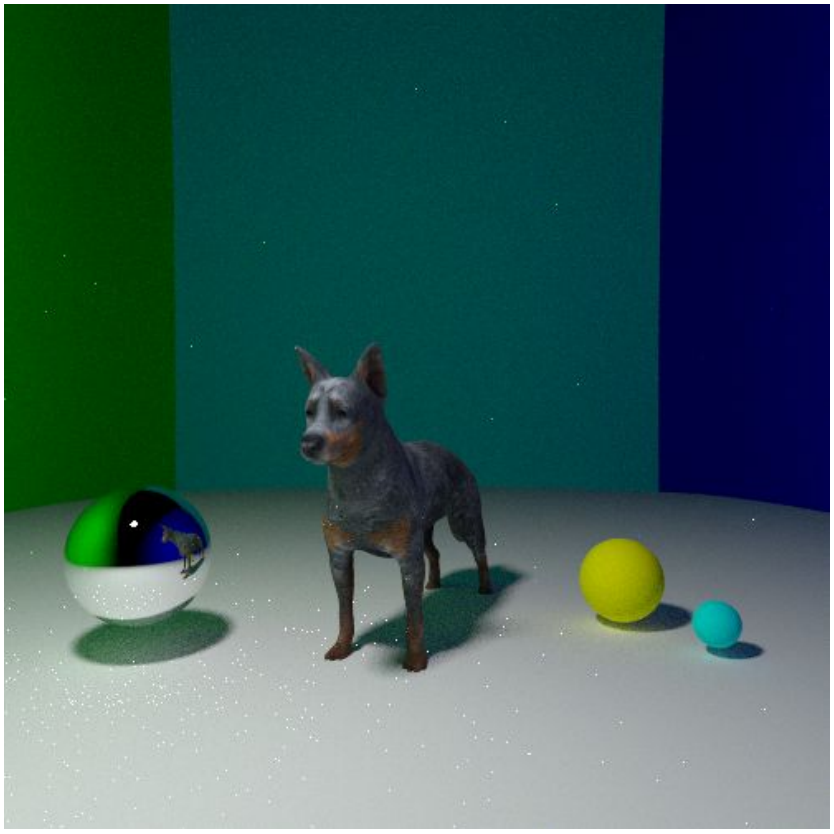
Résolution = 500 x 500 avec 100 rayons.

Pour la texture, j'utilise le fichier "**Australian_Cattle_Dog_dif.jpg**" qui est une image de la texture du chien. Notre mission dans cette étape est de créer une association entre le maillage et la texture.

Résolution = 500 x 500 avec 100 rayons.

5. Résultat final:

Résolution = 500 x 500 avec 200 rayons.



Les images sont de meilleure qualité dans le répertoire de Github.
A cause des problèmes techniques liés à la parallélisation du calcul sur Visual Studio Code, je n'arrivais pas à implémenter la dernière partie liée au changement de la caméra pour faire des vidéos de la scène. Je vais essayer de mettre à jour le répertoire sur Github une fois j'arrive à l'implémenter et à l'exécuter

Conclusion finale

Le cours d'Informatique Graphique était très enrichissant et une opportunité pour appréhender les différentes méthodes de génération et création des scènes par le Ray Tracing ainsi que les autres méthodes de rendu utilisées dans le domaine des jeux de vidéos par exemple. La compréhension de la partie théorique/mathématique des différents concepts était cruciale pour l'implémentation, intéressante et passionnante en même temps: de simples détails peuvent changer complètement la visualisation.

Ce cours était également une opportunité pour me familiariser avec le langage C++ qui m'était nouveau honnêtement. Mais comme je le dis souvent, une fois qu'on a les bases d'algorithmique, le problème revient à chercher et vérifier la syntaxe du langage et corriger les erreurs. Avec le temps, cela devient de plus en plus rapide mais peut être plus compliqué surtout quand on a l'habitude de coder avec Python.

En outre, j'ai rencontré des problèmes techniques avec mon MacBook Pro qui, apparemment, n'est pas fait pour exécuter ce type de calculs graphiques assez lents. Cela avait un effet bloquant pendant quelques séances de cours. Cependant, je pouvais compenser ce gap de retard pendant les jours suivants de la semaine.
Dans ce contexte, si j'étais invité à proposer une petite amélioration, j'aurais bien aimé avoir accès à des machines virtuelles ou des GPUs en cloud de l'école pour l'élaboration de ce projet, surtout quand la machine locale n'est pas aussi performante dans ces types de calcul même avec la parallélisation (qui ne fonctionnait pas correctement sur Visual Studio Code) et pourquoi pas avoir un cours d'initiation au C++ au début du cours pour les étudiants qui n'ont jamais manipulé ce langage.