
École Centrale Lyon

MOS 2.2
Rapport

# Rendu réaliste par raytracing

***Élève :***
Guillaume GROULT

***Enseignant :***
Nicolas Bonneel

29 mars 2021

# Table des matières

# Introduction

Dans le cadre de l'action de formation MOS 2.2 " Informatique Graphique ", nous devons développé un Raytracer qui a but de générer une image de rendu réaliste par l'envoi de rayons lumineux dans la scène. Pour commencer, nous nous intéresserons à l'affichage d'un forme simple : la sphère. Ensuite, nous occuperons de l'affichage d'objets complexes.

# 1 Affichage de sphères

## 1.1 Intersection Rayon-Sphère

Le principe du raytracing repose sur l'envoi de rayons. Pour l'instant, notre scène ne contient qu'une seule sphère et chaque pixel de l'image reçoit un rayon. Si le rayon envoyé depuis la caméra entre en intersection avec la sphère, alors le pixel sera blanc, sinon il sera noir. Nous avons pour cela créer les classes **Vect**, **Ray** et **Sphere**. La classe **Vect** représente un vecteur contenant 3 coordonnées de type `double` ainsi que l'ensemble des opérations vectorielles essentielles pour la suite du projet (de la somme de vecteur au produit vectorielle en passant à la normalisation de vecteur). Soit $P$ un point appartenant à la demi-droite "rayon" d'origine $C$ et de direction $u$ et à la sphère de centre $O$ et de rayon $R$. Nous obtenons le système d'équation (1) qui nous donne l'équation du second dégré (2). La classe **Sphere** contient la méthode `intersect` qui résout cette équation. La FIGURE 1 montre bien la présence d'une sphère dans la scène après une seconde d'éxécution.

$$\begin{cases} P = C + t \times u \\ ||P - O||^2 = R^2 \end{cases} \tag{1}$$

$$t^2 + 2 < u, C - O > t + ||C - O||^2 - R^2 = 0 \tag{2}$$

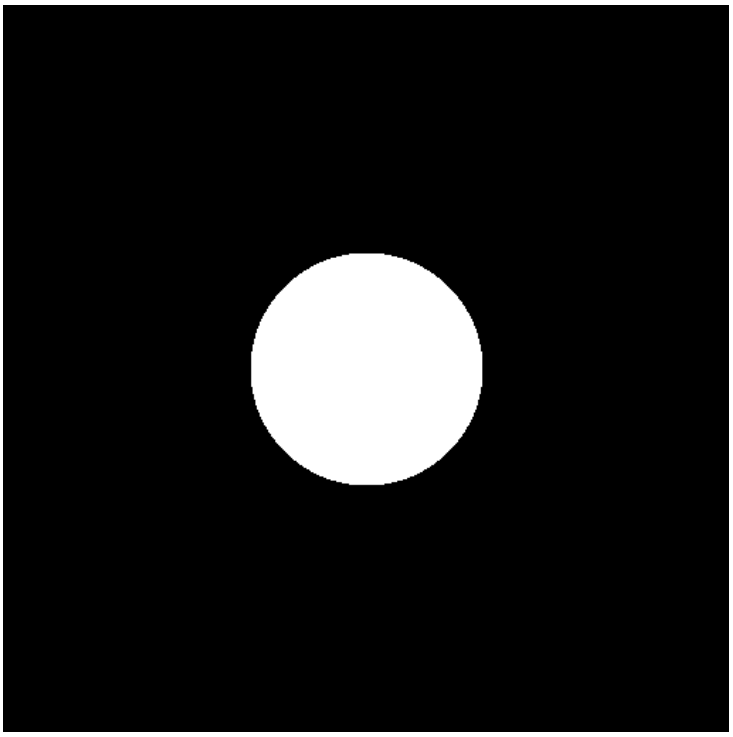

FIGURE 1 – Intersection d'une sphère par un rayon

Maintenant que nous avons affiché la présence de la sphère, intéressons nous à la variation de lumière sur la sphère. Pour cela, nous ajoutons une source de lumière ponctuelle de coordonnées $L$ et d'intensité lumineuse $I$ et nous notons $\vec{n}$ la normale au point d'intersection $P$. La couleur du pixel ne sera plus blanc mais suivra la formule (3) avec $\rho$ le vecteur albedo qui représente la couleur de la sphère. Avec l'albedo (R,V,B) = (1,0,0), nous obtenons une sphère de couleur rouge sur la FIGURE 2 en moins d'une seconde d'exécution.

$$Color = \frac{I}{4\pi||\vec{PL}||^2} < \vec{n}, \frac{\vec{PL}}{||\vec{PL}||} > \frac{\rho}{\pi} \tag{3}$$

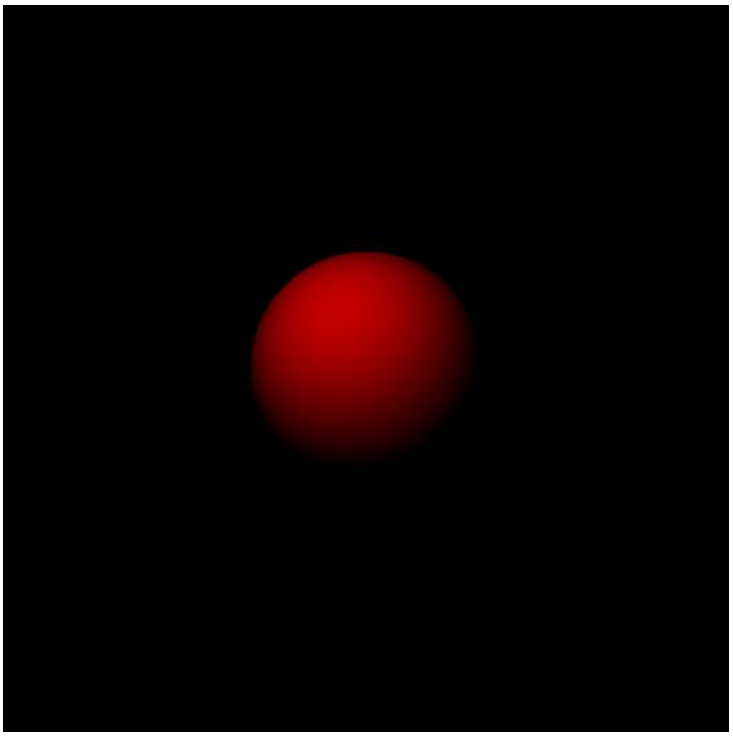

FIGURE 2 – Ajout de la variation de lumière sur la sphère détectée

Maintenant que nous avons établi l'éclairage direct sur notre sphère, nous allons définir le décor de la scène. La scène aura un plafond et un sol blanc ainsi que quatre murs de couleur bleu, rouge, vert et jaune. Ces éléments de décor seront des sphères très éloignés de la caméra ayant un rayon très important pour que nous ayant l'impression d'avoir des murs plans. Cela nous permet de créer la classe **Scene** qui contient elle aussi une méthode `intersect` qui fera appelle à la méthode `intersect` des sphères qui la composent. Comme il est maintenant possible d'avoir plusieurs intersections sphère-rayon pour un même rayon, nous ne prenons en compte que celui dont la racine $t$ est le plus petit (cela désigne l'intersection le plus proche de la caméra) tout en restant positive. Nous obtenons la FIGURE 3 en moins d'une seconde d'exécution.

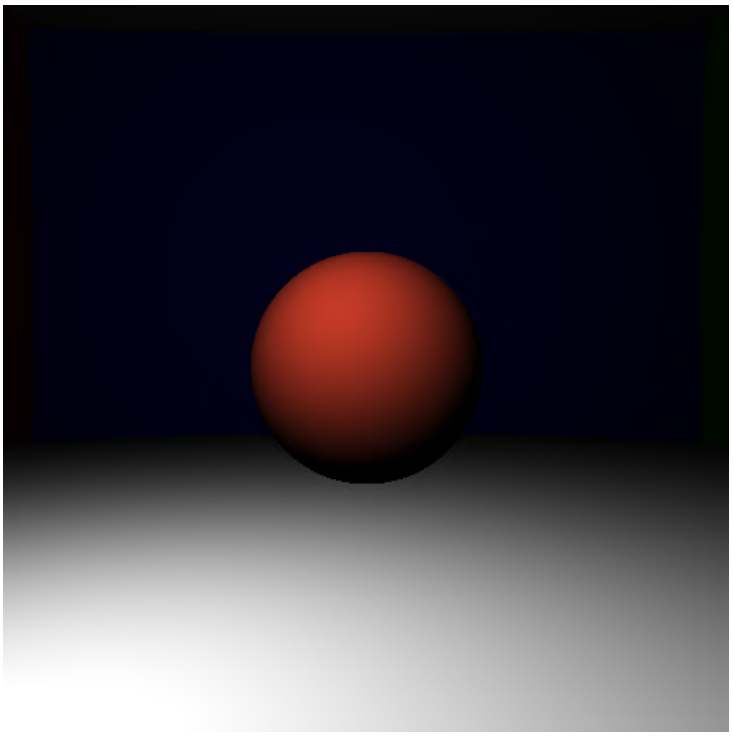

FIGURE 3 – Ajout de la scène

## 1.2 Correction Gamma

Nous avons réussi affiché une scène constitué de plusieurs sphères. Cependant, nous avons du mal à distinguer les murs de la scène et les écrans appliquent un facteur 1,22 aux images générées. Pour cela, nous allons faire de la correction Gamma. Cela consiste à élever la valeur des couleurs RVB à la puissance $\frac{1}{1,22}$. Nous obtenons la FIGURE 4 en moins d'une seconde.

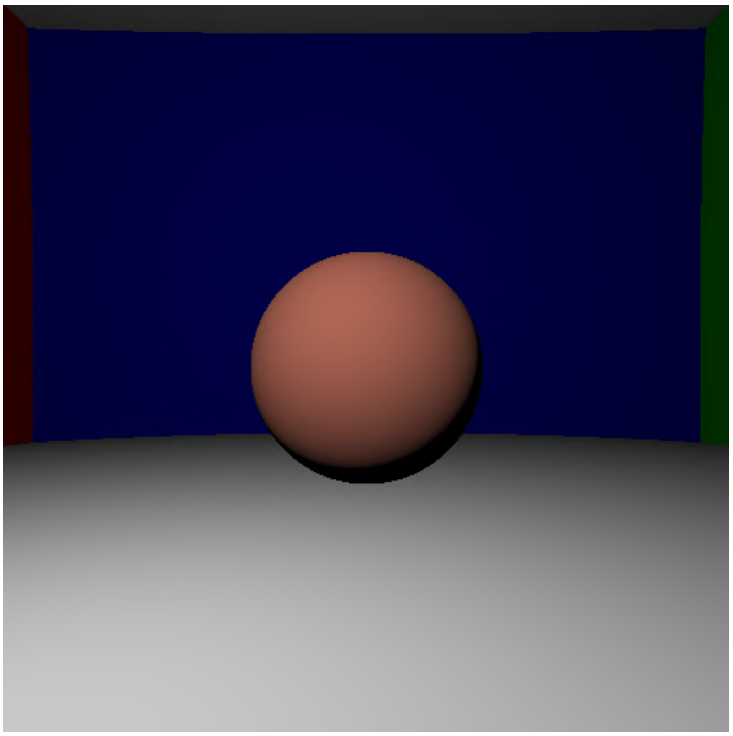

FIGURE 4 – Ajout de la correction gamma

## 1.3 Ombres portées

Nous arrivons à afficher une sphère dans la scène mais comme tout objet éclairé, il y a une ombre portée qui lui est associée. S'il y a une intersection sphère-rayon, nous envoyons un rayon depuis ce point d'intersection $P$ vers la source de lumière $L$. S'il y a une autre intersection situé en $P$ et $L$, le pixel où est situé $P$ sera noir. Nous obtenons la FIGURE 5 en moins d'une seconde.

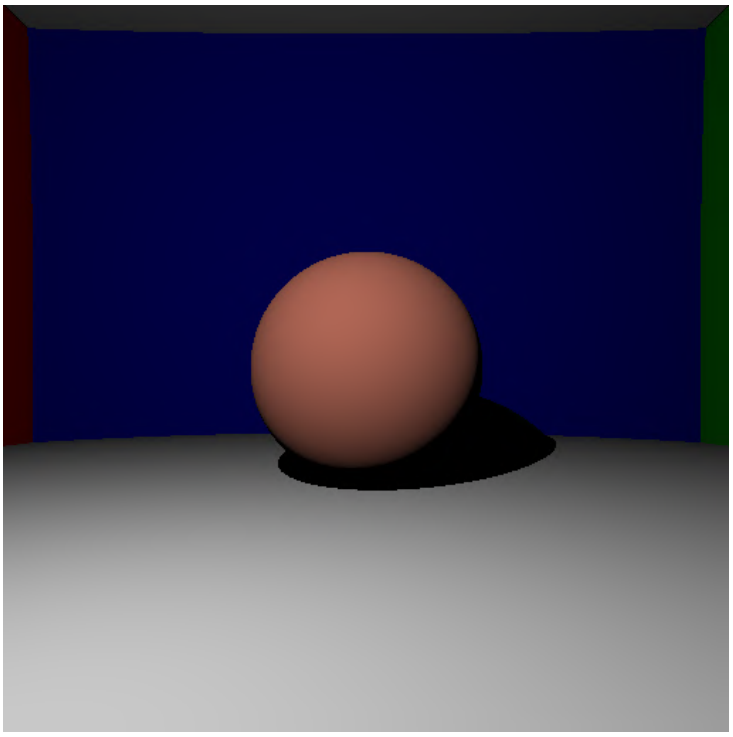

FIGURE 5 – Ajout de l'ombre portée

## 1.4 Miroir

Nous arrivons à afficher des sphères diffuses. Nous nous intéressons maintenant à des sphères miroirs. Un miroir signifie que nous devons prendre en compte le phénomène de réflexion. La direction du rayon réfléchie à la surface du miroir suit l'équation (4). De plus, le calcul de la couleur des pixels de l'image est placé dans la méthode `getColor` de la classe **Scene** afin de pouvoir l'appeler de manière récursive avec le rayon réfléchie. Nous décalons également la composante normale du rayon réfléchie pour des raisons d'imprécisions numériques. Nous obtenons la FIGURE 6 au bout d'une seconde.

$$\vec{u_{reflechie}} = \vec{u_{incident}} - 2 < \vec{u_{incident}}, \vec{n} > \vec{n} \tag{4}$$

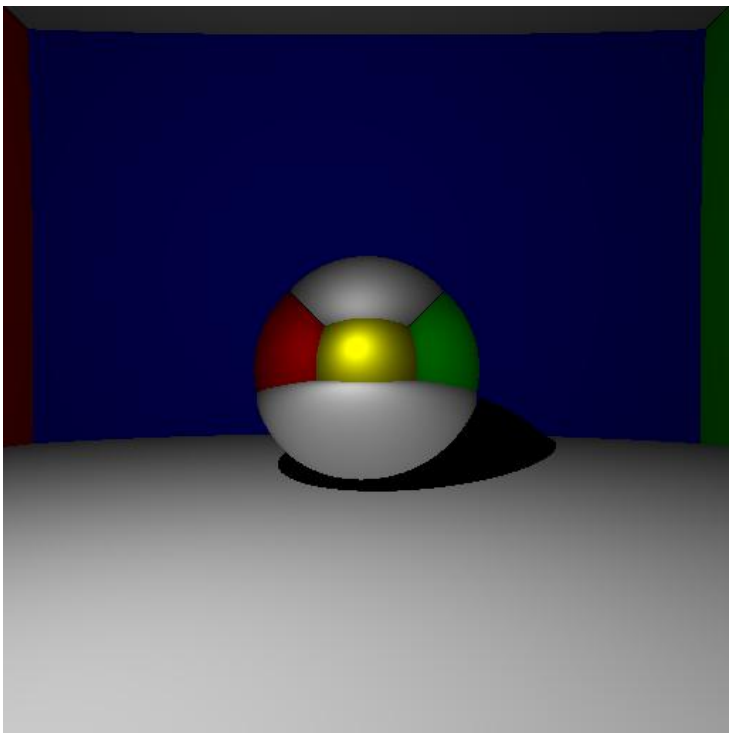

FIGURE 6 – Sphère miroir

## 1.5 Milieu transparent

Pour le milieu transparent, nous appliquons la deuxième loi de Snell-Descartes qui nous détermine la direction du rayon réfracté. Nous supposons que le milieu transparent à pour indice optique celui du verre. Lorsque le rayon sort de la sphère, nous inversons les indices optiques et les normales à la surface de la sphère. Nous obtenons la FIGURE 7 en moins d'une seconde. Nous remarquons que l'ombre portée est visible à la surface de la sphère transparente. Cela s'explique par le fait que nous n'avons pris en compte que l'éclairage directe.

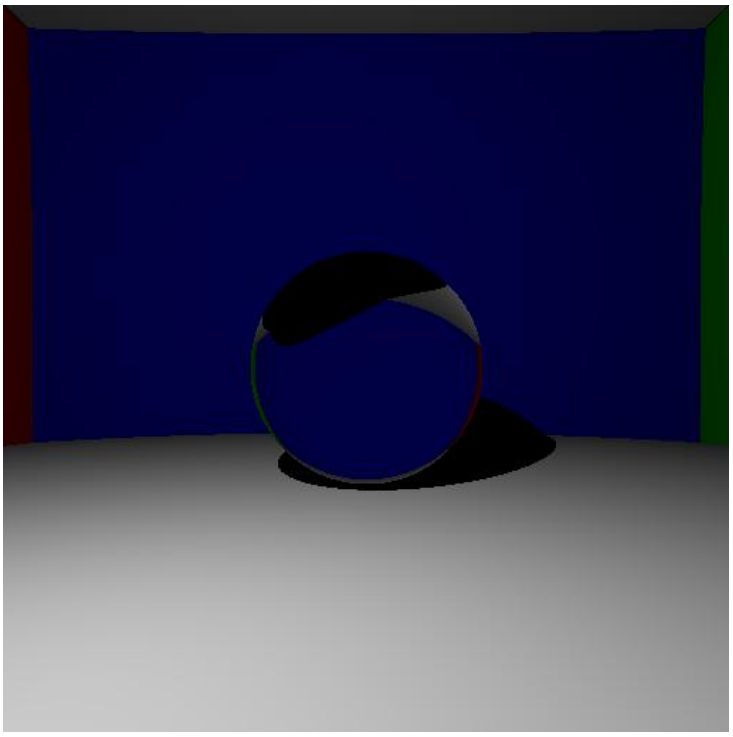

FIGURE 7 – Sphère transparente

## 1.6 Éclairage indirecte

Pour implémenter l'éclairage indirecte, nous devons partir de l'équation du rendu (5). Cette équation donne la couleur d'un pixel. Nous pouvons faire une approximation de l'intégrale par la méthode de Monte-Carlo (cf équation (6)). On génère donc un rayon de direction aléatoire compris dans un hémisphère au niveau du point d'intersection. La couleur obtenue par l'appel de `getColor` sur ce rayon est celle obtenue par éclairage indirecte et sera ajouté à celle obtenue par éclairage directe. Nous envoyons dans plusieurs rayons par pixel et réalise une moyenne sur les couleurs obtenues pour chaque pixel. Pour 100 rayons émis et une limite de 5 rebonds dans la récursion de `getColor`, nous obtenons la Figure 8 au bout de 357 secondes soit environ 6 minutes. De plus, pour économiser du temps d'exécution, nous parallélisons les opérations sur les boucles for avec OpenMP. De cette manière, toujours avec 100 rayons par pixel et une limite de 5 rebonds, nous passons à 112 secondes de temps d'exécution.

$$L_o(x,\vec{o}) = E(x,\vec{o}) + \int f(\vec{i},\vec{o}) L_i(x,\vec{i}) \cos(\theta_i)\, d\vec{i} \tag{5}$$

$$\int f(\vec{i},\vec{o}) L_i(x,\vec{i}) \cos(\theta_i)\, d\vec{i} \approx \frac{1}{n} \sum_{i=0}^{n} f(\vec{i_i},\vec{o}) L_i(x,\vec{i_i}) \cos(\theta_i)/p(\vec{i_i}) \tag{6}$$

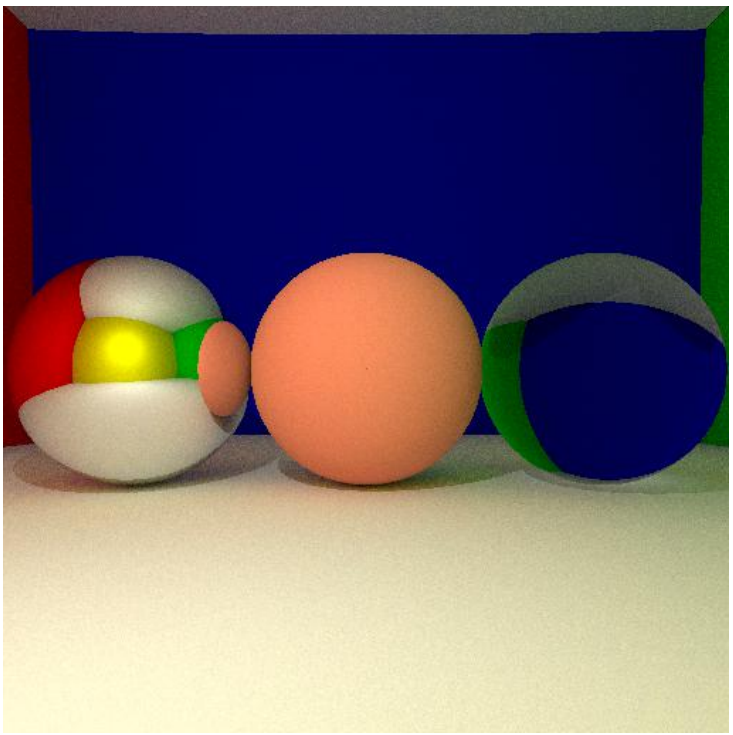

Figure 8 – Ajout de l'éclairage indirecte

## 1.7 Anti-crénelage

En zoomant sur les bordures des images précédentes, nous pouvons observer un effet de créneau de château sur les courbes de la sphère. Cet effet visuel s'appelle le crénelage. Pour supprimer ce défaut, nous envoyons le rayon aléatoirement dans une zone proche du pixel ciblé. L'aléatoire utilisée repose sur des échantillons gaussiens obtenue par la méthode de Box-Muller. Nous obtenons la FIGURE 9 en 114 secondes.

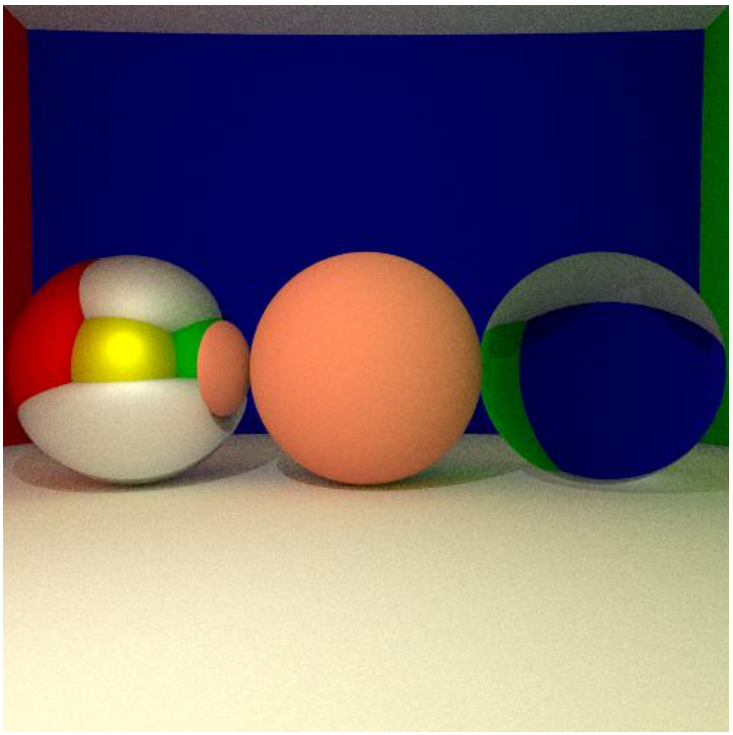

FIGURE 9 – Ajout de l'anti-crénelage

## 1.8 Ombres douces

Pour adoucir les ombres, nous devons changer de source de lumière. Jusqu'à présent, la source de lumière était ponctuelle et nous la remplaçons par une source sphérique. Cela implique que les rayons de l'éclairage indirect doivent atteindre la source de lumière donc nous devons prendre une sphère de lumière de rayon assez importante. De plus, nous devons réimplémenter l'éclairage direct afin que les rayons émis aléatoirement aient plus de chances d'atteindre la source de lumière. Nous obtenons la FIGURE 10 en 112 secondes. Nous remarquons que la source de lumière sphérique n'est pas représentée sur à travers le miroir. Nous avons décidé de ne pas l'afficher sur cette image seulement pour mieux comparer le résultat obtenue avec la FIGURE 9.

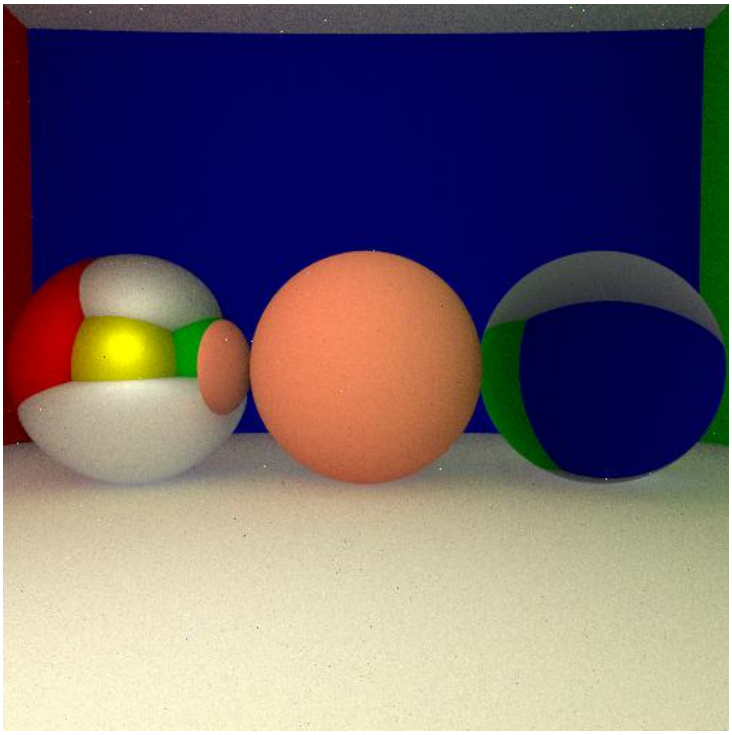

FIGURE 10 – Ajout de l'ombre douce

## 1.9 Profondeur de champs

Intéressons-nous à la profondeur de champs de la caméra. Nous avons pour cela besoin de connaître les coordonnées du point de mise au point de la caméra ( = focus en anglais). Nous supposons que la caméra est assimilé à une lentille de distance focale $f = 55$. Cela nous donne pour mise au point F (7). Maintenant que la mise au point est réalisé, nous envoyons des rayons de plusieurs points dans le disque d'ouverture de la caméra (dans la réalité, ce disque permet le passage de lumière dans l'appareil photo via le diaphragme). Ces rayons auront donc pour origine $C'$ (choisi aléatoirement dans le disque d'ouverture par la méthode de Box-Muller) et pour direction $\vec{u'}$ (8). Nous obtenons la FIGURE 11 au bout d'une minute. Nous pouvons remarquer qu'il y a bien un flou d'appliquer sur les objets qui sont trop éloignés du point de mise au point (comme la sphère verte ou la double sphère transparente). De plus, nous affichons maintenant la source de lumière dans la scène. L'apparition de cette sphère nous révèle la présence de pixels blancs derrière la sphère transparente. Ces pixels sont le résultats de ce qu'on appelle des caustiques.

$$F = C + f\vec{u} \tag{7}$$

$$\vec{u'} = \vec{C'F} \tag{8}$$

FIGURE 11 – Ajout de la profondeur de champs

# 2 Affichage d'objets complexes

## 2.1 Affichage naïve d'un maillage

Dans le chapitre précédent, nous nous sommes concentrés sur l'affichage de sphères dans la sphères. Nous allons maintenant pouvoir afficher des objets complexes. Ces objets existent sous la forme de maillage triangulaire. Nous définissons la classe **TriangleMesh** et **TriangleIndices** ainsi que la méthode `readOBJ` pour extraire le maillage triangulaire d'un fichier OBJ (cette partie a été fourni par l'enseignant). Maintenant que nous pouvons lire des fichiers OBJ, il faut s'intéresser à l'affichage de ces objets. Pour cela, nous devons calculer s'il y a intersection triangle-rayon pour chacun des triangles du maillage. Nous obtenons donc la méthode `intersect` de la classe **TriangleMesh**. De plus, la scène ne comprends plus que des sphères, nous créons donc la classe **Object**, contenant lui aussi une méthode `intersect`, dont les classes **Sphere** et **TriangleMesh** héritent. Pour la suite du projet, nous nous sommes intéressé aux fichiers OBJ suivants :

— A : Diorama sur Link de The Legend of Zelda : The Wind Waker
— B : Bouclier Mojo de The Legend Of Zelda : Ocarina Of Time
— C : Link de The Legend Of Zelda : Ocarina Of Time

Nous avons réduit la résolution de l'image pour limiter les temps de calcul, passant d'une résolution 512x512 pixels à 32x32 pixels. Nous obtenons la FIGURE 12 avec les OBJ A et B au bout de 15 minutes.

FIGURE 12 – Lecture naive de fichier OBJ

## 2.2 Boîte englobante

Nous avons un temps de génération d'image assez long pour une faible résolution. Heureusement, il existe des méthodes pour accélérer la génération d'image. Pour commencer, nous pouvons calculer la boîte englobante du maillage. Nous créons la classe **BoundingBox** qui devra qui déterminera s'il rentre en intersection avec un rayon. S'il y a intersection, alors nous pouvons regarder s'il y a intersection au niveau du maillage. Nous créons donc la méthode `buildBB` qui va réaliser la boîte englobante du maillage. Nous n'avons pas pu obtenir de résultat que des scènes vides ou un seul trait de matière comme sur la FIGURE 13

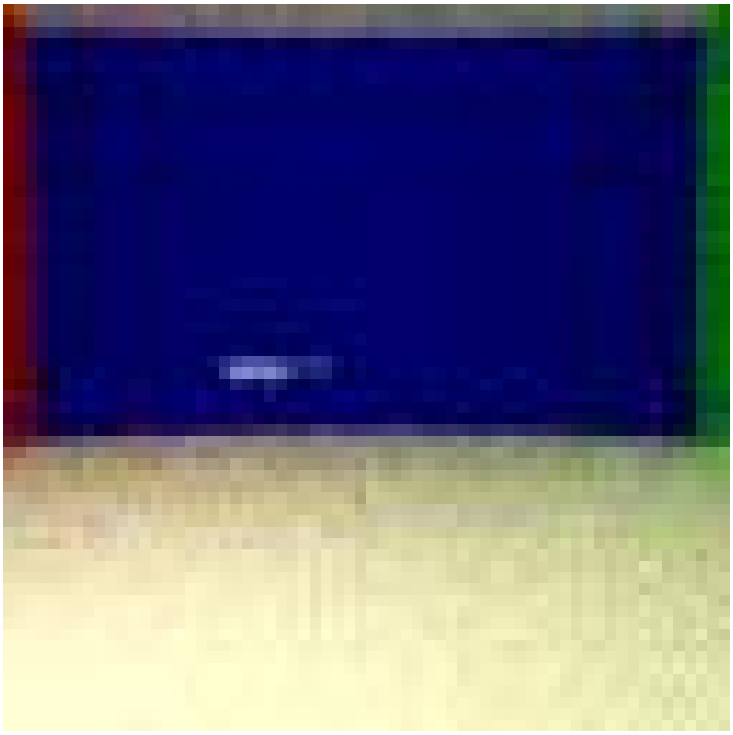

FIGURE 13 – Lecture avec boîte englobante de fichier OBJ

## 2.3 Hiérarchie des boîtes englobantes

Une autre méthode d'accélération est la hiérarchie des boîtes englobantes (BVH en anglais). Elle consiste à représenter le maillage sous forme d'arbre binaire de boite englobante. S'il y a intersection rayon-boîte englobante, nous coupons la boîte englobante en deux autres boîtes englobantes et nous regardons dans quelles boîtes enfants il y a intersection. Nous continuons le découpage des boîtes de cette manière jusqu'à tomber sur une boîte englobante contenant qu'un seul triangle, celui où a lieu l'intersection rayon-maillage. Dans le cas où le rayon entre en intersection avec les deux boîtes enfants, nous regardons que dans celle qui se trouve le plus de l'origine du rayon. Nous réalisons donc la méthode `buildBVH` qui réalise la BVH du maillage. Nous nous sommes appuyés sur le code de l'enseignant pour obtenir une méthode `buildBB` opérationnelle pour l'appel de `buildBVH`. Nous obtenons la FIGURE 14 au bout de 3 minutes pour les OBJ B et C et une résolution de 512x512.

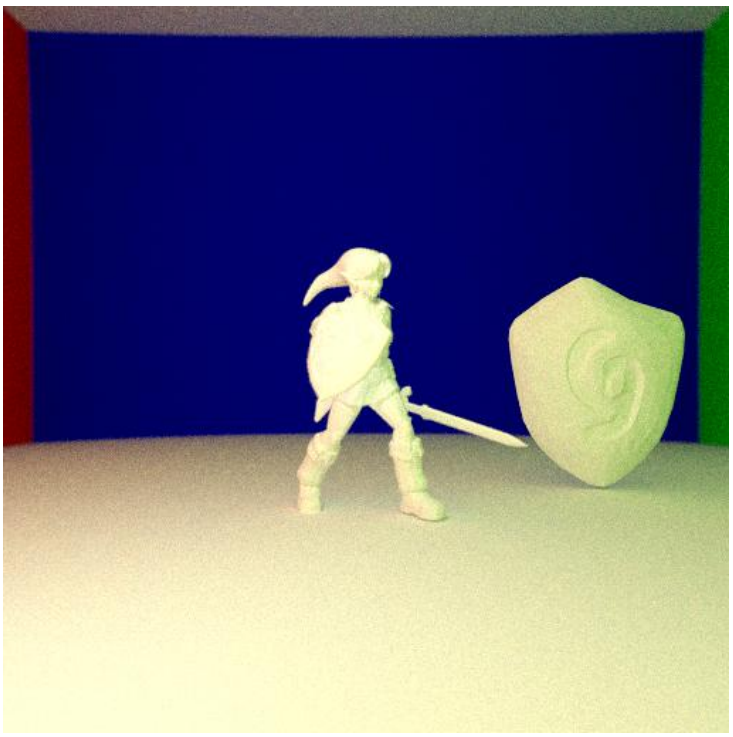

FIGURE 14 – Lecture avec BVH de fichier OBJ

## 2.4 Lissage des maillages

Nous arrivons à afficher des fichiers OBJ assez rapidement sur des images de bonnes résolutions. Maintenant nous pouvons lisser le maillage afin d'enlever le côté "boule à facette" sur les objets affichés. Pour cela, au lieu de prendre la normale calculée pour chacun des triangles au point d'intersection, nous pouvons utilisé les normales présents dans le fichiers OBJ (ces normales sont liées aux sommets des triangles) et faire une combinaison linéaire des ces normales pour obtenir une nouvelle normale au point d'intersection. Les coefficients de la combinaison linéaire sont les coordonnées barycentriques du triangle intersecté. Nous obtenons la FIGURE 15. Nous remarquons que nous avons bien fait un congé d'arrête sur la gravure du bouclier Mojo de l'OBJ B. Nous obtenons aussi la FIGURE 16 pour la même durée avec les OBJ A et C. Cette scène sera utilisée pour la fin du rapport.

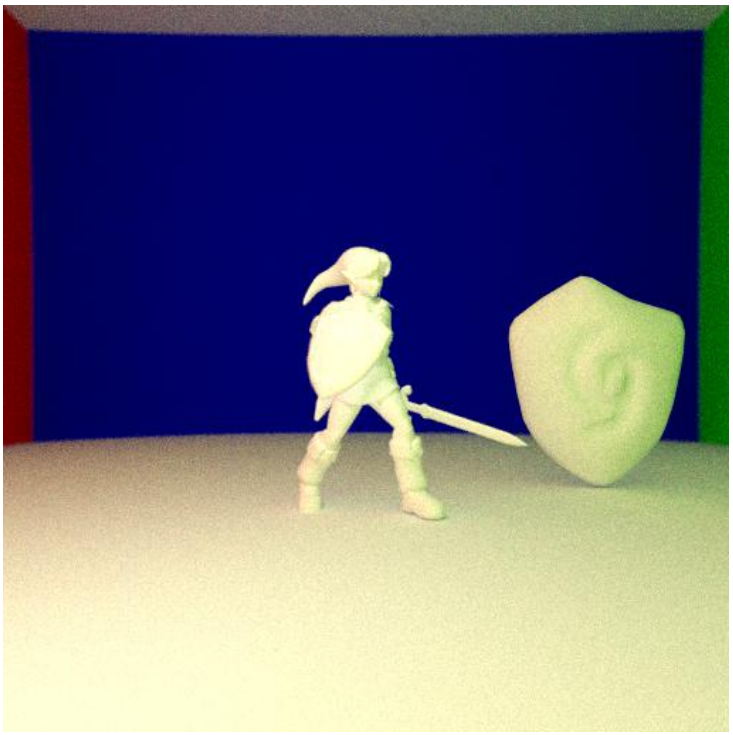

FIGURE 15 – Lissage des maillages (OBJs B et C)

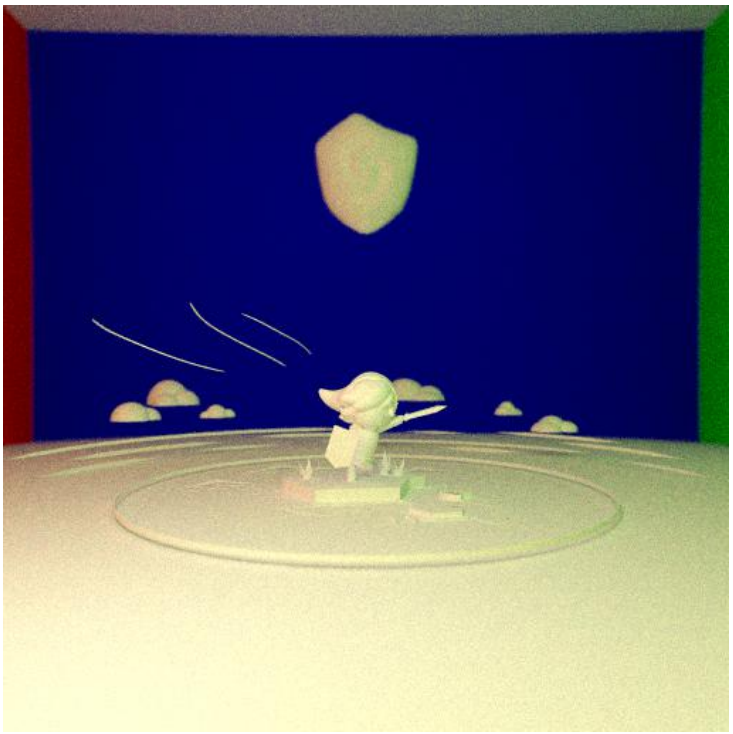

FIGURE 16 – Lissage des maillages (OBJs A et B)

## 2.5 Application des textures

Maintenant que nous avons des maillages lisses, nous pouvons appliqués les textures qui leur sont liées. Nous créons la méthode `loadTexture` qui lit une image et applique les pixels de l'image sur le maillage. Les liaisons entre la texture et le maillage sont réalisés par lignes "vt" du fichier OBJ extraite par la méthode `readOBJ`. Nous utilisons les textures présents dans les FIGURES 17, 18 et 19. Nous obtenons pour le couple d'OBJs (B,C) la FIGURE 20 et pour le couple d'OBJs (A,B) la FIGURE 21. Nous remarquons que l'OBJ C a des problèmes de texture. Cela s'explique par le fait que la texture est divisée en plusieurs images qui sont applicables sur différentes parties de l'OBJ (la texture utilisée sur la FIGURE 20 correspond à la tunique de Link). Pour régler ce problème, il faudrait soit prendre en compte le fichier MTL qui permet l'application de ces textures au bon endroit, soit diviser l'OBJ C en plusieurs OBJs qui auront chacun sa propre texture (les différents OBJs sont séparés par des lignes "o" dans l'OBJ C).

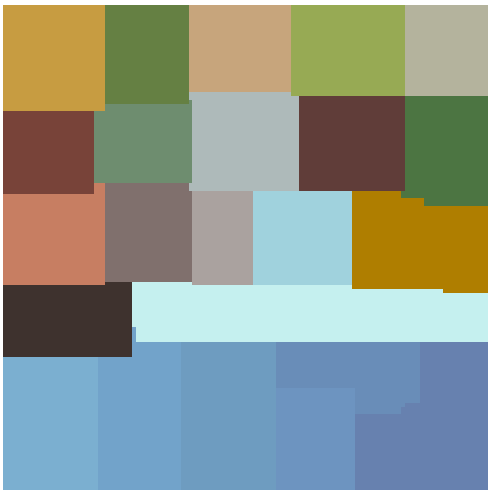

FIGURE 17 – Texture liée à l'OBJ A

FIGURE 18 – Texture liée à l'OBJ B

FIGURE 19 – Texture liée à l'OBJ C

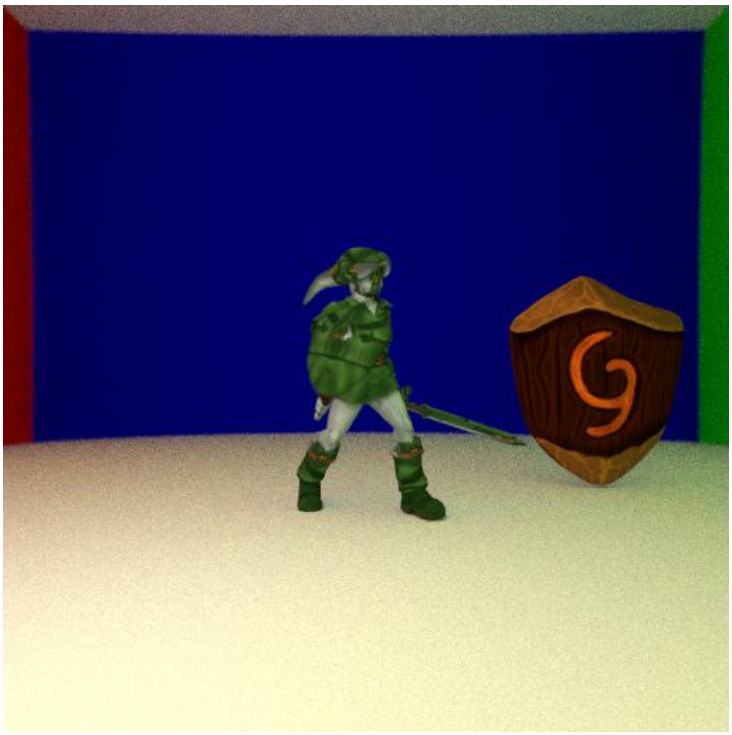

FIGURE 20 – Textures (OBJs B et C)

FIGURE 21 – Textures (OBJs A et B)

### 2.6 Mouvements de caméra

Nous pouvons également déplacé notre caméra dans la scène. Outre les translations sur les trois dimensions de la scène qui sont assez facile à modifier, nous définissons le repère $(\vec{up}, \vec{right}, \vec{viewDir})$ pour prendre en compte les rotations horizontales et verticales. Les vecteurs $\vec{up}$ et $\vec{right}$ sont calculés par la formule de Rodrigues et le vecteur $\vec{viewDir}$ n'est autre que le produit vectoriel de $\vec{up}$ et $\vec{right}$. Nous obtenons la FIGURE 22 pour le couple d'OBJs (A,B) qui est le résultat d'une translation dans les y positifs, les x positifs et les z négatifs ainsi qu'une rotation horizontale trigonométrique (= vers la gauche) et une rotation verticale trigonométrique (= vers le bas). Le fichier GIF (disponible à ce lien : https://i.ibb.co/0BXRfmG/Resultat-mouvement.gif ) nous montre la décomposition des mouvements de la caméra pour obtenir la FIGURE 22.

FIGURE 22 – Mouvements de la caméra (OBJs A et B)

## Conclusion

Nous avons réalisé à travers cette action de formation un raytracer qui capable de faire des rendus de scène réalistes composés de forme plus ou moins complexes. Il y a cependant des points à améliorer comme l'application de plusieurs textures sur un seul fichier OBJ ou la fragmentation d'un OBJ composé en plusieurs maillages (cas d'exemple : l'OBJ C). De plus il existe d'autres moyen d'accélérer la génération d'images comme en optimisant la lecture de la BVH.