# Compte Rendu Raytracing - Phase 1, 2, 3, 4 et Bonus

COMBOT Evan

07/01/2024

**Programmation 3D**

**Université de Montpellier - Master 1 IMAGINE**

**Résumé**

Ce projet vise à développer un moteur de rendu en C++ basé sur du lancer de rayons ou "raytracing". L'objectif est d'implémenter diverses fonctionnalités telles que des modèles d'éclairage comme le modèle de Phong, des modèles d'ombrage comme l'ombrage dure ou l'ombrage doux ainsi que d'autres fonctionnalités.

# Phase 1

## 1. Les fonctions d'intersections

### 1.1 La fonction d'intersection de la sphère

Concernant la fonction d'intersection de la sphère, j'ai d'abord calculé le discriminant en suivant la formule du cours. Si ce discriminant est positif, je procède ensuite au calcul des deux racines. Je détermine la racine la plus petite et la racine la plus grande puis j'attribue la valeur de la plus petite racine à l'attribut 't'. Je calcule ensuite les deux intersections à l'aide de la valeur minimale et maximale de 't'. J'attribue l'intersection la plus petite à l'attribut 'intersection' et l'intersection la plus grande à l'attribut 'secondIntersection'. Enfin, je calcule la normale que je normalise et dont j'attribue la valeur à l'attribut 'normal'.

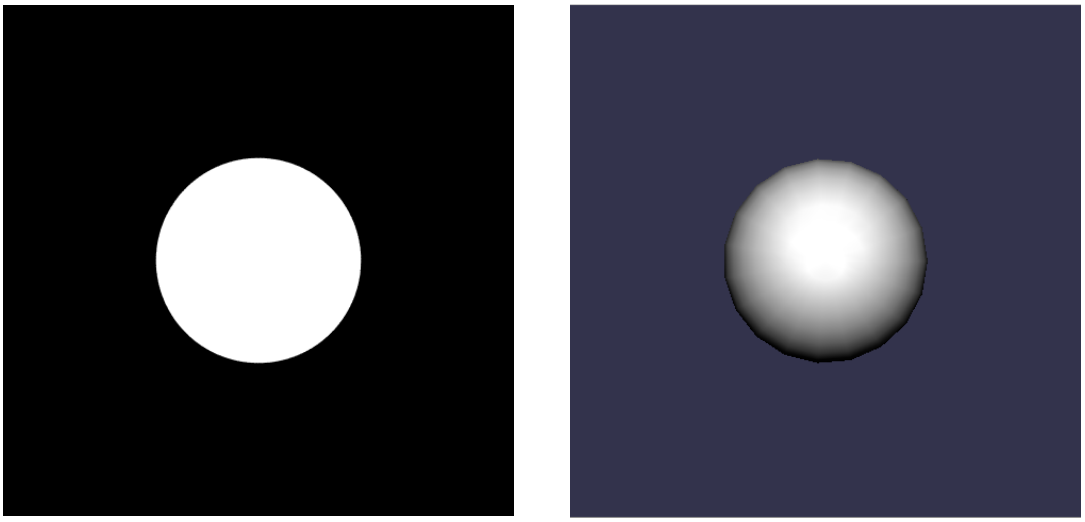

Figure 1 – Exemple de l'intersection rayon - sphère

### 1.2 La fonction d'intersection du carré

Concernant la fonction d'intersection du carré, je calcule le 't' à partir de la formule du cours. Les valeurs de 't' inférieures à un seuil prédéfini sont ignorées pour optimiser l'ombrage. Si la valeur

de 't' est positif, j'utilise cette valeur pour calculer l'intersection puis je détermine les coordonnées horizontales (u) et verticales (v) du point sur le carré. Ensuite, je vérifie que ces coordonnées ne sont pas inférieures à 0 (correspondant à uMin et vMin) et ne sont pas supérieures à 1 (correspondant à uMax et vMax), c'est à dire que ces coordonnées ne dépassent pas les limites du carré.

Concernant la boîte de Cornell, j'ai récrée la scène en remplaçant les carrés existants par des carrés seulement agrandis et translatés. J'ai également modifié leurs axes 'upVector' et rightVector' pour que la rotation de chaque carré corresponde à la scène originale.

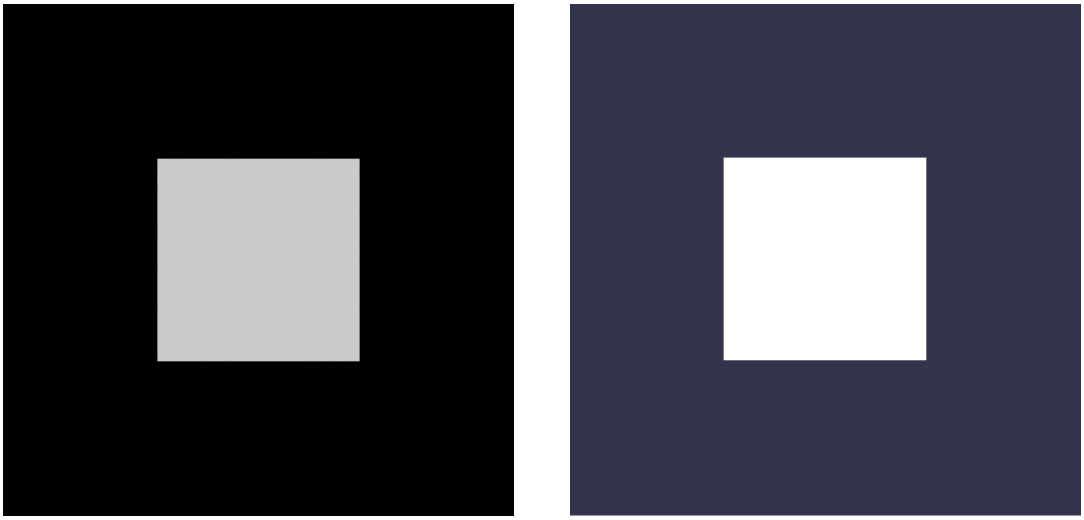

FIGURE 2 – Exemple de l'intersection rayon - carré

## 2. Les fonctions principales

### 2.1 La fonction : RaySceneIntersection computeIntersection(Ray const ray)

Dans cette fonction, je parcours chaque objet (Sphere, Square et Mesh) à l'aide d'une boucle puis j'assigne la valeur de l'intersection de l'objet à l'aide de la fonction 'intersect'. Si une intersection existe et que la valeur de 't' est strictement supérieure à certain seuil (pour prendre en compte l'ombrage) et que cette valeur est strictement inférieure à la valeur de l'intersection précédente 'result.t' (pour conserver l'intersection la plus proche), alors les valeurs de l'intersection obtenues sont attribuées à l'objet.

### 2.2 La fonction : Vec3 rayTraceRecursive( Ray ray , int NRemainingBounces )

Concernant cette fonction, j'attribue la valeur des intersections de la scène dans une variable, j'initialise quelques variables au début puis après avoir vérifié qu'une intersection existait, je vérifie le type de l'objet puis je vérifie le matériau de l'objet. Suivant le matériau de l'objet, j'appelle la fonction qui correspond à l'objet (Sphere, Square et Mesh) et au matériau (Basic, Phong, Réflexion).

### 2.3 La fonction : Vec3 rayTrace( Ray const rayStart )

Pour cette fonction, j'ai simplement crée un vecteur 3 (Vec3) que j'initialise à zéro. J'attribue ensuite la valeur retournée par la fonction rayTraceRecursive à ce vecteur, qui est ensuite retourné.

## 3. Exemples

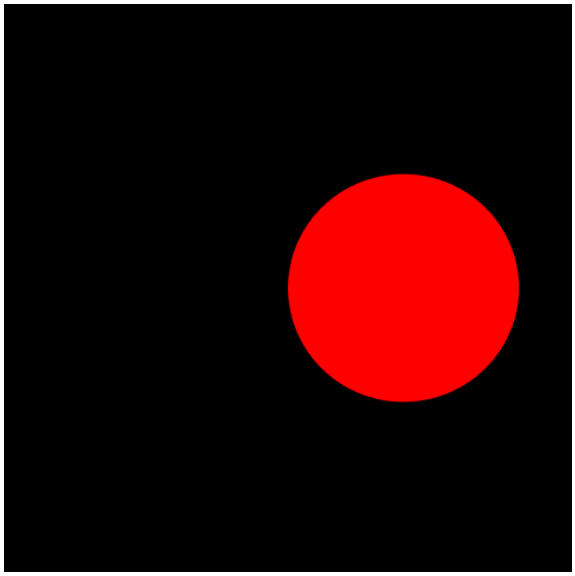

FIGURE 3 – Centre de la sphère déplacé en : (1.0, 0.0, 0.0)

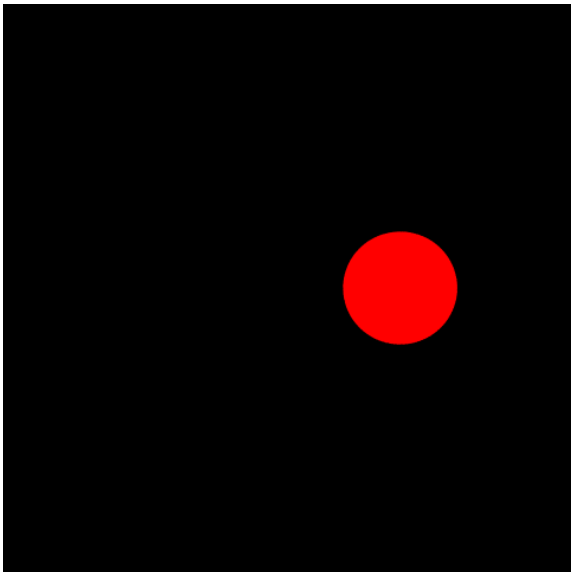

FIGURE 4 – Centre de la sphère déplacé en : (1.0, 0.0 , 0.) et dont le rayon est de 0.5

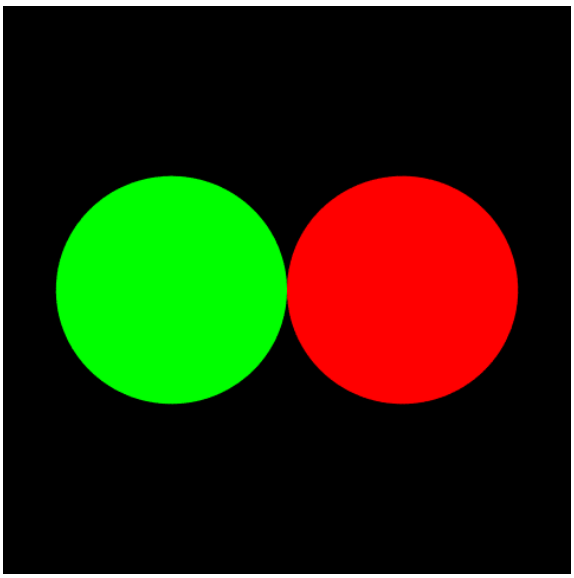

FIGURE 5 – Centre de la sphère rouge déplacé en : (1.0, 0.0 , 0.) et celui de la sphère verte déplacé en : (-1.0, 0.0 , 0.)

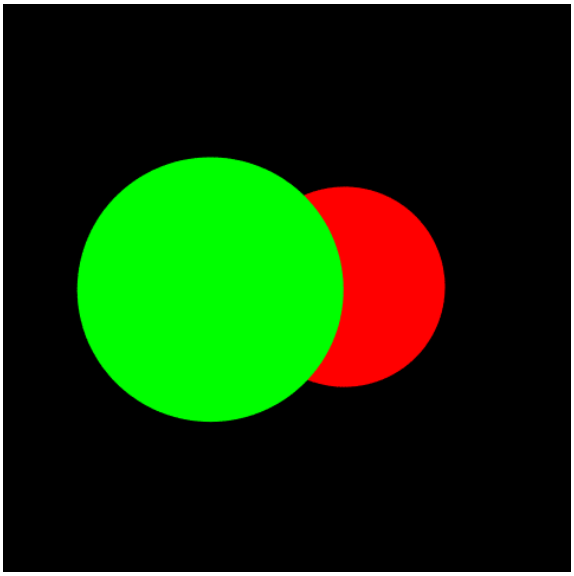

FIGURE 6 – Centre de la sphère rouge déplacé en : (1.0, 0.0 , 0.) et celui de la sphère verte déplacé en : (-1.0, 0.0 , 0.) (Point de vue différent)

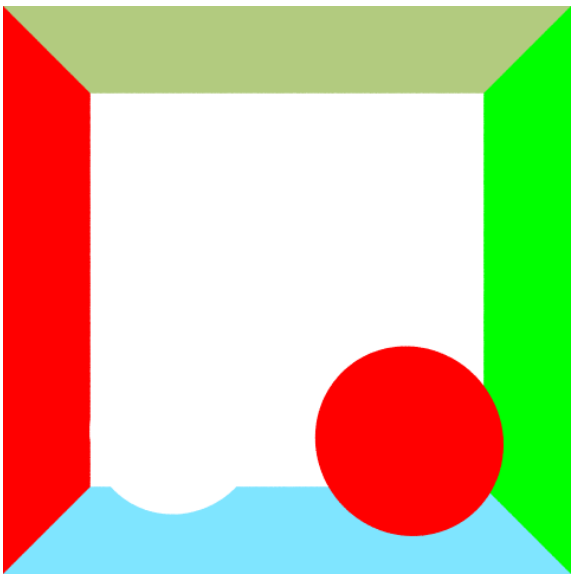

FIGURE 7 – Boîte de Cornell

# Phase 2

## 4. Le modèle de Phong

Concernant le modèle de Phong, j'ai créé une fonction pour calculer l'éclairage de Phong pour chaque objet (Sphere, Square et Mesh). En suivant les formules du cours et en normalisant correctement les vecteurs L (direction de la lumière), R (direction de la vue) et V (vecteur réfléchi), l'éclairage de Phong fonctionne correctement. J'ai généralisé le modèle d'éclairage en tenant compte de plusieurs sources lumineuses.

## 5. L'ombrage

### 5.1 Les ombres dures

Concernant l'ombrage dur, je vérifie d'abord que le type de la lumière est une lumière sphérique. Ensuite, je relance un nouveau rayon à partir de l'intersection avec une certaine marge, et je vérifie qu'une intersection existe entre la lumière et ce nouveau rayon. Si une intersection est présente et que la valeur de 't' pour cette intersection est inférieure à la distance du vecteur L, alors il y a de l'ombre, et un booléen est défini à 'true', sinon à 'false'. Si le booléen à pour valeur 'true', j'attribue la couleur noire au Vec3 color, qui est le vecteur retourné. Sinon, je calcule l'éclairage de Phong.

### 5.2 Les ombres douces

Concernant les ombres douces, je vérifie d'abord que le type de la lumière est une lumière carrée. Ensuite, j'itère sur un nombre préalablement défini d'échantillons (le nombre de rayons d'ombre). Pendant cette boucle, je parcours un carré de lumières en effectuant un échantillonnage aléatoire sur celui-ci. Je détermine la position du point, puis je réalise un calcul d'ombrage classique, avec la seule différence qu'une variable est incrémentée à chaque fois qu'une "ombre" est détectée.

Dans une autre variable, je calcule la moyenne entre la variable précédemment incrémentée et le nombre total d'échantillons. J'inverse ensuite cette valeur pour ne pas que le rendu soit noir. Enfin, je multiplie cette valeur aux composants spéculaire et diffuse du modèle de Phong.

## 6. Exemples

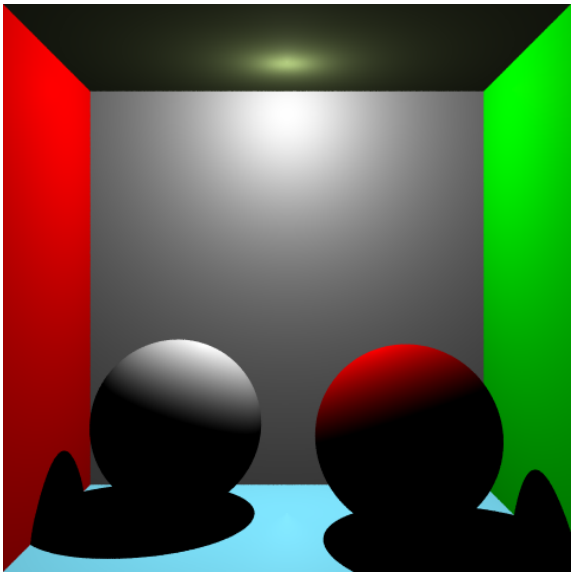

FIGURE 8 – Modèle de Phong (avec les ombres dures)

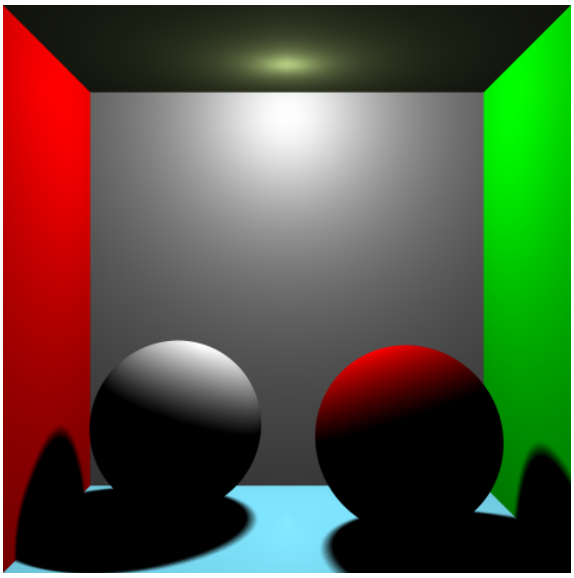

FIGURE 9 – Modèle de Phong (avec les ombres douces) (tailleCarre = 1, nbRayonsOmbres = 1)

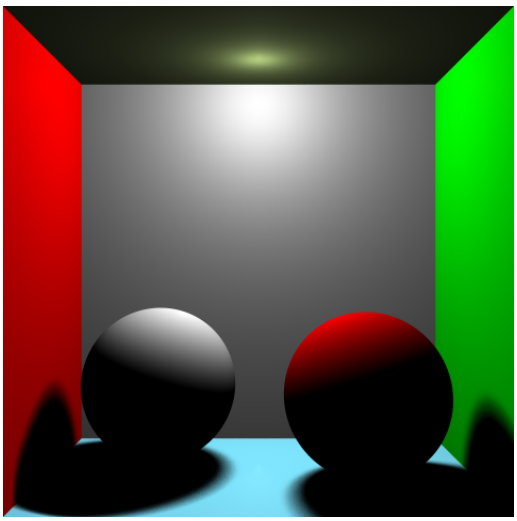

FIGURE 10 – Modèle de Phong (avec les ombres douces) (tailleCarre = 2, nbRayonsOmbres = 1)

# Phase 3

## 7. L'intersection Rayon-Triangle

Concernant l'intersection du triangle, j'ai rempli les fonctions principales du fichier Triangle.h. Pour la fonction getIntersection(Ray const  ray), je récupère la valeur retournée par la fonction getIntersectionPointWithSupportPlane() dans une variable 't'. Ensuite, je vérifie que le rayon n'est pas parallèle, que la valeur de 't' n'est pas négative puis je calcule les coordonnées barycentriques et enfin, je vérifie que ces coordonnées appartiennent au polygone. J'interpole ensuite les normales dont la formule est donnée par :

$$\text{Normale\_Interpolée} = w_0 \cdot \mathbf{n}_0 + w_1 \cdot \mathbf{n}_1 + w_2 \cdot \mathbf{n}_2$$

Dans le fichier Mesh.h, je parcours tous les triangles, je créer un triangle, je récupère la valeur de l'intersection puis je vérifie que la valeur de 't' est supérieure à 'seuilOmbre' qui est défini auparavant ce qui permet d'éliminer les petites valeurs de 't' pour l'ombrage. Ensuite, je vérifie que l'intersection existe et j'attribue enfin les valeurs de l'intersection à une variable créée précédemment, que je retourne.

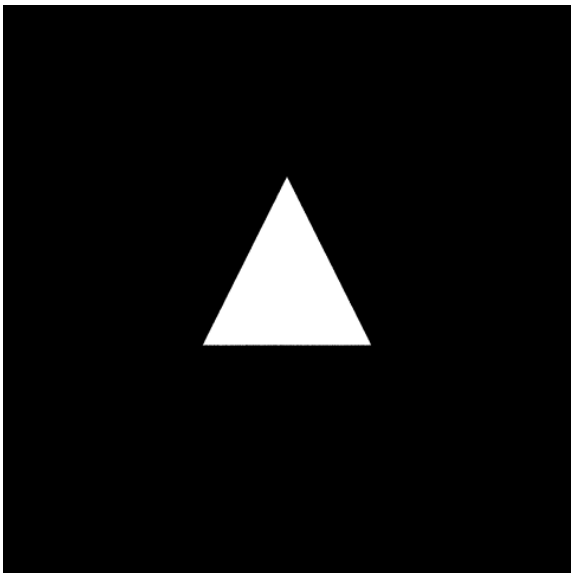

FIGURE 11 – Exemple de l'intersection rayon - triangle avec pour mesh : un triangle

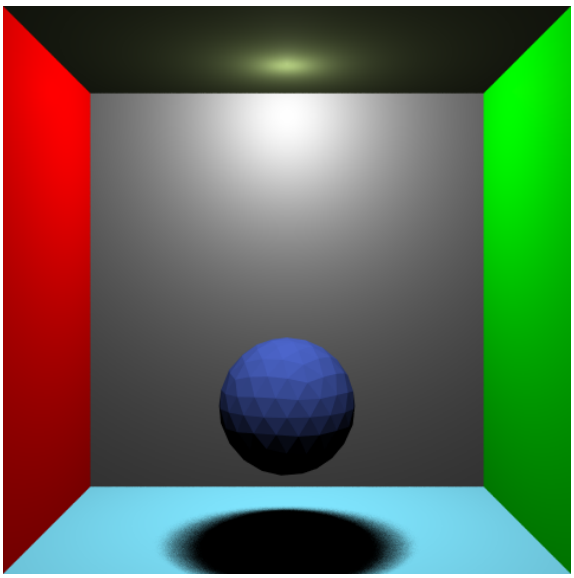

FIGURE 12 – Exemple de l'intersection rayon - triangle avec pour mesh : une sphère (avec les ombres douces) (les normales ne sont pas interpolées)

### 8. La réflexion

Concernant la réflexion, j'ai créé une fonction qui permet de calculer la réflexion pour une sphère mais également pour les autres objets (Square et Mesh). Je calcule le rayon réfléchi à l'aide de la formule :

$$\text{Rayon réfléchi} = \text{Incident} - 2 \cdot \text{dot}(N, \text{Incident}) \cdot N$$

Ensuite, je décrémente le nombre de rebonds restant de 1, puis j'appelle récursivement la fonction raytraceRecursive() jusqu'à ce que le nombre de rebonds soit égal à 0.

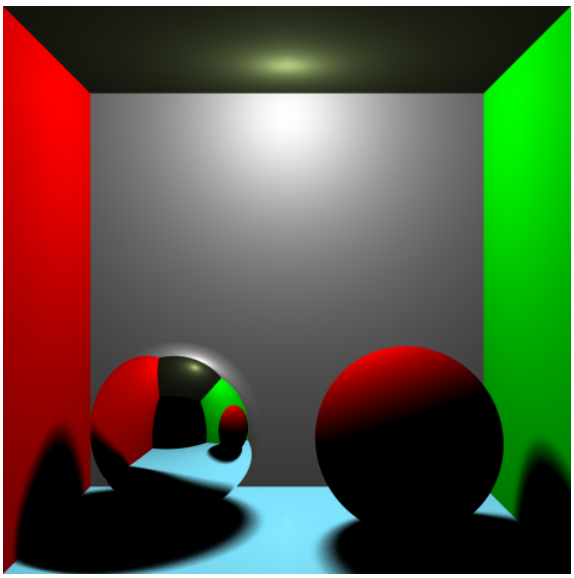

FIGURE 13 – Boîte de Cornell avec une sphère réfléchissante

## Phase 4

### 9. La réfraction

Concernant la réfraction, j'ai créé une fonction pour chaque objet (Sphere, Square et Mesh) qui comme la réfléxion sont récursives. J'ai appliqué l'équation :

$$\text{Rayon réfracté} = \frac{n_1}{n_2} \cdot \text{Incident} + \left( \frac{n_1}{n_2} \cdot \text{dot}(N, \text{Incident}) - \sqrt{1 - \left( \frac{n_1}{n_2} \right)^2 \cdot (1 - (C_1)^2)} \right) \cdot N \cdot \text{Incident}$$

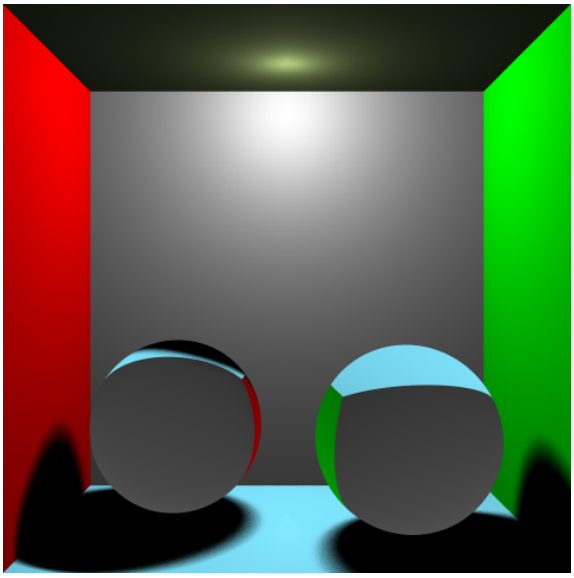

FIGURE 14 – Sphères réfractés avec un indice de réfraction (n2) de 3 (à gauche) et de 2 (à droite)

## 10. Le KD-Tree

Concernant le KD-Tree, je n'ai pas réussi à l'implémenter en entier. J'ai simplement réussi à implémenter une Axis Aligned Bounding Box (AABB) qui est une composante essentielle du KD-Tree car chaque noeud du KD-Tree est une AABB.

Pour implémenter l'AABB, j'ai créer une classe qui contient la méthode pour trouver l'intersection. J'ai également créé une fonction qui permet de faire marcher cette AABB.

En utilisant l'AABB, la différence de temps pour charger des maillages est considérable. Par exemple, pour un maillage comme celui de Suzanne qui mettait 30 minutes à charger sans l'AABB, le maillage met 40 secondes à charger avec l'AABB.

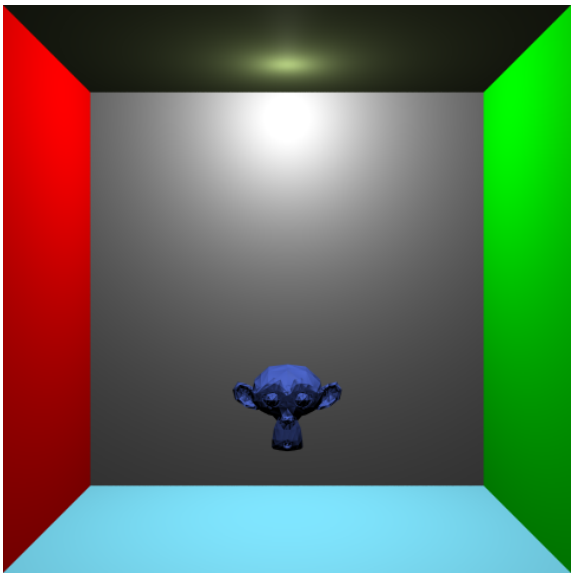

FIGURE 15 – Maillage "Suzanne" chargé grâce à l'AABB

# Bonus

## 11. Le modèle d'éclairage de Blinn-Phong

Pour mettre en oeuvre le modèle d'éclairage de Blinn-Phong, j'ai simplement dû remplacer le calcul de $R \cdot V$ dans la composante spéculaire par $N \cdot H$ avec $H = \frac{L+V}{\|L+V\|}$. Ce vecteur représente le half vector ou "demi vecteur" en français.

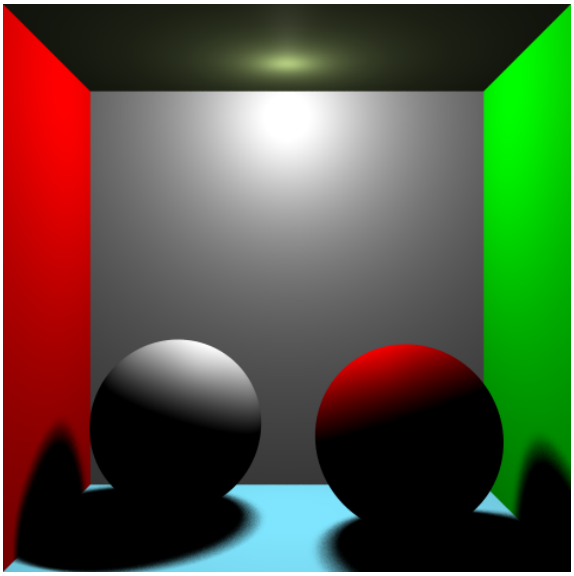

FIGURE 16 – Exemple de l'éclairage de Blinn-Phong sur tous les objets de la scène

## 12. L'effet de Rim / Fresnel

L'effet de Rim permet de donner l'impression qu'une lumière est derrière l'objet ce qui donne un effet de surbrillance sur les contours d'un objet. Pour créer cet effet, je calcule le facteur de Fresnel : $F = (1 - \mathbf{v} \cdot \mathbf{n})$. Je contrôle la puissance du halo avec la fonction smoothstep puis je multiplie la valeur obtenue par une couleur pour colorer le halo. J'ajoute cette valeur au calcul de l'éclairage de Blinn-Phong. J'ai également créé une fonction qui implémente ce type d'effet pour chaque objet.

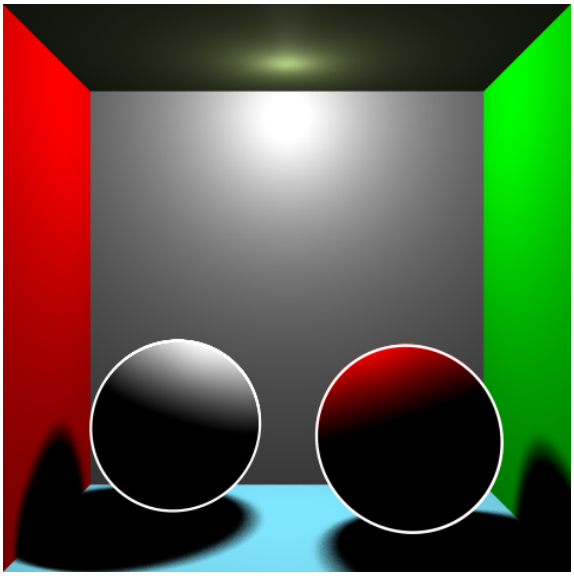

FIGURE 17 – Exemple de l'effet de Rim sur les deux sphères

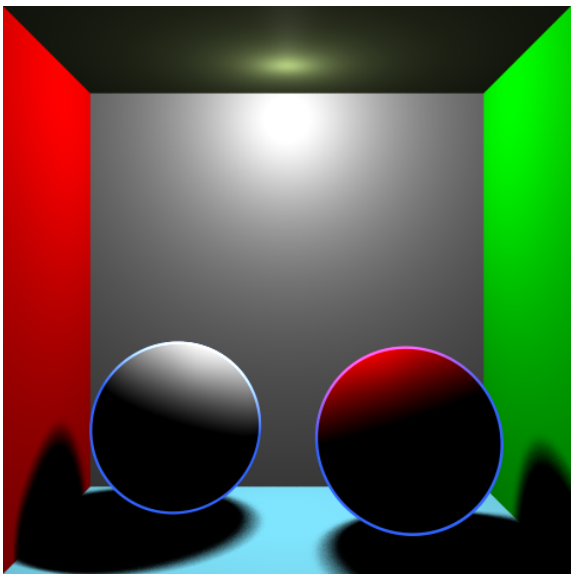

FIGURE 18 – Exemple de l'effet de Rim sur les deux sphères avec une couleur différente

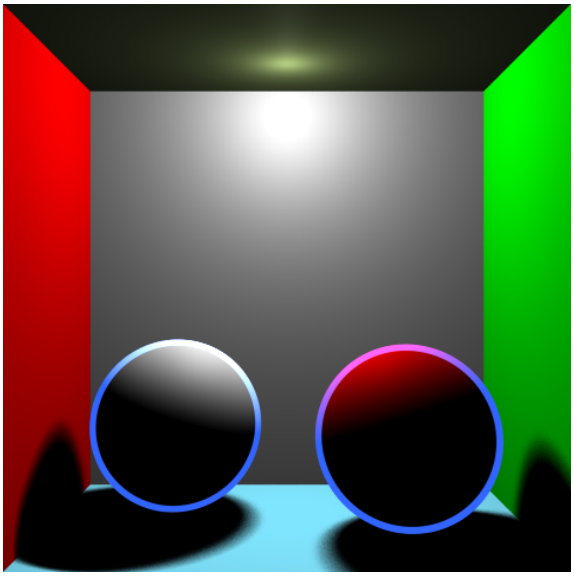

FIGURE 19 – Exemple de l'effet de Rim sur les deux sphères avec une intensité différente

## 13. Le Toon Shading

Le principe du Toon Shading est d'avoir un effet "dessin animé", "cartoon". Pour faire cela, j'ai utilisé l'effet de Rim pour accentuer l'effet "cartoon" et pour la couleur, j'ai utilisé une rampe de couleurs pour avoir une démarcation bien précise entre chaque couleur. J'ai utilisé le modèle de Blinn-Phong en changeant seulement la composante diffuse.

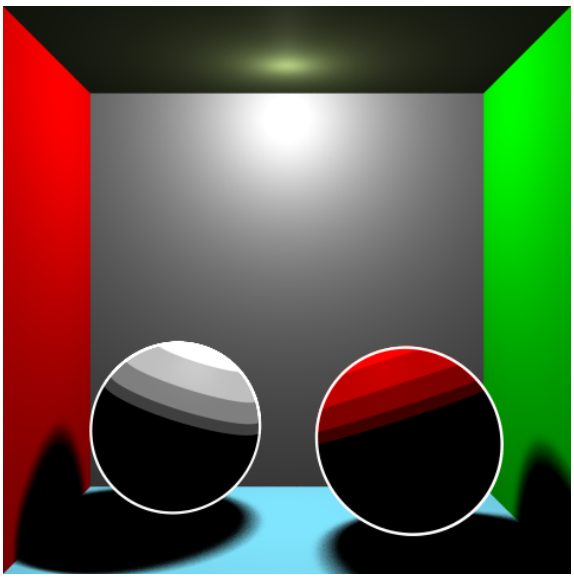

FIGURE 20 – Exemple du Toon Shading sur les deux sphères avec une rampe composée de 4 couleurs

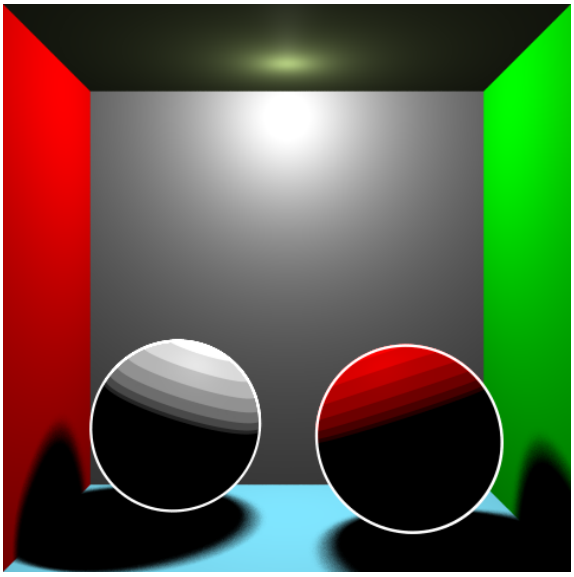

FIGURE 21 – Exemple du Toon Shading sur les deux sphères avec une rampe composée de 7 couleurs

## 14. PBR : le BRDF de Cook-Torrance

Ayant vu le PBR en cours, j'ai voulu l'implémenter dans mon raytracer. Pour cela, j'ai repris les principales fonctions d'un précédent TP sur les textures que j'ai adapté pour le raytracer.

La formule du BRDF de Cook-Torrance est : BRDF = diffuse + spéculaire avec la partie diffuse qui est égale à :

$$\text{diffuse} = \frac{\pi}{c} \quad \text{(Lambertian Diffuse)}$$

et la partie spéculaire qui est égale à :

$$\text{spéculaire} = \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)}$$

Avec :
D, la fonction de distribution normale
F, la fonction de Fresnel
G, la fonction géométrique

Concernant la fonction de distribution normale, j'ai choisi la "Trowbridge-Reitz GGX". Concernant la fonction de Fresnel, j'ai utilisé l'approximation de Schlick qui prend en compte les indices de réfraction n1 et n2 et pour la fonction géométrique, j'ai utilisé la méthode de Smith qui utilise la "Schlick-GGX". Toutes ces fonctions sont dans le fichier PBR.h

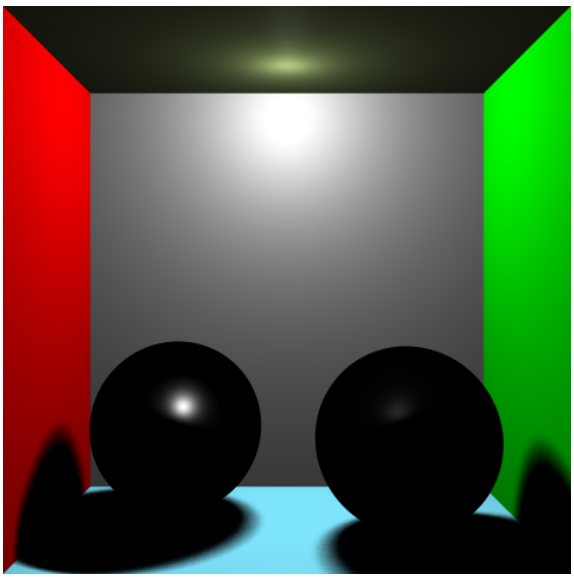

FIGURE 22 – Exemple du PBR sur les sphères : Metallic : 0.9, Roughness : 0.2

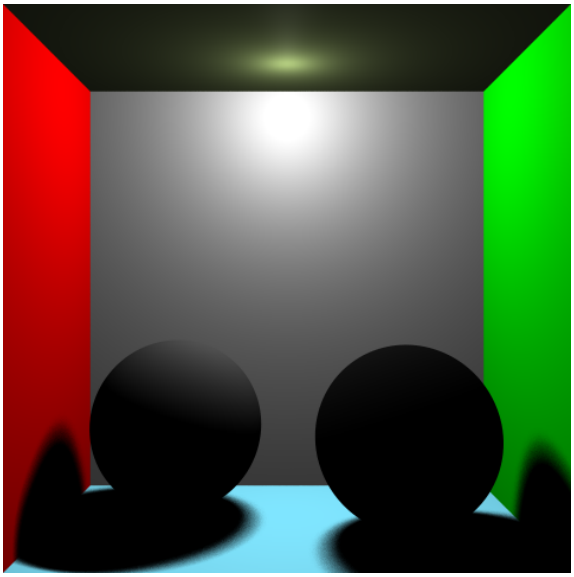

FIGURE 23 – Exemple du PBR sur les sphères : Metallic : 1.0, Roughness : 1.0

On peut remarquer que les rendus paraissent très sombres mais les effets de rugosité et de métallique fonctionnent correctement.

## 15. Le Texture Mapping

J'ai implémenté un système de texture seulement pour la sphère et le carré.

Concernant le carré, c'était le plus simple car il y avait déjà les composants u et v. J'ai créé le fichier "Texture.h" qui permet d'appliquer une texture grâce aux coordonnées u et v que j'ai créé moi-même ou que l'on importe d'un fichier.

En ce qui concerne la sphère, j'ai ajouté les attributs u et v au fichier "Sphere.h" et les ai calculés comme ceci :

$$u = 1 - \frac{\text{atan2}(-\text{intersection.normal}[2], \text{intersection.normal}[0]) + \pi}{2\pi}$$
$$v = \frac{\text{acos}(-\text{intersection.normal}[1]) + \frac{\pi}{2}}{\pi}$$

Ce qui donne :

$$u = 1 - \frac{\phi + \pi}{2\pi}$$
$$v = \frac{\theta + \frac{\pi}{2}}{\pi}$$

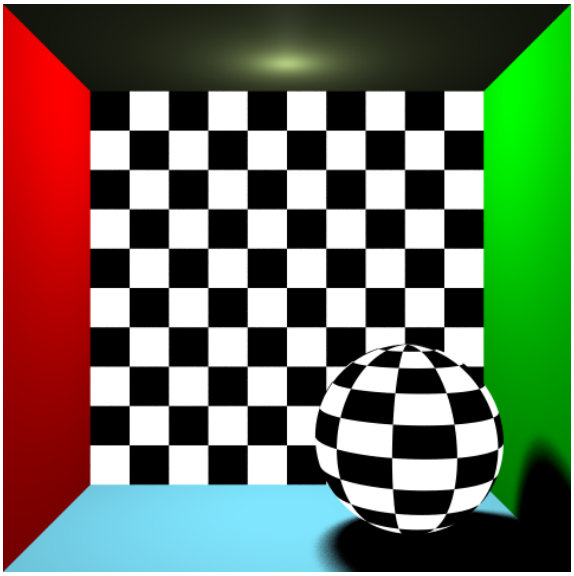

FIGURE 24 – Texture "motif à carreaux" sur la sphère et le carré

On peut remarquer que la texture est correctement appliquée sur le carré ainsi que sur la sphère.

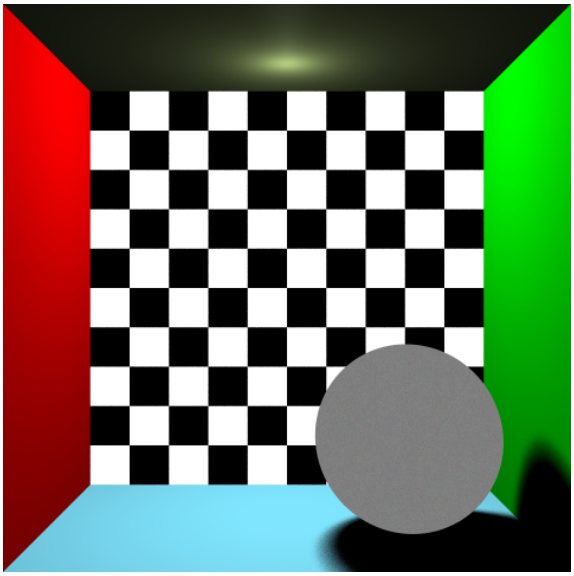

FIGURE 25 – Texture "bruit blanc" sur la sphère

## Ce que j'aurais voulu ajouter

J'aurais aimé ajouter l'effet de bump mapping et/ou de normal mapping ainsi que l'effet de profondeur (Depth of Field).