

ÉCOLE  
**CENTRALE**LYON

# ÉCOLE CENTRALE LYON

## COMPTE-RENDU DE BE

### MOS 2.2 - Informatique graphique

*Groupe :*  
Oriane CHARBONNEL

*Encadrant :*  
Nicolas BONNEEL

## Contents

1	Introduction	2
2	Premiers pas	2
3	Ombres, miroir et transparence	3
4	Eclairage indirect	4
5	Ombres douces	5
6	Profondeur de champs	7
7	Maillage triangle	7
8	Commentaires	10

## 1 Introduction

Ce BE a pour objectifs d'introduire les bases du raytracing et de les appliquer à des situations simples, en utilisant des sphères, plans... Le raytracing consiste à simuler le comportement de la lumière. Mais si on simulait simplement le trajet d'un rayon d'une source de lumière vers une caméra, seulement un petit nombre de rayons parviendraient au niveau de la caméra. Nous allons donc plutôt simuler le trajet du rayon inverse : de la caméra à la source de lumière. Pour cela nous utilisons le principe de réciprocité d'Helmholtz, qui nous dit que nous obtiendrons le même résultat avec cette méthode. Le raytracing va donc consister à envoyer des rayons de lumière, modélisés par des demi-droites, depuis la caméra et vers chaque pixel de l'image.

## 2 Premiers pas

Pour pouvoir faire du raytracing, il faut dans un premier temps générer un rayon et voir si ce rayon intersecte la sphère placée dans notre scène. On enverra ensuite un rayon pour chaque pixel et on regardera si il y a intersection avec la sphère.

Pour implémenter le raytracing en C++, il nous faut plusieurs classes :

- Une classe Vector qui permet de manipuler les données 3D facilement. Elle contient les coordonnées x, y et z. Elle contient également une surcharge des opérateurs classiques pour les vecteurs : addition, multiplication, division, soustraction, et des fonctions pour les produits scalaires, l'obtention de la norme (au carré) et la normalisation d'un vecteur.
- Une classe Sphere, définie dans un premier temps par son origine O et son rayon R. Elle contient une fonction renvoyant si la sphère intersecte un rayon et calculant son point d'intersection avec ce rayon et la normale à la sphère passant par ce point.
- Une classe Ray, définie par son centre C et son vecteur directeur  $\vec{u}$

Ces classes seront complétées/modifiées au cours du BE, et d'autres classes pourront être ajoutées.

Pour pouvoir utiliser ce raytracer, il nous faut une fonction principale `main()`. Dans cette fonction `main`, nous définissons la hauteur et la largeur de l'image, les positions des centre de la caméra et de la sphère et le rayon de la sphère. Dans un premier temps, nous allons juste envoyer un rayon vers chaque pixel de l'image. Avec la fonction `intersect` de notre classe `Sphere`, nous pouvons regarder pour chaque rayon si il intersecte la sphère. Si c'est le cas, le pixel correspondant au rayon est coloré en blanc, sinon il reste noir.

Grâce à ce premier raytracer, nous obtenons un rond blanc sur un fond noir.

Cependant le rendu de cette image n'est pas très réaliste, notamment à cause du manque de relief. Pour pallier à cela nous allons ajouter un éclairage. Nous considérons que les sphères sont diffuses, c'est-à-dire qu'elles réfléchissent la lumière dans toutes les directions, indépendamment de la direction incidente (Figure 1).

Pour pouvoir avoir une image plus complexe, il est nécessaire de pouvoir représenter une scène constituée de plusieurs sphères. Pour cela nous allons créer une nouvelle classe, la classe `Scene`, constituée d'une fonction `intersect()`, et qui contiendra un tableau de sphères.

Dans la fonction main(), nous pouvons maintenant définir différentes sphères, de position et d'albedo différents, et les ajouter à notre scene. Il faut également maintenant utiliser la fonction intersect() de la classe Sphere maintenant pour déterminer la couleur de chaque pixel.

En choisissant correctement le rayon et la position des sphères nous pouvons obtenir l'image 2. Pour avoir un meilleur rendu, nous prenons en compte un facteur gamma de 2.2 (Figure 3).

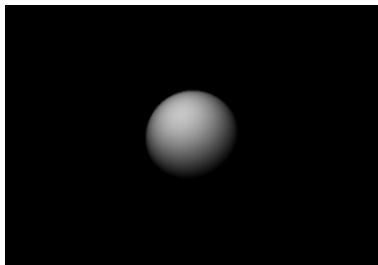


Figure 1: *Image obtenue avec une sphère diffuse*

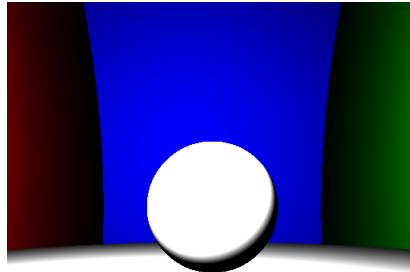


Figure 2: *Scène obtenue en utilisant plusieurs sphères*

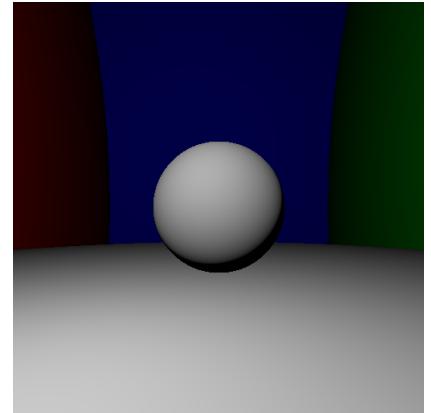


Figure 3: *Scène obtenue après correction gamma*

### 3 Ombres, miroir et transparence

Nous nous intéressons maintenant aux ombres portées sur les boules. Pour cela, dès qu'un intersection P est trouvée, on génère un autre rayon entre P et la source de lumière et on regarde si cette intersection est derrière la source de lumière. Si non il y a une ombre sur la sphère et la couleur du pixel est donc noire.

Pour ne pas obtenir une image bruitée, il faut se placer à un point légèrement décollé de notre surface ( $P + \text{epsilon}$ ).

Après cette correction nous obtenons la Figure 4, en 5 secondes environ.

Nous pouvons maintenant faire des surfaces miroirs. Pour cela nous allons également générer un nouveau rayon partant du point d'intersection P trouvé avec la sphère. Ce rayon va pouvoir être à son tour réfléchi sur une nouvelle sphère, jusqu'à atteindre une sphère diffuse. Le pixel prendra alors la couleur de cette sphère diffuse.

Pour pouvoir faire cela nous avons besoin d'une fonction récursive getColor, qui renvoie la couleur du pixel.

Avec la sphère centrale comme miroir nous obtenons la Figure 5. Cette image a été obtenue en 10 secondes.

Nous allons maintenant gérer la possibilité de surfaces transparentes. Dans ce cas là nous ne gérons plus des rayons réfléchis mais des rayons réfractés par la surface de la sphère. Les directions du nouveau rayon généré sont obtenus grâce à la loi de Snell-Descartes.

Nous avons ajouté un attribut transparent à la sphère, qui est par défaut sur false.

Cette fois-ci pour éviter le bruit, nous plaçons P légèrement à l'intérieur de la sphère ( $P - \epsilon$ ).

Nous obtenons alors la Figure 6, en 11s.

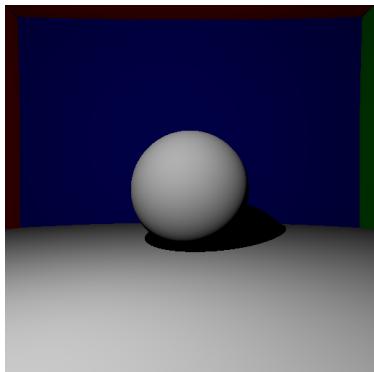


Figure 4: Scène obtenue avec une ombre portée

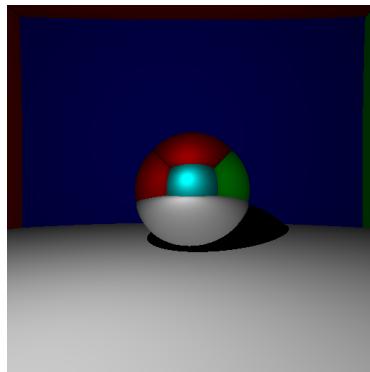


Figure 5: Scène obtenue avec la sphère centrale miroir

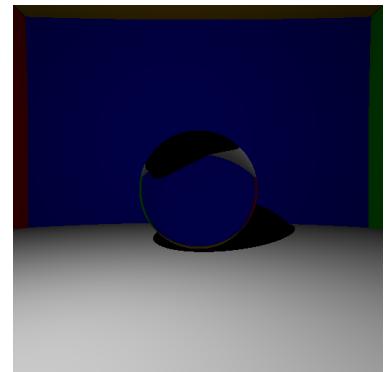


Figure 6: Scène obtenue avec la sphère centrale transparente

## 4 Eclairage indirect

Pour un rendu plus réaliste il faut considérer un éclairage indirect : la lumière va rebondir à différents dans la pièce.

Comme pour les miroirs, nous allons faire rebondir les rayons dans la pièce de manière récursive. Pour calculer la lumière d'un pixel nous allons utiliser l'équation suivante, l'équation du rendu :  $L(\omega_o) = E(\omega_o) + \int_{S+} f(\omega_i, \omega_o).L(\omega_i) < \vec{N}, \vec{\omega}_i > d\omega_i$

avec  $f$  la BRDF.

Pour pouvoir résoudre l'équation du rendu, il faut pouvoir calculer des intégrales de grandes dimensions. Pour cela nous utilisons la méthode de Monte-Carlo.

Pour générer des nombres aléatoires  $x_1$  et  $x_2$  qui suivent une loi gaussienne, on génère  $u_1$  et  $u_2$  qui suivent une loi uniforme centrée réduite, puis on a avec la formule de Bose Muller.

Pour 10 rayons nous obtenons la Figure 7, en 1min05.

Pour 50 rayons nous obtenons la Figure 8 en 5min.

Plus on envoie de rayons plus l'image est de bonne qualité, mais plus le temps d'exécution est long.

Après anti-aliasing on remarque qu'il n'y a plus de "marche" sur les contours de la sphère lorsqu'on zoom (Figure 9 avec 10 rayons envoyés).

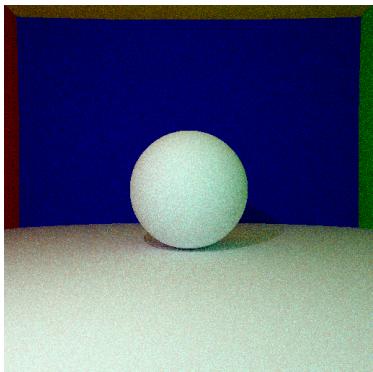


Figure 7: Scène avec éclairage indirect et 10 rayons envoyés

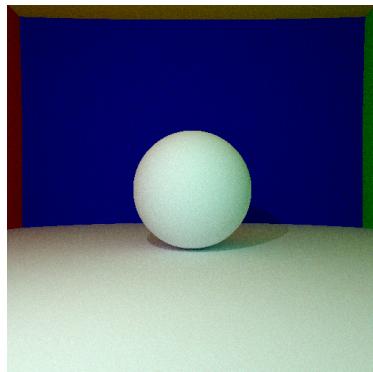


Figure 8: Scène avec éclairage indirect et 50 rayons envoyés

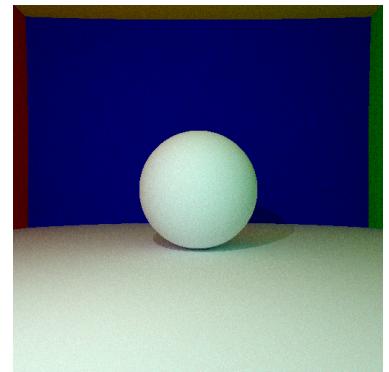


Figure 9: Scène avec éclairage indirect, anti-aliasing

## 5 Ombres douces

Pour ajouter encore du réalisme à notre scène, il serait bien que les ombres soient douces. Pour cela on utilise non plus des sources ponctuelles mais des sources ayant une certaine aire.

Il existe une première possibilité pour avoir des ombres douces, une méthode "naïve". Cela consiste à fixer une valeur positive pour l'émission  $L_e$  de toutes les sources de lumière sphériques, et à attendre que nos rayons aléatoires atteignent cette source de lumière.

Cette méthode fonctionne mais produit des images très bruitées : plus la source de lumière est petite, moins de rayons aléatoires vont l'atteindre et donc plus il y aura de bruit.

Avec une petite source j'obtiens la Figure 10.

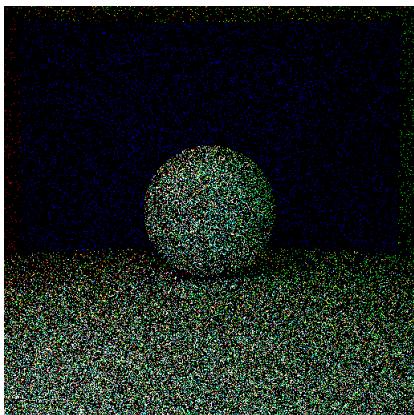


Figure 10: Scène avec ombre douce par la méthode naïve, avec une source de lumière de rayon 5 et en lançant 10 rayons



Figure 11: Scène avec ombre douce par la méthode naïve, avec une source de lumière de rayon 15 et en lançant 10 rayons

Pour supprimer ce bruit, il existe une autre méthode. Dans cette solution on prend en charge séparément l'éclairage direct et l'éclairage indirect. Il y a deux solutions possibles : on peut essayer de trouver des rayons qui vont aller dans la direction de la sphère, ou on

peut générer des échantillon aléatoirement sur la sphère, puis tirer des rayons dans cette direction. Nous avons choisi la deuxième solution.

On obtient alors des images moins bruitées, même avec des sources de lumières de petite taille (Figure 12).

Avec cette méthode, plus la source de lumière est petite, moins il y a de bruit (Figure 13).

Avec cette méthode, nous pouvons simuler des caustiques avec des boules transparentes (Figure 14). On peut remarquer des petites taches blanches, qui sont du bruit. La sphère a du être changée de position pour faire apparaître la caustique.

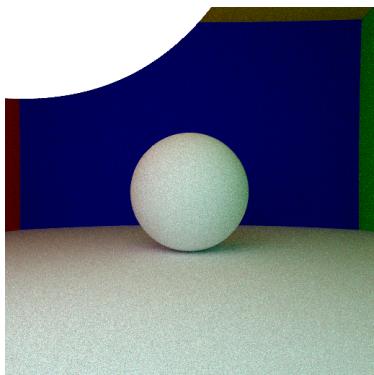


Figure 12: Scène avec ombre douce, avec une source de lumière de rayon 15 et en lançant 50 rayons

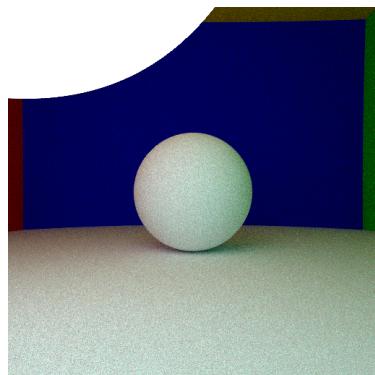


Figure 13: Scène avec ombre douce, avec une source de lumière de rayon 1 et en lançant 50 rayons

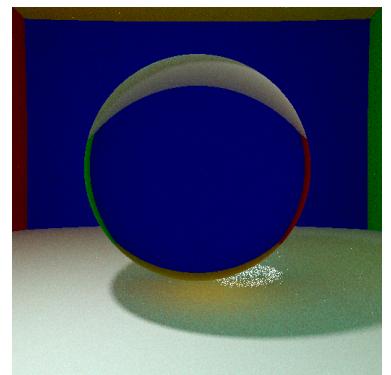


Figure 14: Scène avec une sphère transparente permettant de simuler une caustique

On peut également combiner les différents points vus jusqu'à présent :

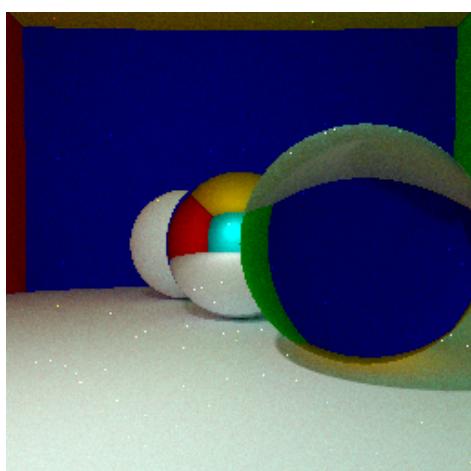


Figure 15: Image de sphères possédant des propriétés différentes

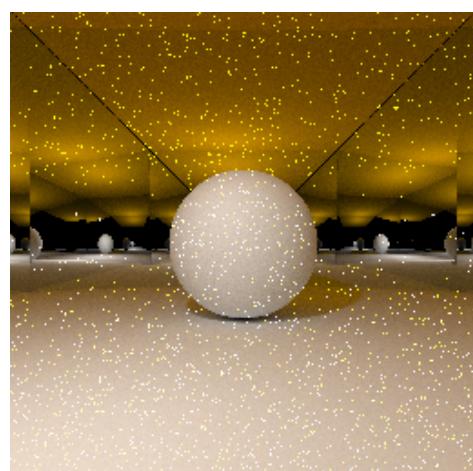


Figure 16: Image d'un scène dans tous les murs sont des miroirs

## 6 Profondeur de champs

Pour l'instant nos images sont nettes dans tous les plans. Or on aimerait avoir un effet "appareil photo", où la focalisation se fait sur un plan, et les objets dans les autres plans sont plus ou moins flous.

Dans notre modèle, la caméra correspond à une caméra pinhole : une boîte noire de taille  $f$  (qui correspond à la distance focale), percée d'un petit trou (pinhole).

On obtient alors des images comme celle ci-après. On peut voir que selon leur position dans la scène, les sphères sont plus ou moins nettes.

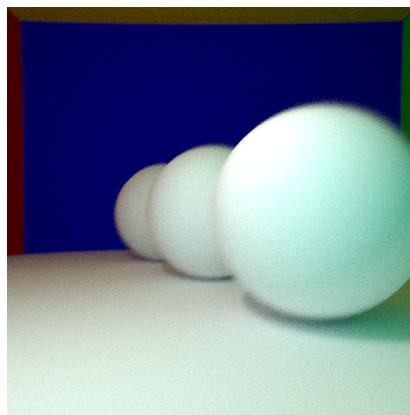


Figure 17: Scène prenant en compte la profondeur de champ, avec 100 rayons

## 7 Maillage triangle

En informatique graphique, le maillage est le plus souvent réalisé avec des triangles.

Il faut stocker ce maillage : pour cela on stocke dans un tableau un ensemble de sommets. On a ensuite une liste de triangles qui sont des références à ces sommets.

Pour l'implémentation, il faut maintenant calculer l'intersection entre un rayon et un triangle.

En implémentant cela, on peut obtenir une image de chien (maillage pris sur internet). Mais le temps de calcul est très important, pour obtenir une image dans un temps raisonnable, on envoie donc un seul rayon et on réduit la taille de l'image à 64x64 (Figure 18, image obtenue en 5 minutes environ).

Pour améliorer un peu le temps de calcul, on peut faire une première amélioration. On voudrait tester tous les triangles que si le rayon va à peu près dans la direction du maillage.

Pour faire cela, on peut utiliser une boîte englobante.

Ca nous permet de gagner du temps et d'obtenir une image en 256x256 et avec 10 rayons en un temps raisonnable (environ un quart d'heure) (Figure 19).

Ce code peut encore être accéléré en utilisant la méthode de hiérarchie de boîte englobante (Bounding Volume Hierarchies), qui consiste à implémenter la méthode précédente récursivement.

Cela permet de gagner un facteur 15 en temps et donc d'obtenir des images de meilleure qualité en un temps raisonnable (j'ai également entre temps replacé correctement le chien dans l'espace) (Figure 20 obtenue en environ 7 minutes).

Si on zoom sur cette image, on peut observer des traits sur le chien. Pour résoudre ce problème on fait une interpolation des normales. Chaque sommet peut en effet avoir plusieurs normales. (Figure 21).

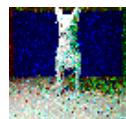


Figure 18: *Image de chien 64x64, avec un seul rayon envoyé*

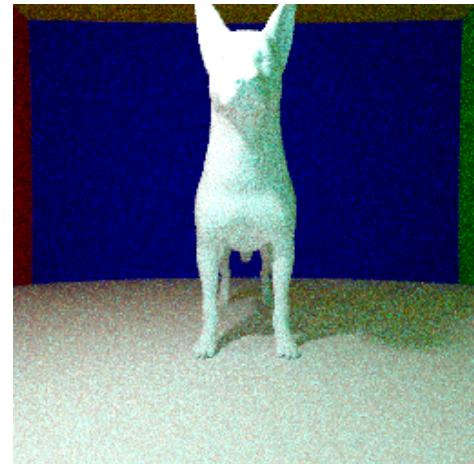


Figure 19: *Image de chien 256x256, avec 10 rayons envoyés*

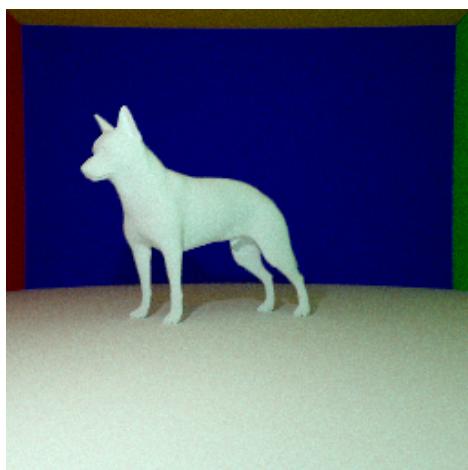


Figure 20: *Image de chien 256x256, avec 100 rayons envoyés*

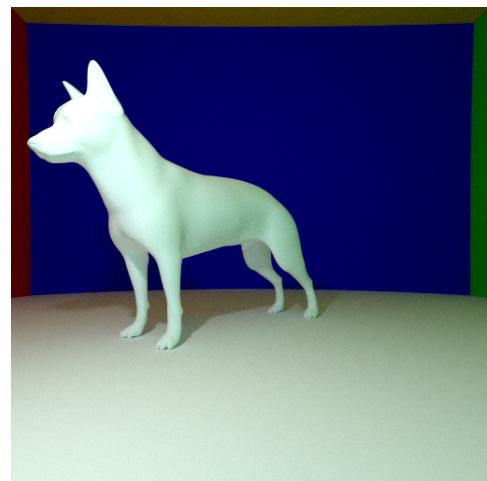


Figure 21: *Image 256x256, avec 200 rayons envoyés, avec interpolation des normales*

On a ensuite ajouté les textures, pour que le chien ait un pelage (Figure 22 obtenue en environ 5 minutes pour 256x256 et 100 rayons envoyés).

Pour pouvoir observer la scène sous différents angles, il est également possible de modifier l'angle de la caméra (Figure 23).



Figure 22: *Image du chien en ajoutant les textures*

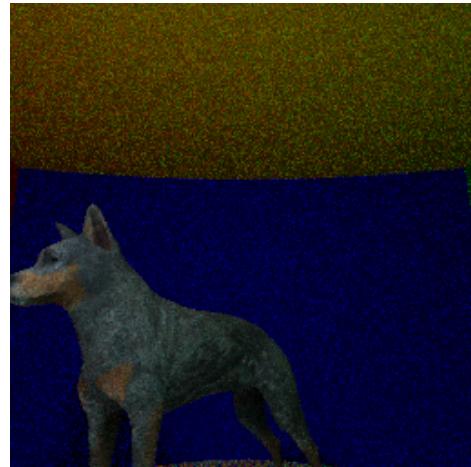


Figure 23: *Image du chien en tournant la caméra vers le haut*

On peut alors faire une scène plus complète, en combinant les sphères et le maillage de chien.

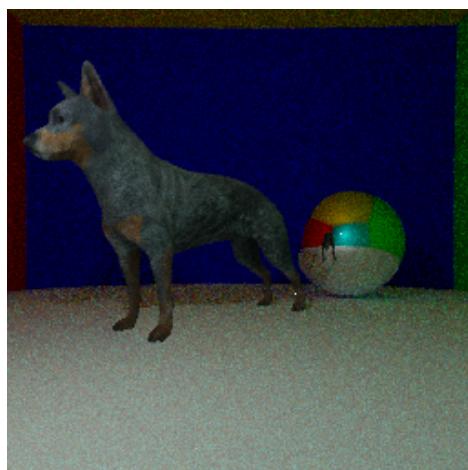


Figure 24: *Image d'un chien et d'une sphère miroir*

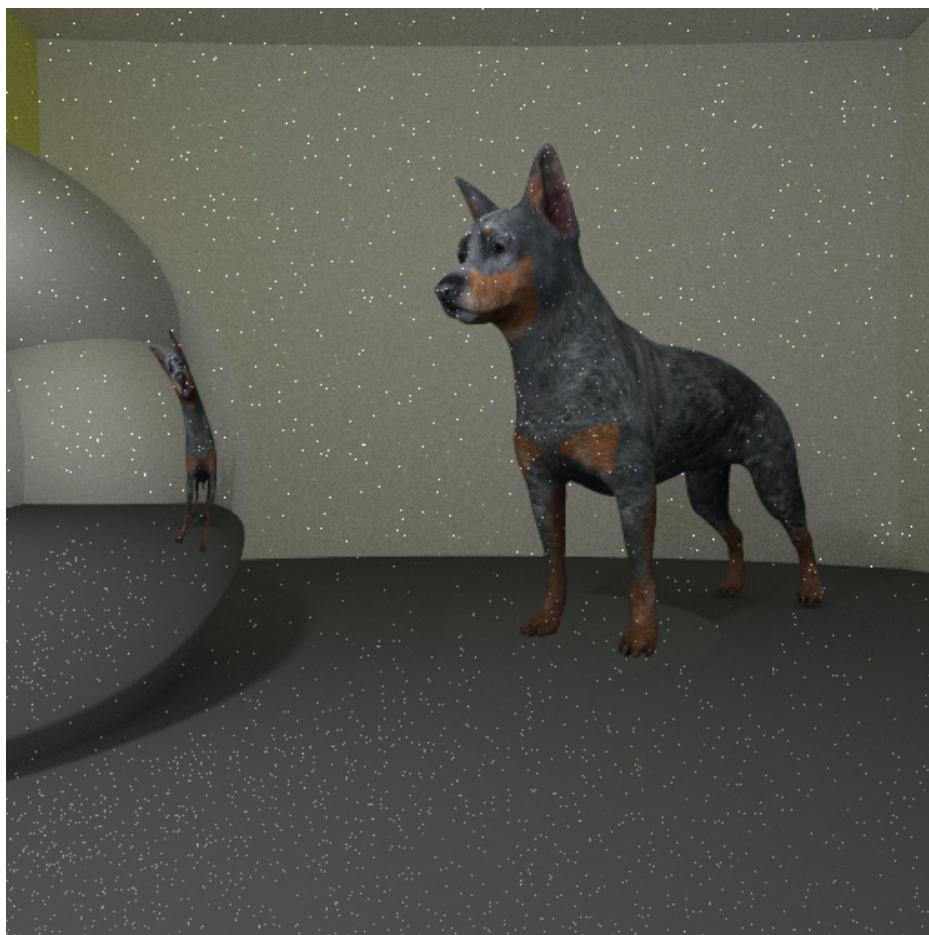


Figure 25: Image d'un chien se regardant dans le miroir (512x512, 200 rayons)

## 8 Commentaires

Avant ce cours je n'avais jamais codé en C++ (seulement en Python donc je n'ai pas pu le faire avec un autre langage), c'était donc difficile, notamment au début, de développer son propre code et cela demande beaucoup de travail personnel pour s'approprier la syntaxe en plus de comprendre les principes du raytracing. Par exemple avec la classe Object, j'ai du faire comme vous car je ne savais pas utiliser "virtual". De même je ne connaissais pas l'existence de "this", ou l'utilisation de "->" à la place de "..".

Sinon le cours est intéressant, à distance le fait que vous écriviez sur le tableau et qu'il y avait plusieurs supports écrits disponibles ont fait que le cours était bien compréhensible. Cela m'a permis de découvrir un domaine que je ne connaissais pas du tout.