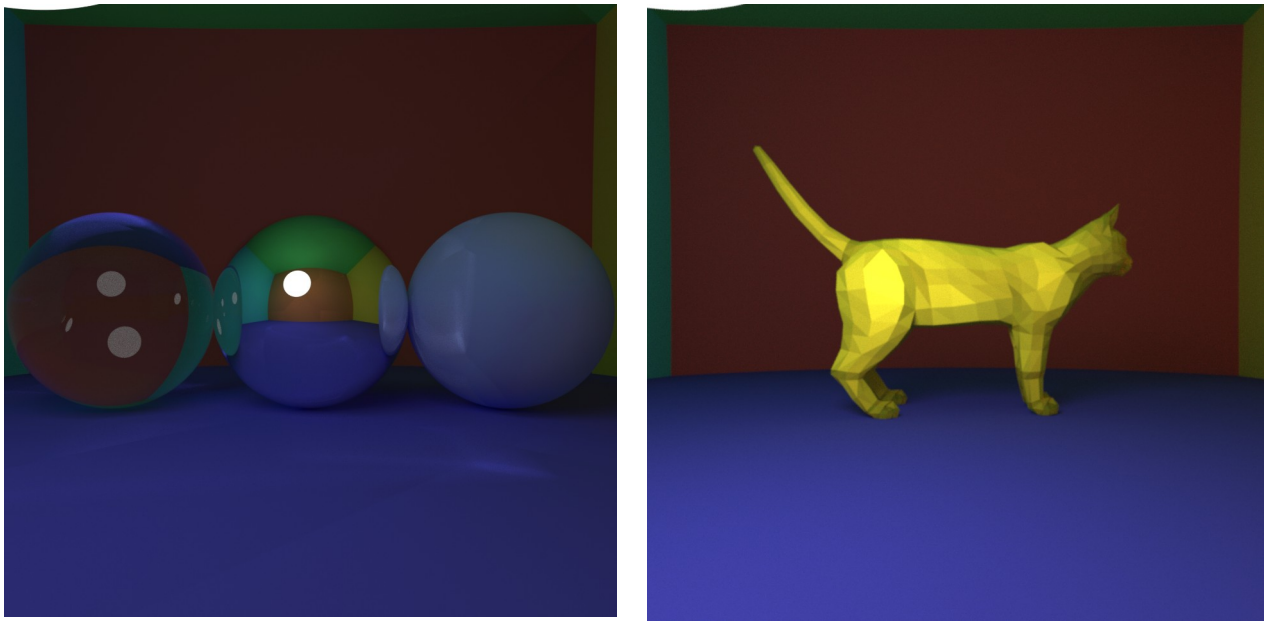
Guillaume Paczek
22/03/2024

Mos Informatique Graphique

Rapport de projet

Figure 1: Possibilités du ray tracer



Introduction

Le projet présenté est un « ray tracer » codé en c++ dans le cadre du cours d'informatique graphique enseigné en troisième année à l'école centrale de lyon. Un ray tracer permet de générer une image d'une scène (ensemble d'objets 3D) en tenant compte de différents phénomènes physiques : Sources de lumières, ombres, réflexions, effet optiques.

Ce rapport présente la démarche, les résultats et les pistes d'amélioration du projet

Fonctionnalités

Modèle lambertien

Après avoir défini les classes nécessaires à la manipulation de sphères, rayons de lumière et vecteurs, la première étape de notre ray tracer est de générer une image en lançant autant de rayons qu'il y a de pixels, depuis le centre de la scène, et de colorier le pixel seulement s'il intercepte un objet de la scène. On obtient la figure 2.

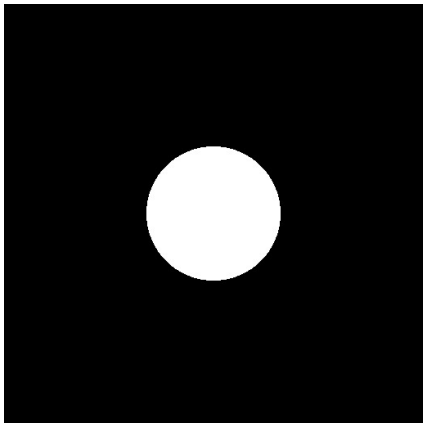


Figure 2: Intersection rayon / sphère

Pour la couleur et les ombres, on choisit d'implémenter un modèle lambertien de la diffusion de la lumière. Valable en théorie pour les diéléctriques, ce modèle stipule que l'intensité émise par un corps éclairé par une source ponctuelle est la même dans toutes les directions, et inversement proportionnelle à la distance à la source lumineuse . Cela permet d'obtenir une bonne base comme montré sur la figure. Notons qu'on a aussi appliqué une correction gamma pour réhausser les couleurs.

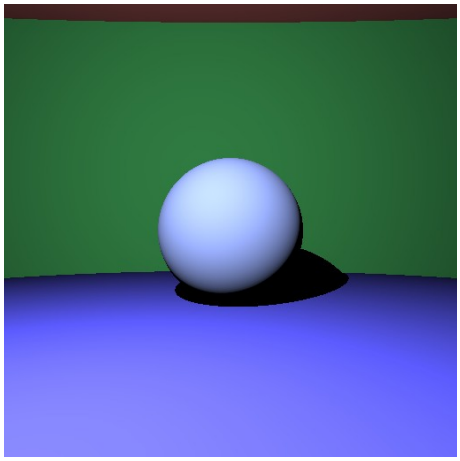
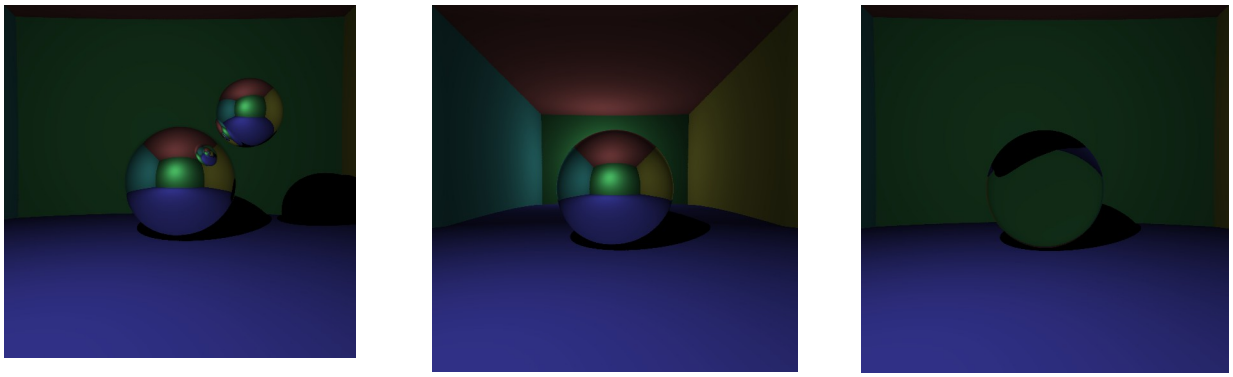


Figure 3: Modèle lambertien de la diffusion de la lumière

Réflexion / Réfractions

On souhaite désormais rendre possible l'ajout d'objet transparents ou de miroirs dans notre scène. En utilisant les lois de Snell-Descartes, on peut facilement calculer la direction du rayon transmis ou réfléchi. Ainsi, au lieu d'afficher la couleur du matériau, on relance un rayon et affiche cette intensité. Ceci nous impose de rendre notre fonction de calcul de couleur récursive. Par ailleurs, dans le cas de la réfraction il faudrait renvoyer un rayon réfléchi et un réfracté. Pour limiter le nombre d'appels récursifs, on envoie un rayon ou l'autre, selon les coefficients de Fresnel, qui donnent plus d'importance à la réflexion lorsque la différence d'indice est forte et que le rayon est proche de la normale. On obtient la figure 4

Figure 4: Réflexion et réfraction. A gauche : 2 miroirs. Au milieu : Le fond est un miroir. A droite : Une sphère transparente



Ombres douces / Eclairage indirect.

Deux gros ajouts à la qualité de l'image sont les ombres douces et l'éclairage indirect. L'éclairage indirect consiste à prendre en compte l'éclairage de la source lumineuse mais aussi l'éclairage dû à chaque point de la scène qui se comporte aussi comme une source lumineuse. En principe, on doit intégrer l'intensité lumineuse sur toute la scène. Mais en pratique, le calcul est simplifié grâce à l'approximation des intégrales par la méthode de Monte-Carlo : Pour chaque rayon, on calcule la composante directe en fonction de la distance à la source lumineuse, et la composante indirecte en envoyant un rayon dans une direction aléatoire Il ne reste qu'à faire la moyenne des résultats obtenus, et on peut espérer qu'en envoyant suffisamment de rayons, une majeure partie des directions de l'éclairage indirect ait été couverte.

Les ombres douces ajoutent une autre dimension au problème : Au lieu de considérer des sources ponctuelles, on considère des sources de lumière sphériques. Ainsi, la composante directe de l'éclairage se calcule aussi en principe avec une intégrale, que l'on approxime en pratique grâce à Monte-Carlo encore une fois, en tirant un point au hasard sur la source lumineuse.

Une fois ces deux effets implémentés, le rendu paraît beaucoup plus naturel : La pénombre apparaît et les objets prennent légèrement les couleurs de leurs voisins. Cf figure 5

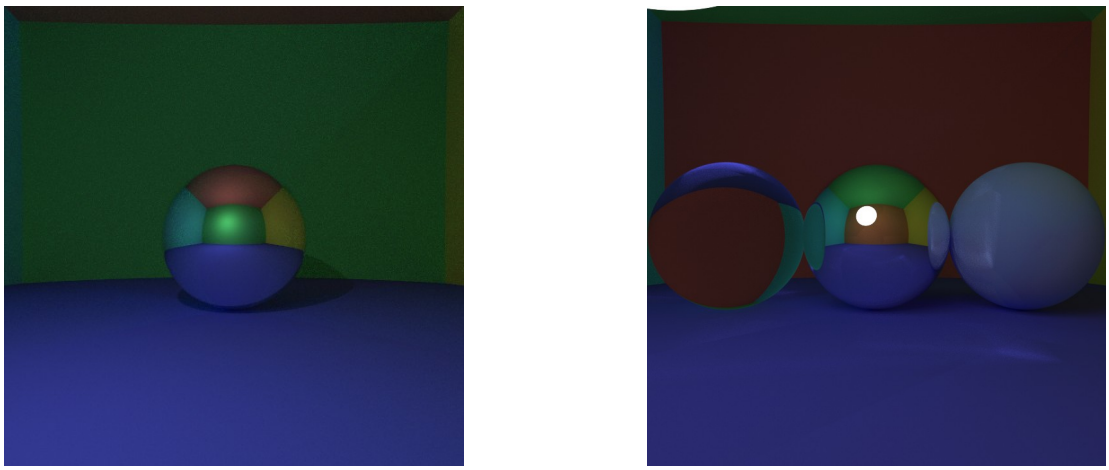


Figure 5: Ajout de l'éclairage indirect (gauche) puis des ombres douces (droite)

Maillages

Au delà des sphères, on aimerait pouvoir mettre n’importe quelle forme dans la scène. Les maillages semblent être appropriés : On peut construire plus ou moins n’importe quel objet à partir de triangles, et les formules d’intersection rayon/plan et rayon/triangles sont simples et rapides à calculer. Le problème à partir de cette étape du projet est le temps d’éxécution. En effet, les formules ont beau être simples, l’intersection rayon/scène ne faisait intervenir qu’une dizaine de formules rayon/sphère. En rajoutant le maillage d’un chat de 4000 triangles, le nombre d’objets à tester est ainsi décuplé. On implémente donc l’algorithme BVH (bounding volume hierarchies) qui calcule récursivement des bounding box, en divisant en deux la taille de chacune, créant ainsi un arbre de bounding box. Pour tester si un rayon intersecte le maillage, il suffit de ne visiter que les nœuds intersectés par le rayon, ce qui fait effondrer le nombre de calculs à faire.

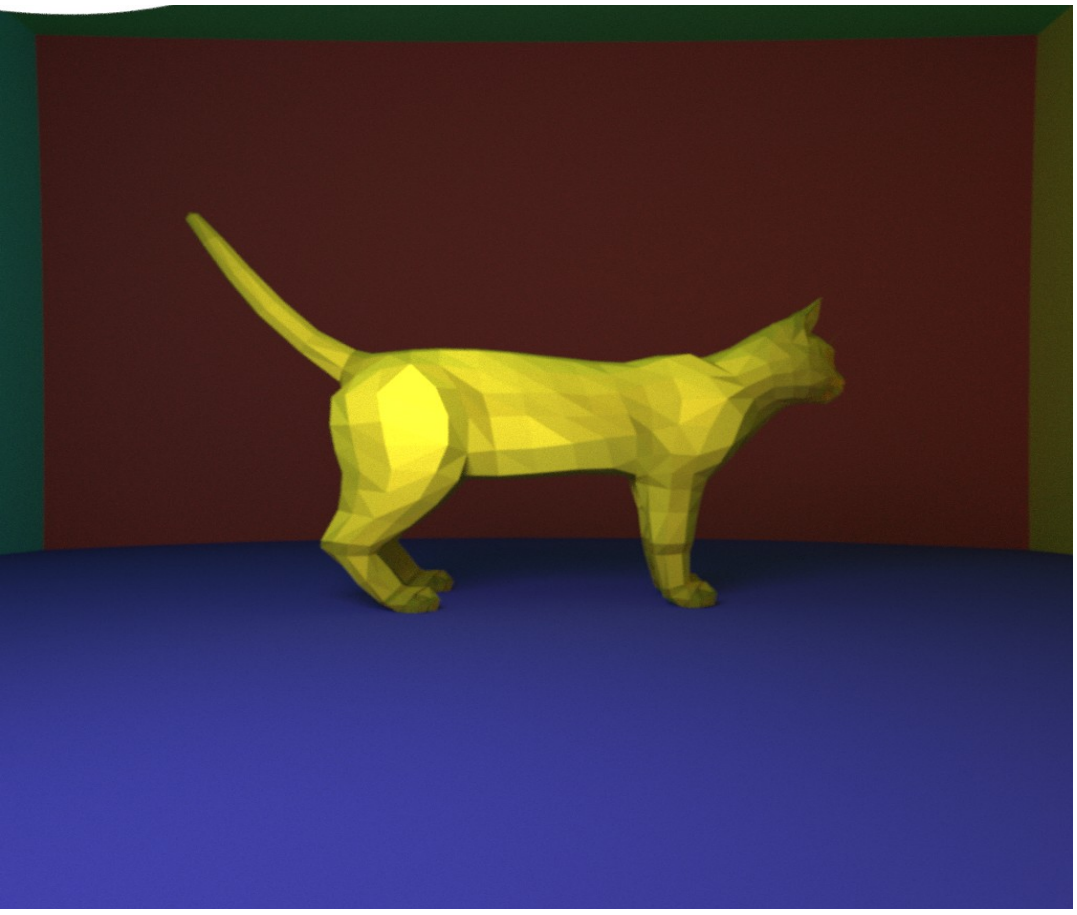


Figure 6: Ajout des maillages dans la scène

Enfin, on peut s’amuser à mélanger tous les éléments implémentés dans une seule image :

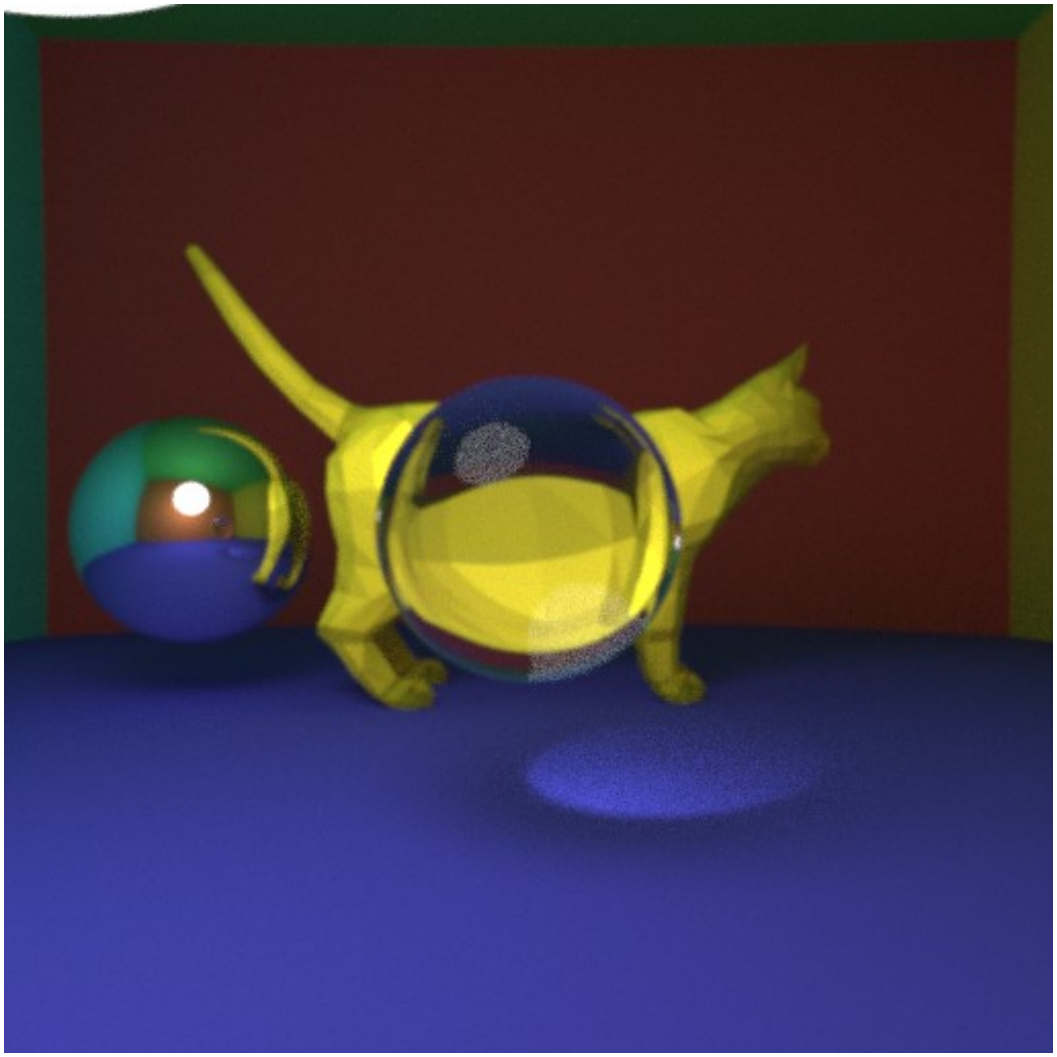


Figure 7: Image contenant un maillage, une sphère transparente, une sphère miroir

Difficultés rencontrées

Temps d’exécution

Avant l’implémentation des maillages, je n’ai pas eu de difficultés particulières. Mon code tournait avec un ordre de grandeur similaire à celui du poly. En revanche, j’ai eu un problème qui m’a longtemps bloqué lors de l’implémentation du parcours en profondeur et que je n’ai à ce jour pas résolu : Il n’y a aucune nette amélioration suite à l’implémentation de ce dernier !!! Je ne vois pas d’où peut venir le problème : J’ai vérifié si mon algorithme BVH était correct, et l’arbre se génère correctement, j’ai aussi comparé mon code à celui du poly et le parcours en profondeur ne me semble pas si différent, et pourtant on a aucune amélioration de performance, ce qui donne un code particulièrement lent…

Voici les temps d’exécutions que j’ai pu relever :

time	feature name	resolution	rays	bounces
3min	soft shadows	1024	128	5
55min	simple bbox	1024	128	5
68min	bvh	1024	128	5

Aides extérieures

Ayant manqué quelques séances, mon camarade Baptiste Perreyon m’a aidé à rattraper et m’a fourni son code pour le calcul de minimum et maximum dans une liste de vecteurs, ainsi que les formules d’intersection rayon/plan

Je me suis par ailleurs inspiré du poly pour l’algorithme BVH ainsi que le parcours de l’arbre en profondeur.

Avis sur le cours

- Excellent, merci beaucoup, c’était vraiment très intéressant, j’avais envie d’aller en cours chaque vendredi. Technique comme il fallait et très amusant de voir la progression cours après cours. J’ai peur que ça soit compliqué pour ceux qui ne maîtrisent pas le c++, mais je n’ai pas eu ce problème car j’avais quelques bases.

- Si vous avez une idée pour mon problème de lenteur de programme, je serais très curieux d’avoir des éléments de réponse (guillaume.paczek@ecl20.ec-lyon.fr)

Merci ! Et bonne continuation