Daniel Douheret

MOD rendu 3D

20/02/2020

ÉCOLE CENTRALE LYON

Rapport MOS – Informatique Graphique

Rendu par Ray Tracing

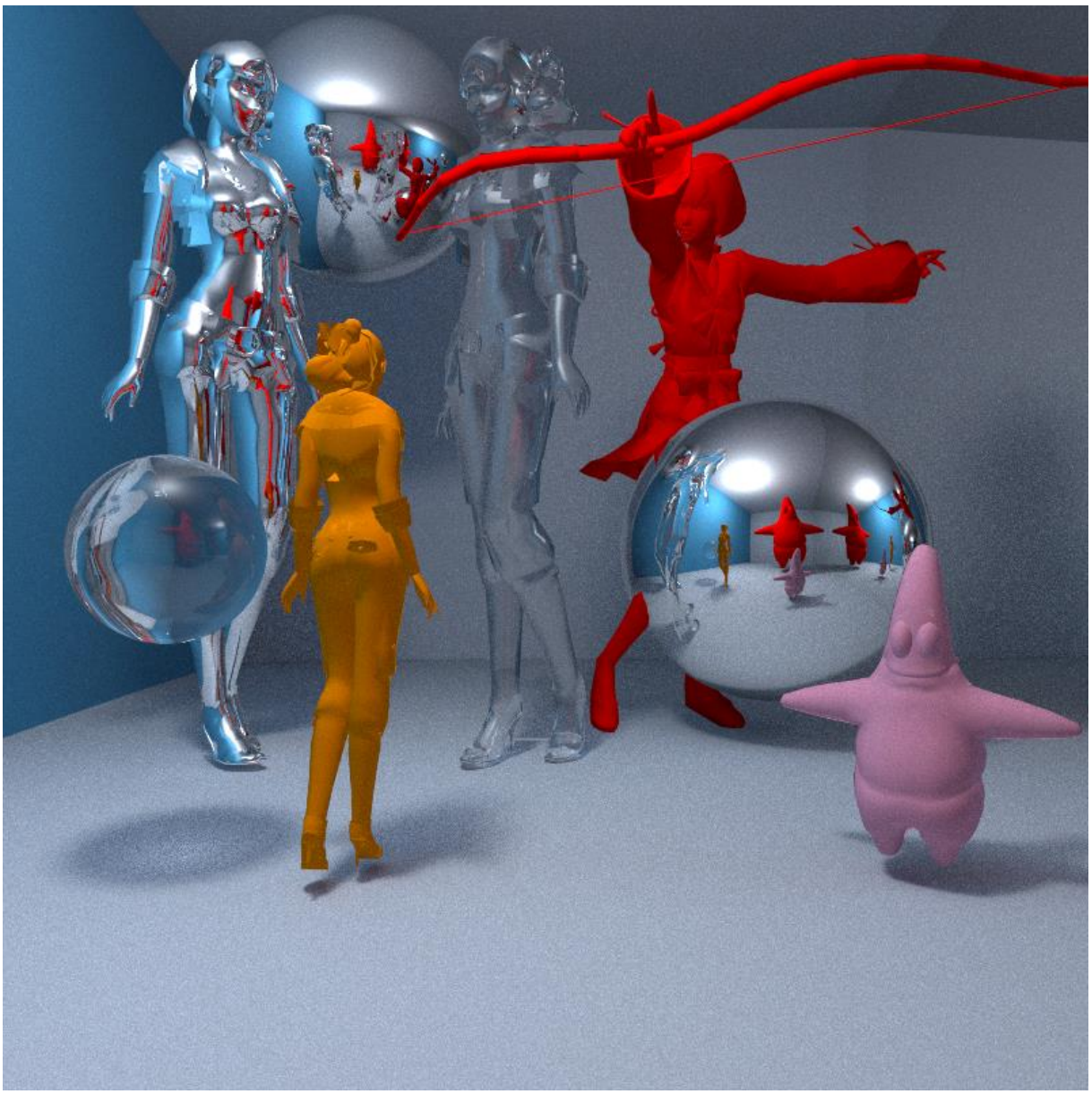


Table des matières

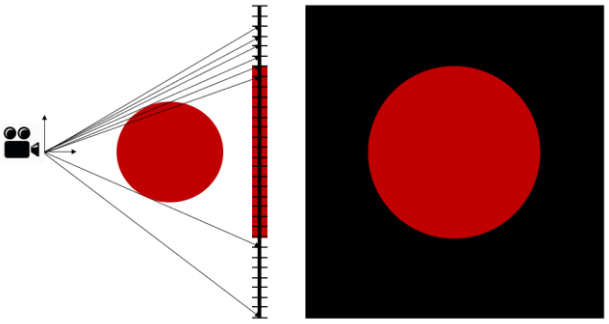
Introduction

Le lancer de rayon (Ray Tracing) est une méthode de rendu 3D permettant d'obtenir des scènes extrêmement réalistes. Cette méthode étant néanmoins très gourmande en complexité algorithmique, elle est surtout utilisée en cinématographie et photographie. La méthode de base date de 1980 avec le Raytracer de Whitted.

Modèle de base

L'idée de base est de se placer en tant que caméra devant une grille. Cette grille représente les pixels de l'image, initialement noirs. Chaque pixel sera illuminé seulement si un objet se trouve entre nous (la caméra) et ce pixel. Pour le savoir, il faut lancer un « rayon ». Un rayon n'est qu'une droite, que l'on définit avec une origine et une direction.

Sur le schéma 2D ci-dessous, on imagine une seule sphère rouge dans l'espace 3D. Chaque pixel de la grille sur lequel un rayon aura intersecté la sphère sera coloré en rouge. On obtient l'image avec un simple fond rouge sur fond noir.

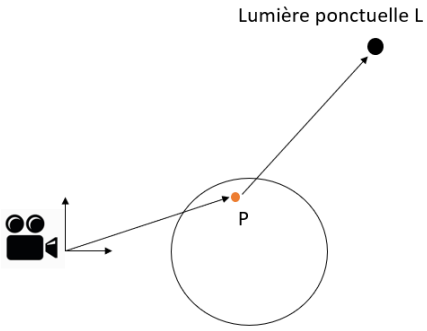


Le calcul de l'intersection se fait en résolvant l'équation de la sphère et l'équation de la droite représentant le rayon.

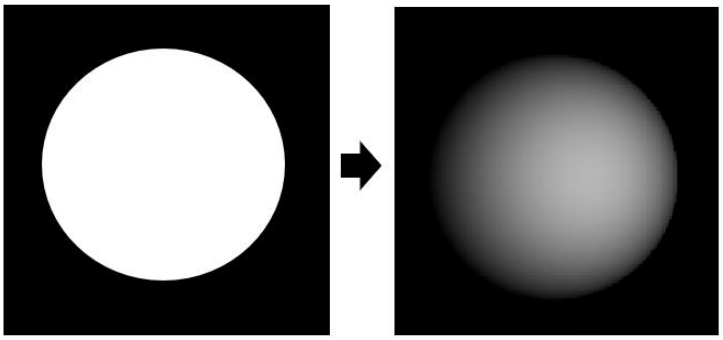
Dans un premier temps, nous n'afficherons que des sphères, car les méthodes d'intersection entre une droite et une sphère est assez simple.

Lumière ponctuelle

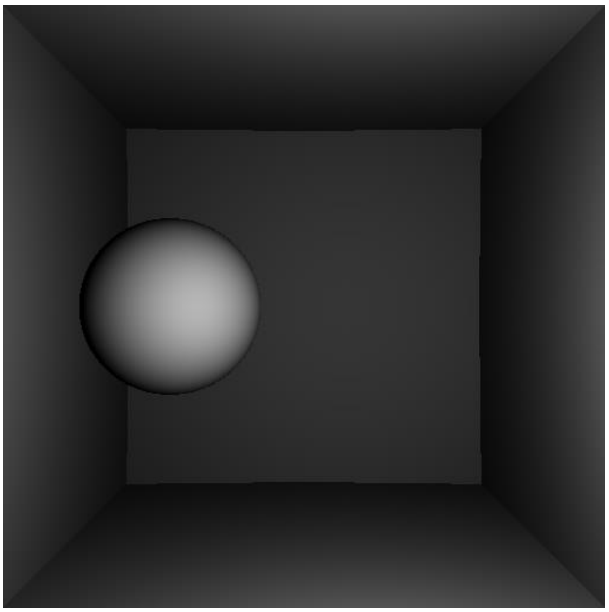
Pour le moment, chaque point de la sphère est illuminé avec la même intensité. Pour obtenir une meilleure représentation de la lumière, l'intensité de la teinte des pixels va être modulée. L'idée est de créer une lumière ponctuelle dans l'espace 3D. Pour chaque point d'intersection P, on lancera un second rayon en direction de la lumière. La distance $d = PL$ va permettre de moduler l'intensité sur une lois carré inverse.

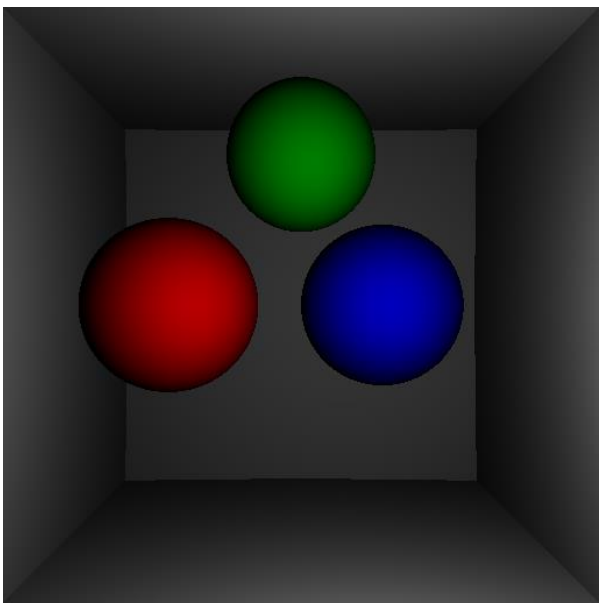


Pour une seule sphère blanche, on obtient la scène suivante



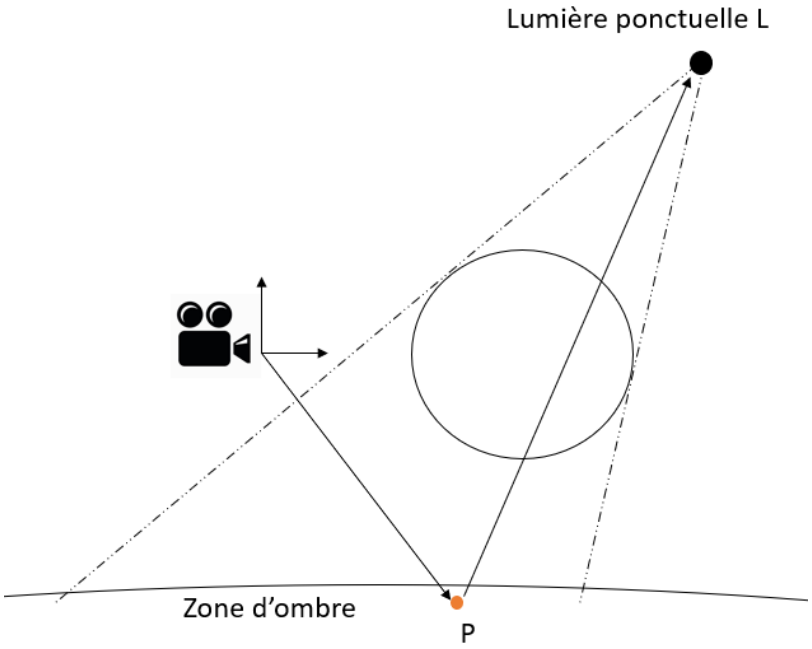
On va pouvoir insérer des murs dans la pièce en disposant de très grandes sphères sur les côtés et le fond.



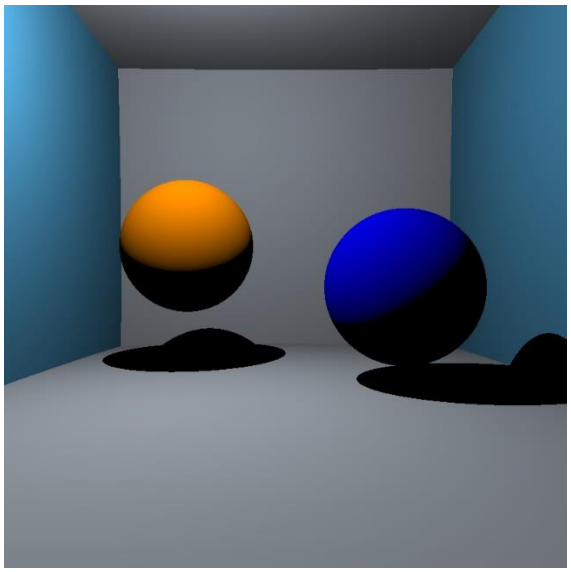


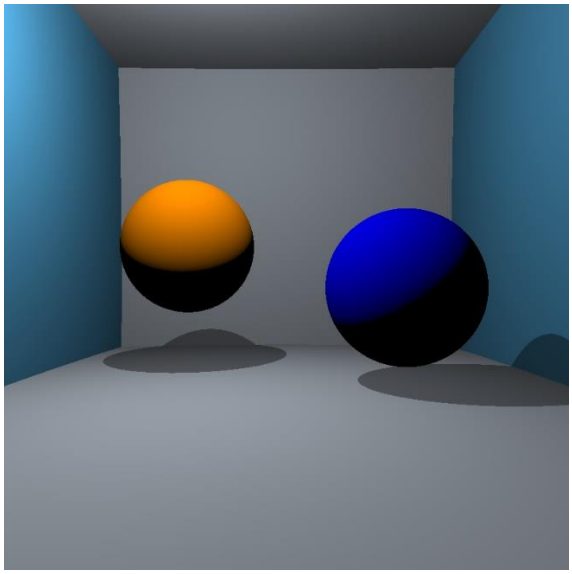
Ombres brutes

Pour visualiser l'ombre des sphères, à chaque point d'intersection P, on vérifiera qu'aucune autre sphère ne se trouve entre la lumière et le point P. Pour cela, il suffit de relancer un second rayon depuis P (*shadow_ray*). Si ce second rayon intersecte un objet qui se situe entre P et la lumière, on renvoie un pixel noir.



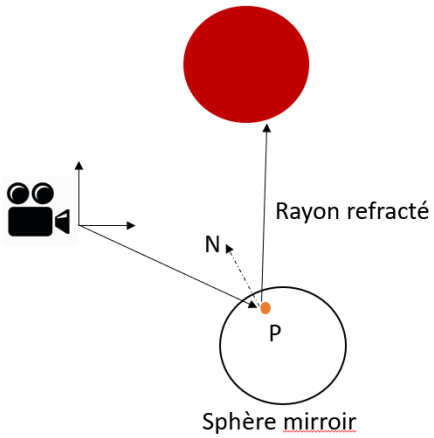
Au lieu de renvoyer un pixel noir, on peut choisir de renvoyer simplement l'intensité qu'aurait reçu le pixel, mais avec un facteur de reduction (0.3 par exemple).

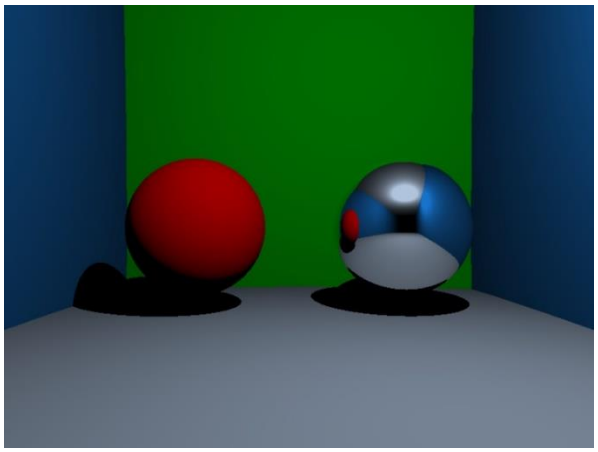




Surfaces spéculaires

Pour simuler un effet miroir, le mécanisme est assez simple. A chaque point d'intersection P, si la surface est spéculaire, on calculera la direction d'un nouveau rayon réfléchi grâce à la normale de la sphère au point P.

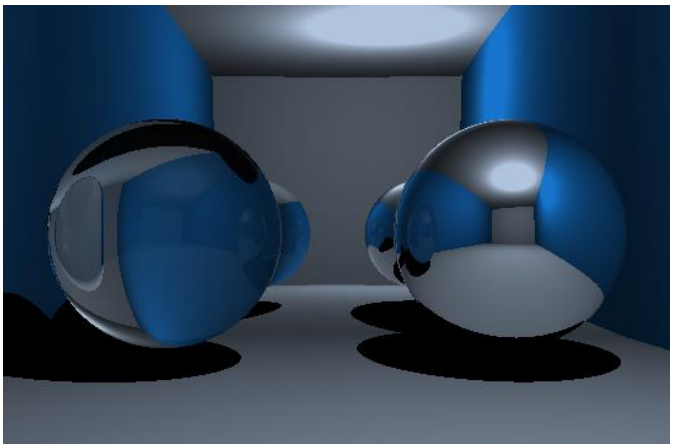


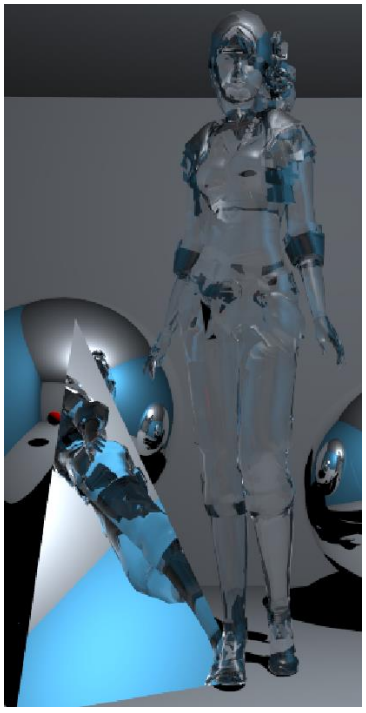


Pour éviter un nombre infini de rebonds (mise en abîme), on donne un paramètre *max_recursion* (=5 à 10) à notre fonction principale d'intersection.

Surfaces transparentes

De la même manière que le rayon réfracté, on va pouvoir calculer le rayon réfracté des surfaces transparentes à l'aide de la loi de Descartes n1.sin(a1) = n2.sin(a2). On ajoute de la réflexion pour plus de réalisme en lançant deux rayons : un réfracté et un réfléchis, avec des coefficients respectifs de 0.9 et 0.1.



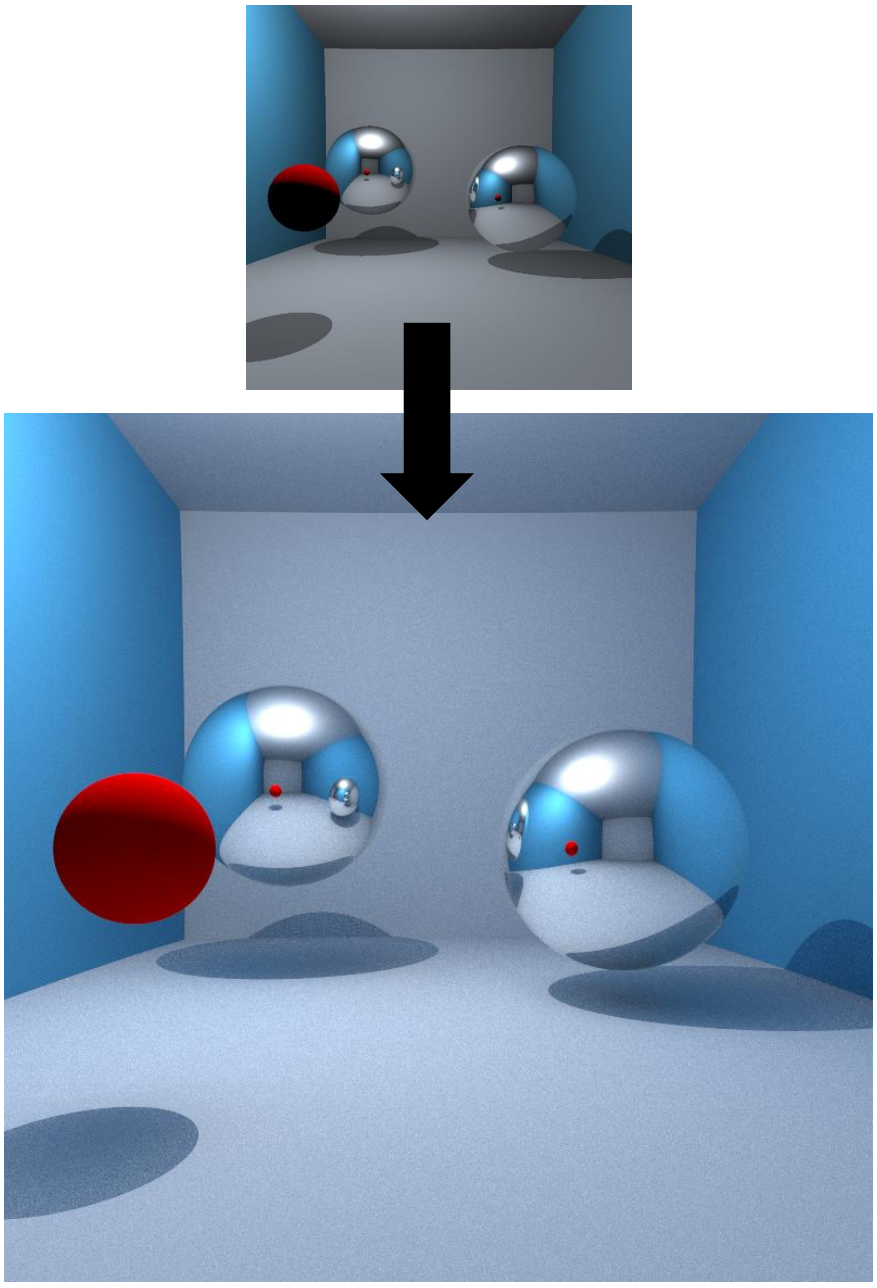


Eclairage indirect

Jusque-là, les objets présents dans la scène sont illuminés seulement par les sources de lumière : un mur blanc ne reflète pas la lumière qu'il reçoit et les ombres sont complètement noires.

Pour obtenir un résultat plus réaliste, il faudrait que la lumière rebondisse sur les objets. Pour cela, à chaque intersection, on relance un rayon dans une direction aléatoire dans l'hémisphère orienté par la normale de l'objet au point d'intersection. Un point aura donc deux contributions de lumière : la source de lumière directe, ainsi que la contribution indirecte de direction aléatoire.

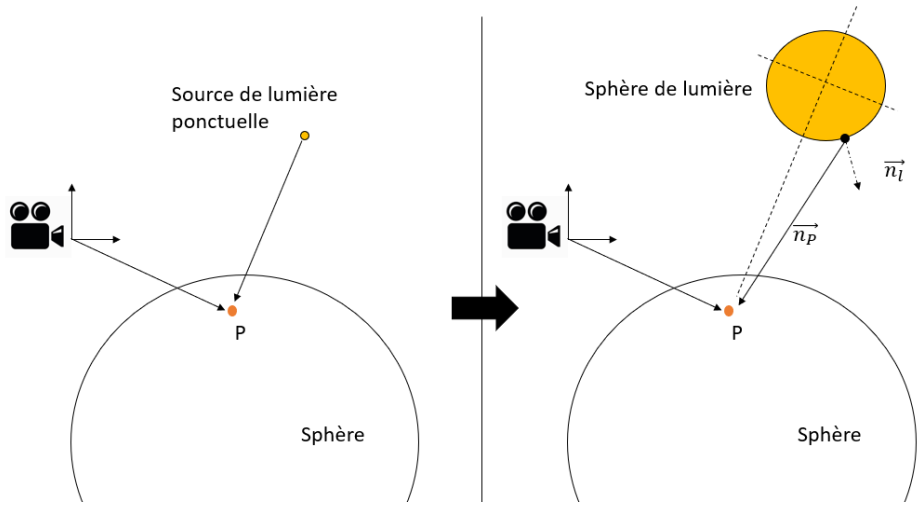
Cependant, pour converger vers un résultat parfait, il conviendrait de lancer le rayon dans tout l'hémisphère concerné. Il faut donc relancer plusieurs rayons par pixel pour approximer l'intégrale.



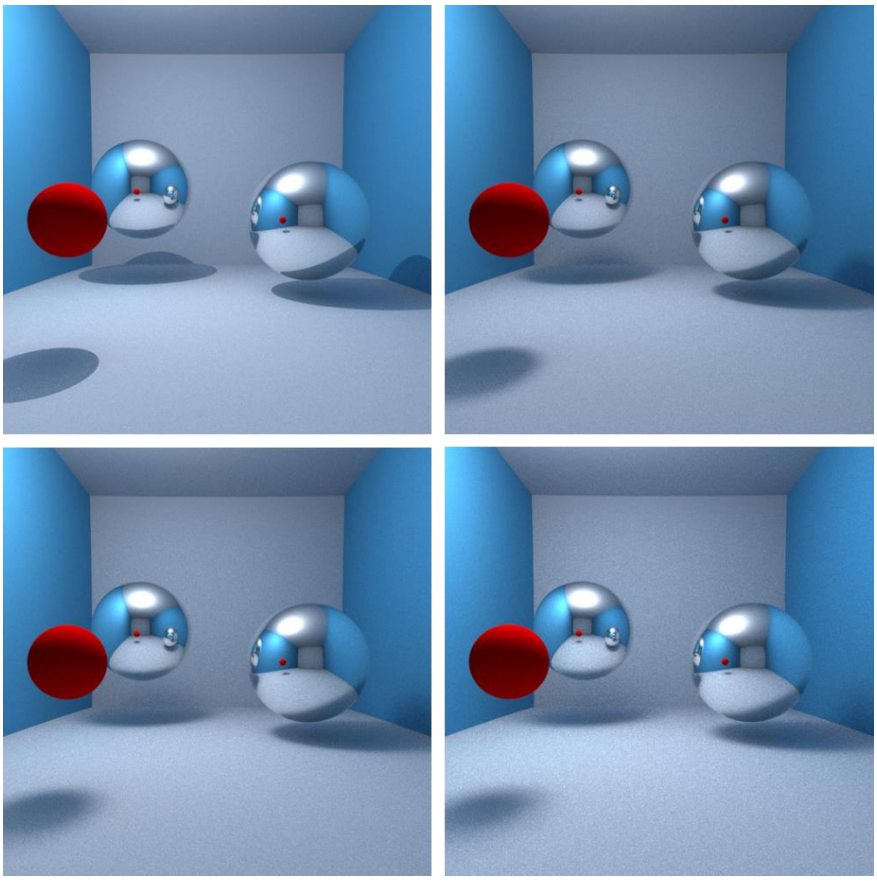
L'image ci-dessus a été générée avec 100 rayons/pixel.

Ombres douces

Jusqu'ici, les ombres sont brutes, il n'y a pas d'intensité graduelle. Pour donner un coté ombre douce, on va considérer la source de lumière comme une sphère lumineuse.



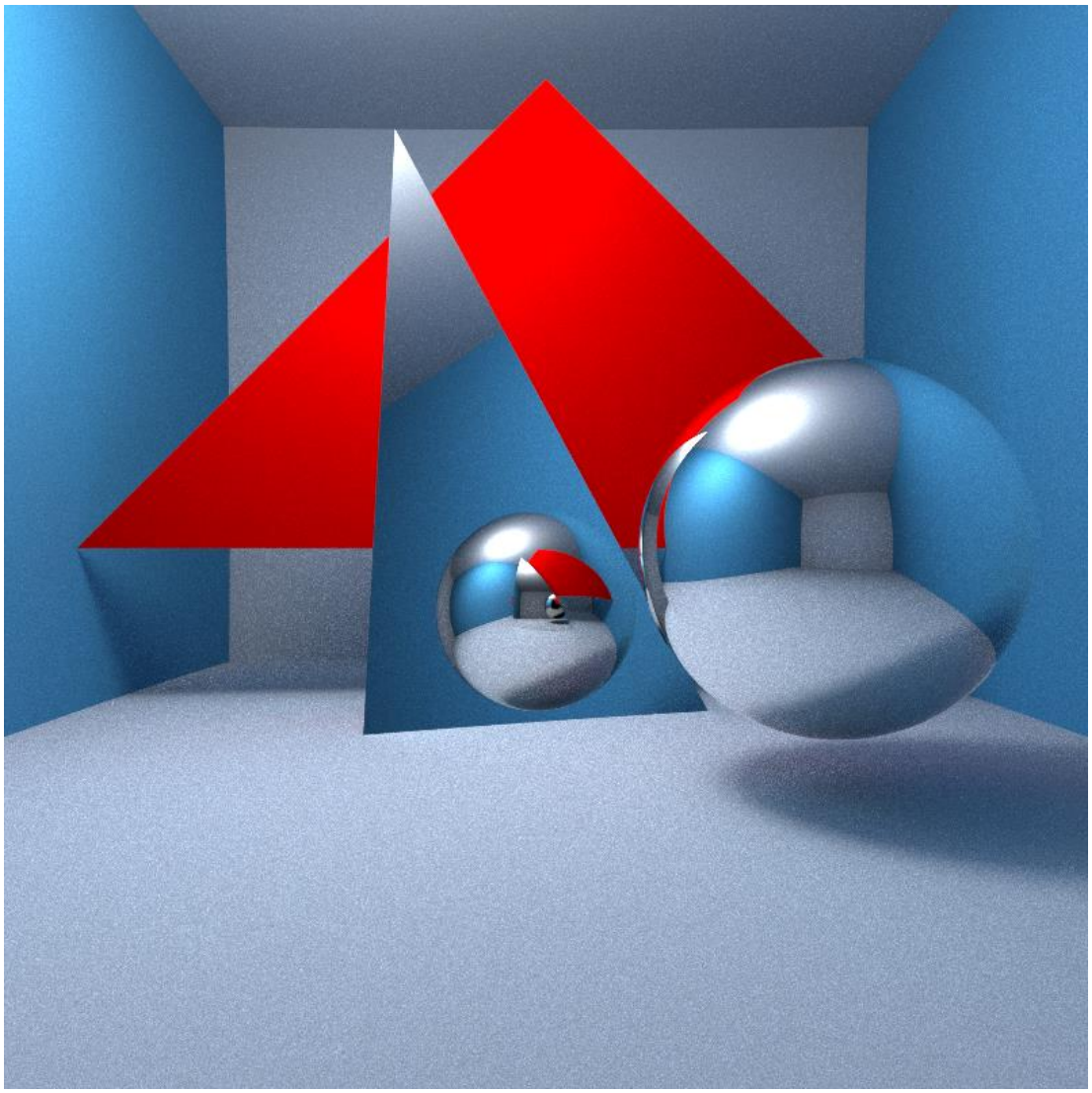
Au moment de calculer la contribution de lumière directe, on va déterminer un point aléatoire de la sphère de lumière dans son hémisphère dirigé vers P. C'est de ce point que le point P recevra son taux de lumière. L'intensité reçu sera modulée par le produit scalaire $\overrightarrow{n_p}.\overrightarrow{n_l}$.



Triangles

On souhaite maintenant ajouter d'autres primitives que les sphères à notre scène. Pour cela il suffit de créer une classe Triangle similaire à la classe Sphère. Sa méthode *intersect* consiste à vérifier que le rayon intersecte le plan contenant le triangle, puis que le point d'intersection P se trouve effectivement à l'intérieur du triangle grâce aux coordonnées barycentriques.

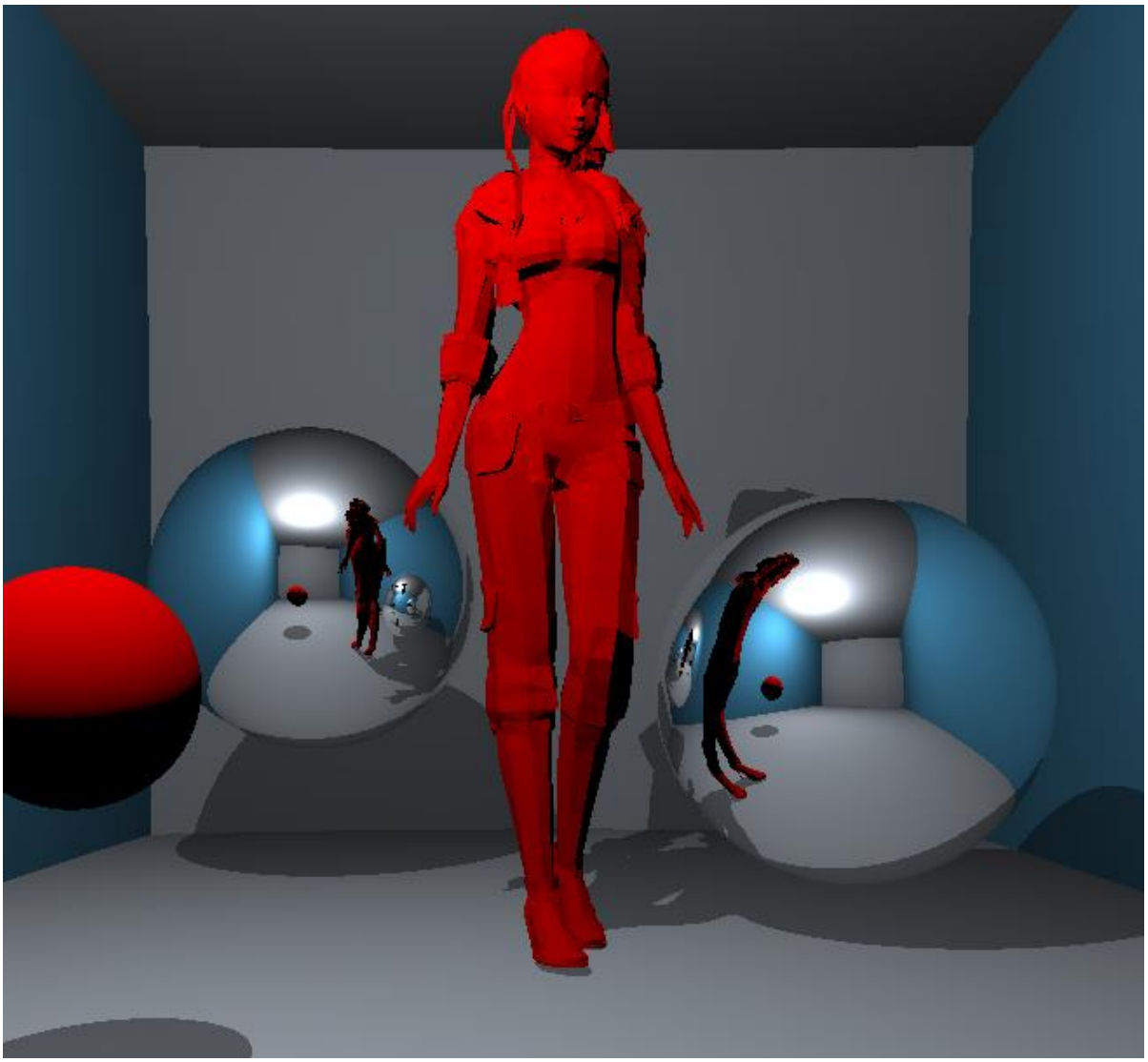
Pour rendre le code plus structuré, on crée une classe *Object* avec une méthode virtuelle *intersect*, dont héritera nos classes *Sphere* et *Triangle*.



Mesh

Un mesh est un ensemble de Triangle. On va donc créer une classe *Mesh*, héritant encore une fois de la classe *Object*, et avec pour attribut une liste de *Triangle.* Sa méthode *intersect* consiste à appeler la méthode *intersect* de chaque triangle et à sélectionner le plus proche s'il existe.

J'utilise ici des fichiers.off, dont le loader est très simple à mettre en place. Chaque mesh peut être positionné avec un offset, le scale et la rotation souhaités.



Structure d'accélération

Le code devient très lent à l'execution, puisque pour chaque pixel, on parcourt désormais des milliers de Triangle en plus.

Une première structure d'accélération est de créer une boîte englobante au mesh, *BBox*. Ainsi, avant de parcourir chaque triangle, on vérifiera d'abord que le rayon intersecte bien la boîte englobant le mesh.

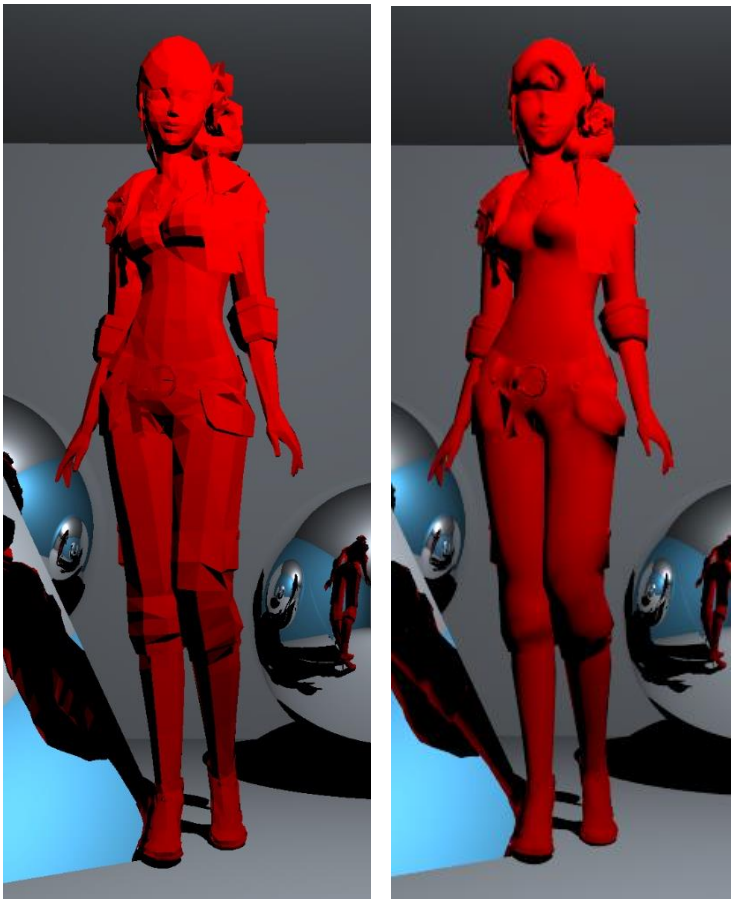
Cette méthode permet déjà d'obtenir un gain significatif. On peut cependant faire encore mieux en implémentant une strucutre BVH . La méthode consite à subdiviser notre mesh en des sous bôites englobantes. Une fois la *Bbox* initiale intersectée avec un rayon, on vérifiera laquelle de ces deux sous boites intersecte le rayon. Cette fois-ci le gain de temps est conséquent.

Interpolation des normales

Jusque-là, les triangles étaient bien visibles sur les mesh. Pour obtenir des surfaces bien lisses, il va falloir interpoler les normales aux points d'intersection avec les normales des vertex associés à chaque triangle.

Les normales des vertex sont calculées en sommant les normales des faces incidentes, puis sont normalisées. Néanmoins, il faut bien prêter attention à certains vertex sur les bords dont la normale est nulle après ce calcul ; dans ce cas-là, la normale d'un de leurs triangles incidents leur est attribuée.

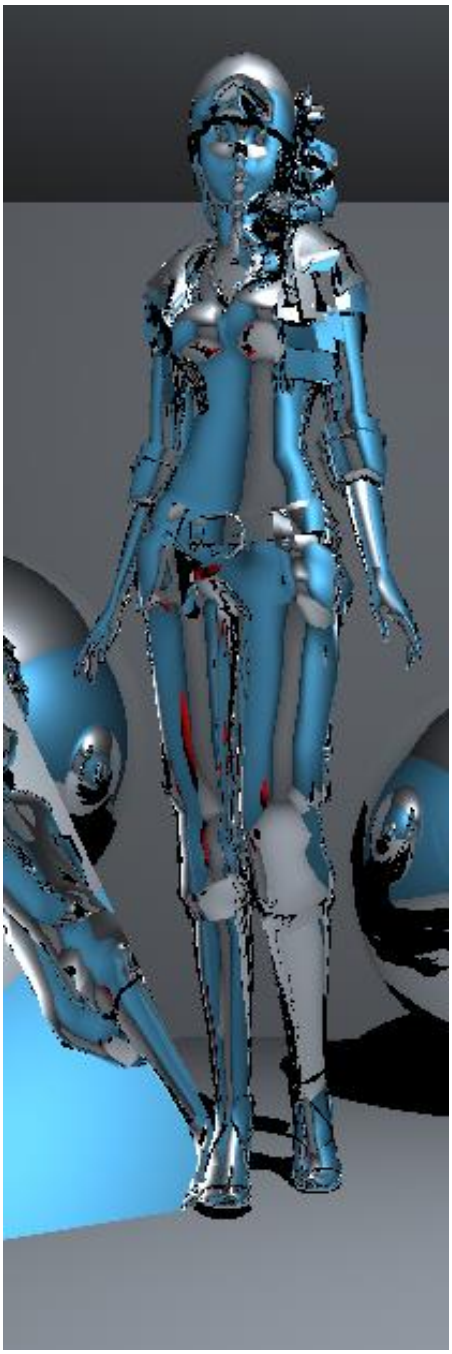
Ci-dessous, l'image sans contribution indirecte, avec interpolation des normales des vertex à droite.

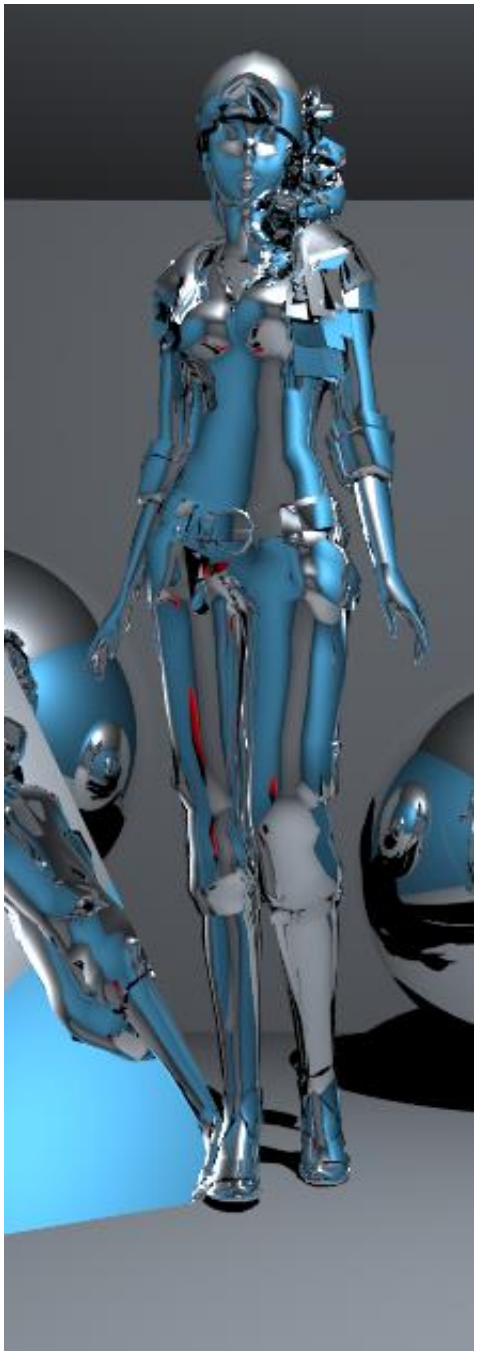


Anti-Aliasing

Pour remédier à l'effet crénelage sur les bords de surface, nous allons mettre en place un effet anticrénelage = anti-aliasing. Jusqu'ici, nous lancions les rayons au centre de chaque pixel. Pour gagner en précision, il convient d'envoyer les rayons dans une zone plus large, centrée sur le pixel, et de moyenner le résultat final.

Ci-dessous, l'image sans lumière indirecte, avec l'anti-aliasing avec 100 rayons/pixel à droite.

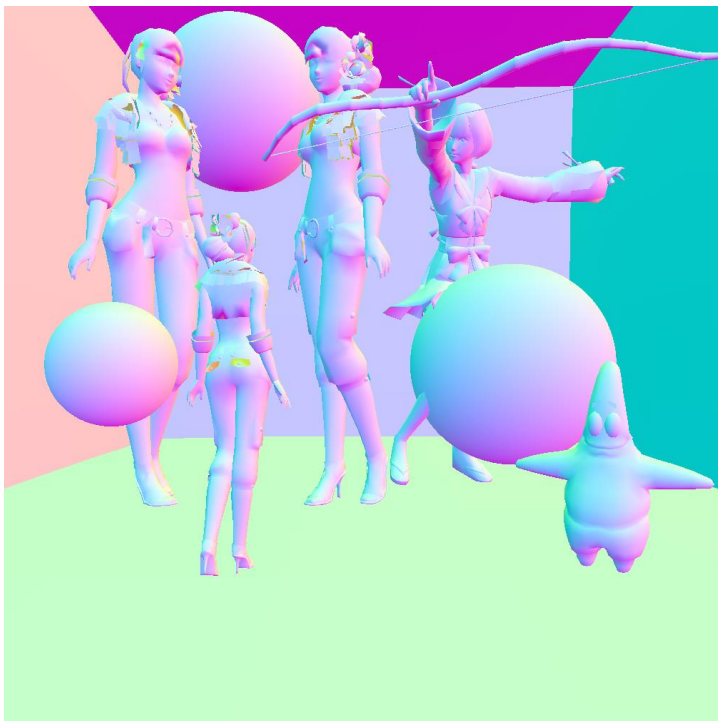




Debug

Un moyen simple qui m'a aidé à débuguer certaines scènes est de colorer les pixels en fonction des normales aux point d'intersections. Le résultat visuel est intéressant aussi.

L'image ci-dessus m'a permis de comprendre que mon calcul des normales de certains vertex était problématique puisque la norme de certaines est nulle (notamment sur la ceinture du mesh).



Commentaires

Le code et les fichiers 3D utilisés se trouve sur Github : https://github.com/Daniel-Dht/ray_tracing

Je suis ravi d'avoir pu participer à ce cours, car je ne me serais peut-être pas motivé à me lancer seul dans le code de raytracing, et ça aurait été bien dommage.

J'avais peu d'expérience en C++, et j'ai beaucoup appris, même si je dirais que j'ai passé le plus clair de mon temps à débuguer.

Peut-être faudrait-il juste envoyer en amont aux élèves un mail leur demandant d'installer un environnement de code stable et s'y familiariser avant le début des cours.