MOS 2.2 : Informatique graphique
Programation d'un raytracer

Théo Cavignac

23 mars 2020

Introduction

Le *raytracing* ou «lancer de rayon» est une technique de rendu de scène 3D qui consiste à simuler le trajet de rayons lumineux hypothètiques arrivant sur le capteur d'une camera virtuelle. Cette technique se basé sur l'optique géometrique permet d'obtenir des images très réaliste contenant des effets de lumière variés en suivant une méthode assez simple. Elle s'oppose au *rasterizing* qui est une technique plus légère consistant à projeter des triangles sur le capteur de la caméra virtuelle et à faire des approximation pour calculer l'éclairage.

Ce document rend compte de la réalisation d'un code de *raytracing* en C++ au cours du MOS 2.2. Il est rédigé en trois parties. La première partie présente les bases géométriques utilisé et les résultats obtenu au cours des premières étapes de dévellopement. La partie 2 présente les effets de lumières possibles grace aux lancer de rayons stochastiques. La partie 3 présente l'ajout des maillages comme primitives, les structures d'accélérations associé et l'implémentation du support des textures. En annexes se trouvent d'autres rendu produit avec des paramètres plus lourds (et donc des résultats plus propres).

1 Base du raytracing

Les premières étapes (tags step1 à steps_5_6) du dévellopement consistent à impléménter la capacité à rendre des sphères de couleurs ombré.

La méthode consiste à lancer des rayons de lumière pour chaque pixel dont l'origine est le centre de la caméra et la direction est donnée par la position du pixel sur le «capteur».

On commence par détecter une sphère placé devant la caméra virtuelle. Pour cela on implémente une fonction permettant de déterminer l'existence d'une intersection entre une sphère et un rayon. En présence d'une telle intersection on rend le pixel correspondant blanc. Sinon on rend le pixel noir. On obtient la figure 1.

On va ensuite utiliser la position de l'intersection pour calculer la luminosité réfléchie par la surface vers la camera. Pour cela on considère que la surface est blanche et parfaitement diffusive, ce qui correspond à une surface ayant une microstructure très rugeuse comme du plâtre. La formule correspond est implémenté en C++ dans la figure 2.

Cette formule exprime deux fait important :

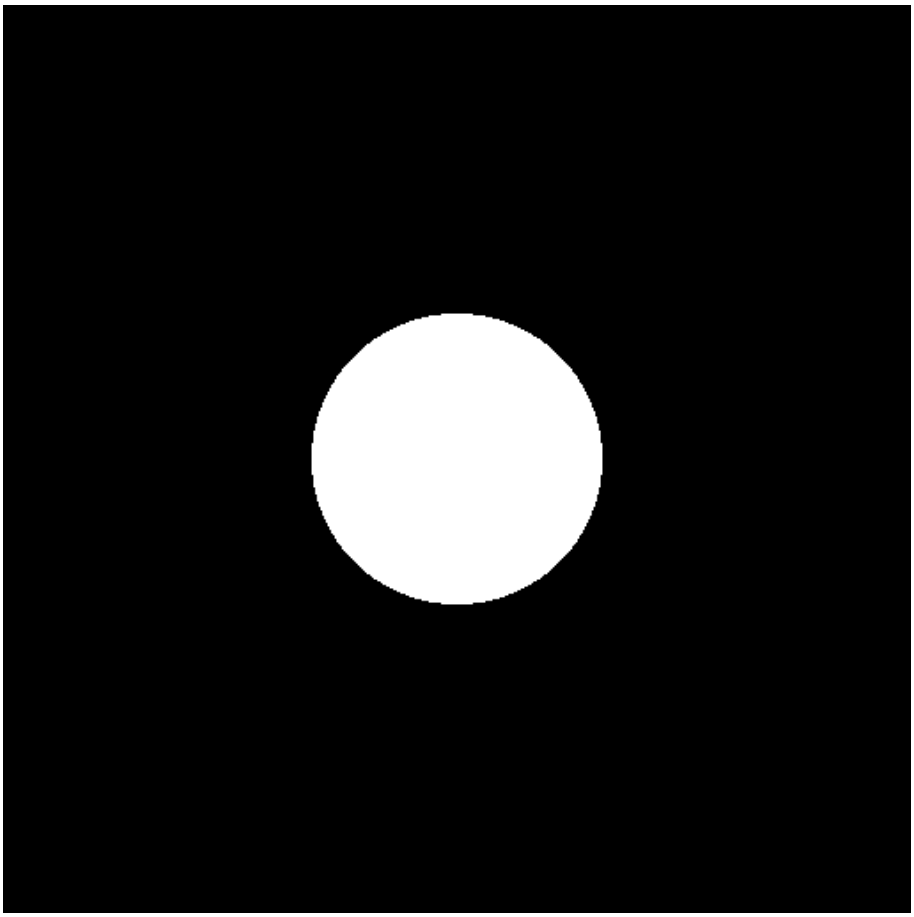

FIGURE 1 – Détection d'une sphère devant la camera

— si le rayon de lumière provenant de la source eclaire la surface de façon tangeante, aucune lumière n'est réfléchi, si le rayon est normal à la surface un maximum de lumière est réflechi
— l'intensité rééfléchie ne dépend pas de la direction du rayon allant vers la caméra car la lumière est réflechie de façon homogène dans toutes les directions (sauf si le rayon est derrière la surface)

Le résultat est visible dans la figure 3

```
// s est une instance de Sphre et r une instance de Ray
// la méthode intersection retourne un vecteur correspond
// à la position de l'intersection
Vec p = s.intersection(r);

// n est le vecteur normal à la surface au point
// d'intersection
Vec n = (p - s.origin()).normalized();

Vec vl = light - p;

// I est l'intensité de lumière reçu pas la caméra
double I = I0 * std::max(0.0, n.dot(vl.normalized()))
    / vl.norm_sq();
```

FIGURE 2 – Calcul de l'intensité reçu par la caméra après réflexion sur la surface de la sphère

L'étape suivante consiste à refactoriser le code pour :
— faire le rendu de multiples spheres
— attribuer une couleur au spheres

Pour cela on a définit une classe `Sphere` qui est définie par un centre, un rayon et un albedo (un vecteur 3D correspondant à la couleur en format RGB). On a également une classe `Scene` qui contient plusieurs instances de `Sphere` ainsi qu'une instance de `Light`. Cette classe est chargé de calculer la couleur d'un rayon donné avec une méthode `get_color`. Cette méthode restera au centre du programme tout au long de son dévellopement.

On définie une scène dans laquelle cinqs sphères de rayon 1000 formeront des murs encadrant un sphère de rayon 10 au centre de la sphère. Pour supporter les couleurs on réutilise la formule donné dans la figure 2 mais multipliant `I` par l'albedo de la sphère pour obtenir le vecteur couleur du rayon final.

On obtient la figure 4.

On applique aussi pour la première fois un transformation gamma inverse qui permet de rendre compte de la non linéarité de l'œil humain dans la perception de la lumière ce qui a pour effet de diminuer les contrastes de l'image.

Enfin on ajoute les ombres en cherchant si un objet se trouve sur le chemin entre la source de lumière et le point d'intersection entre la surface et le rayon. Si un tel objet existe le point en question n'est pas (directement) éclairé et apparait donc noir. On obtient la figure 5.

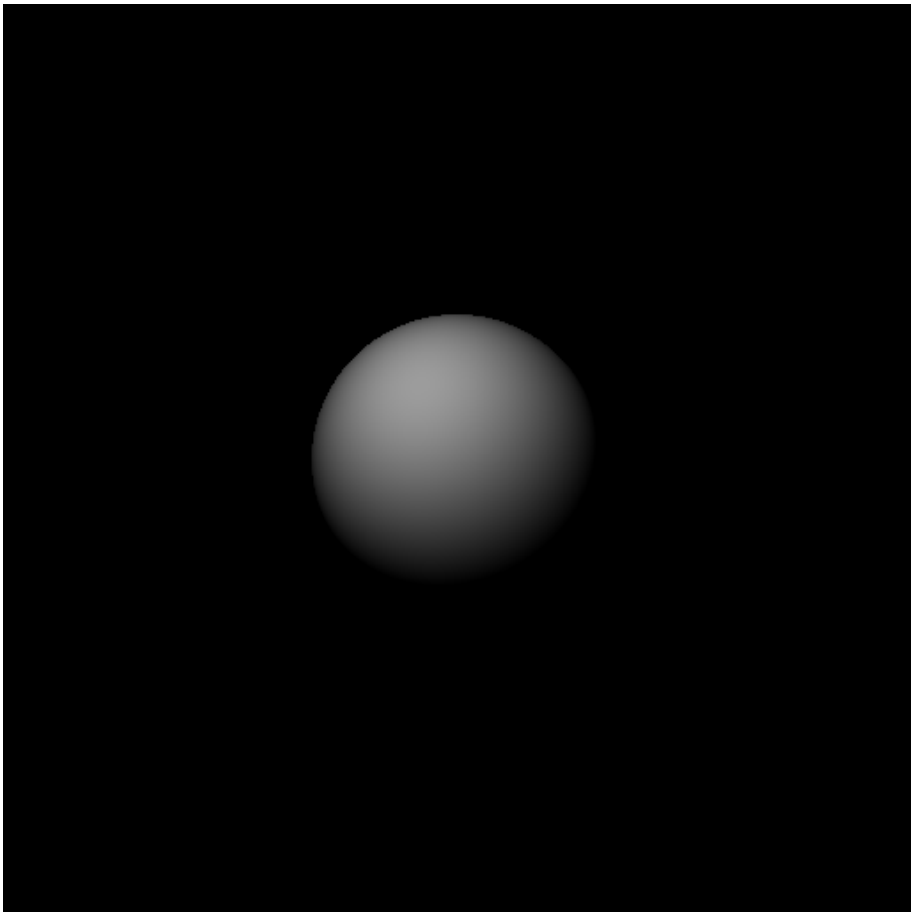

FIGURE 3 – Sphere diffusive ombré

Plus tard l'albedo de `Sphere` est remplacé par une instance d'une classe `Material` qui servira de classe mère pour différent type de matériaux correspondant à différentes BRDF. On ajoute notement un materiaux réfléchissant (d'abord représenté par un drapeau booléen `mirror` dans la classe `Material` puis par une classe fille spécialisé `Reflective` cf. step7) dont la BRDF consiste en un dirac dans la direction du rayon réfléchie, c'est à dire que toute la lumière reçu est réfléchie dans la même direction.

Cet effet est le premier qui traduit le phénomène d'éclairage indirecte que l'on généralise dans l'étape suivante. On obitient la figure 6.

2 Effets de lumière réaliste

Dans cette partie (tags steps_5_6+ à step7) on introduit le lancer de rayon stochastique pour réaliser divers effets de lumière réalistes tel que l'éclairage indirecte qui permet les ombres douces, la transparence, les sources étendues et autres.

Le premier effet est la transparence qui consiste à relancer un rayon depuis le premier point d'intersection dans une direction donnée par la loi de Descartes. Comme il y a dans la plupart des cas un rayon réfléchie et un rayon refracté on choisi aléatoire l'une de ces deux

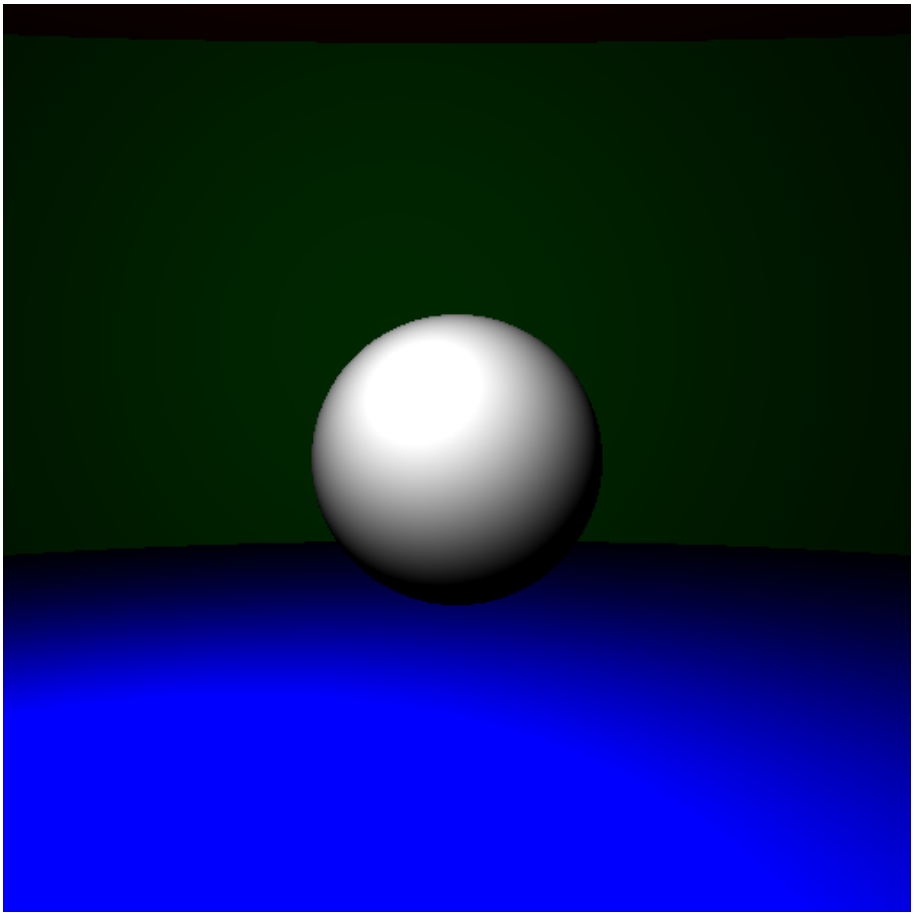

FIGURE 4 – Support d'une scene contentant plusieurs spheres coloré

directions pour lancer le nouveau rayon sans avoir une croissance exponentielle du nombre de rayon. La probabilité de chaque rayon est donnée par les facteurs de reflexion et de transmission en puissance approximé par la formule de shlick. Pour lisser cet effet aléatoire on lance donc plusieurs rayons pour chaque pixel et on moyenne les intensités obtenus. Pour lancer le nouveau rayon `get_color` s'appelle récursivement avec une profondeur de récursion finie. A chaque fois que l'intersection tombe sur un surface transparente, un rayon réfléchie ou réfracté est relancé et ainsi on construit la lumière finale vue par la caméra. L'aspect stochastique fait apparaitre un peu de bruit dans l'image qui est lissé en lançant plus de rayons par pixel. On peut voir cet effet dans la figure 7.

On ajoute ensuite l'éclairage indirecte. Cela consiste à relancer un rayon également pour les surfaces diffuse et a utiliser la couleur de ce rayon comme éclairage supplémentaire (en plus de l'éclairage directe). C'est ici que la création des classes `Material` commence à faire sens car elle permet d'unifier la réflexion totale des surfaces réfléchissantes et la réflexion stochastique des surfaces diffuse au travers d'une méthode `reflex_dir` qui associe à un rayon émergent et une normale, un rayon incident. Dans le cas de la réflexion totale cette méthode est déterministe. En revanche dans le cas des surfaces diffuse elle est stochastique et renvoie une direction selon un distribution qui permet toute les directions de l'hémisphère en favorisant les directions proches de la normale. Cette fonction ne dépend pas du rayon

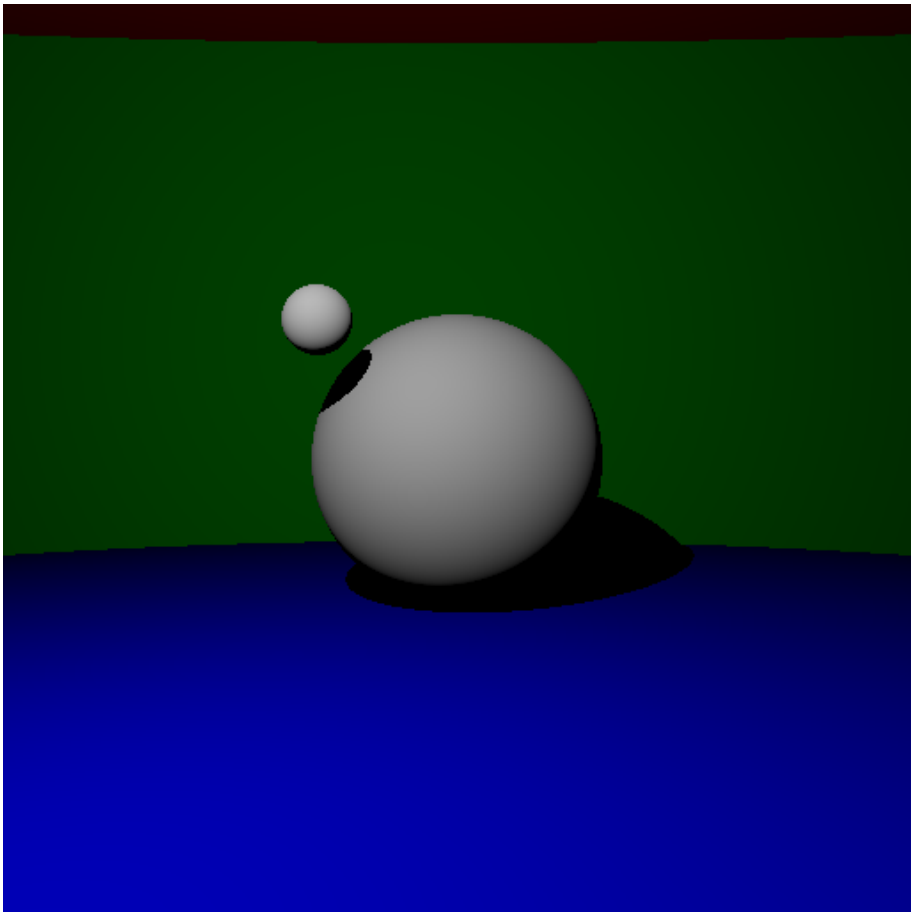

FIGURE 5 – Sphères avec des ombres portés

émergent car la surface est parfaitement diffuse. La figure **??** montre la fonction `random_cos` qui permet de générer une direction et fourni également la probabilité de cette direction particulière qui sert de facteur dans la méthode de Monte-Carlo qui se cache derrière cette méthode.

Concretement l'effet de cette modification est que les ombres ne sont pas totalement noires, car elles reçoivent un peu de lumière réfléchie sur d'autres objets.

Un autre effet cousin de celui-ci est d'étendre la source de lumière qui était jusqu'ici ponctuelle. On considère maintenant une sphère qui emet de façon homogène une lumière blanche. Cette sphère est représenté par la classe `Light` qui contient maintenant un rayon.

Lorsque l'on calcule l'éclairage directe on utilise `random_cos` pour choisir un point sur la surface de la sphere lumière. Ce point sert ensuite de source ponctuelle pour le point courrant. La luminausité du point correspond à la luminosité nominale de la source pondéré par le cosinus de l'angle entre la normale au point sur la sphère lumineuse et la direction vers l'objet éclairé.

Le résultat de ces deux ajouts est visible dans la figure 8.

Enfin les derniers effets ajouté utilisant le lancer de rayons stochastique sont l'effet de profondeur de champ qui consiste a simuler l'effet de l'optique d'une caméra sur a netteté des différents plan de l'image et l'anti-crénelage.

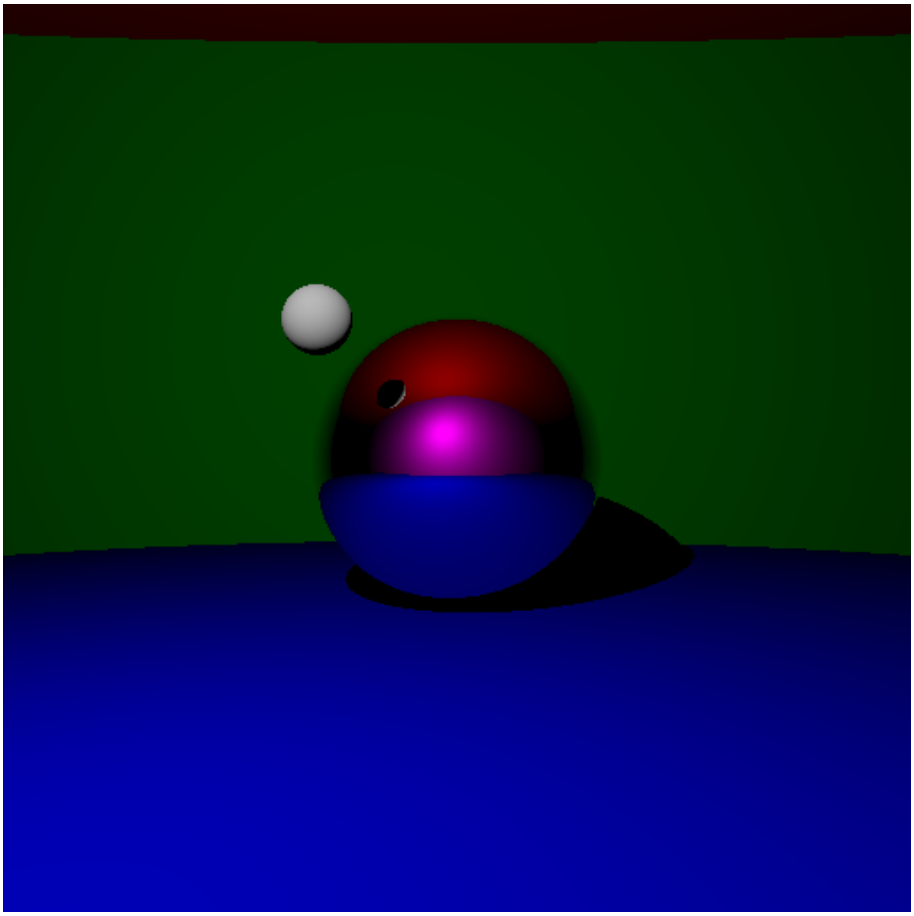

FIGURE 6 – Introduction des surfaces réfléchissantes

Ces deux effets consistent à perturber le rayon initiale. La profondeur de champ est obtenu en perturbant l'origine du rayon, mais en choisisant la direction de sorte quelle que soit l'origine les rayons converge au même point d'un plan parallèle au plan du capteur. Ce plan est définit par la longeur de champ que l'on fixe pour la figure **??** à 55 unité, soit le plan dans lequel se trouve la sphère miroire.

L'anti-crénelage consiste à perturbé légerement la direction du rayon de sorte qu'il ne passe pas systématiquement au même endroit du pixel. On obtient donc des transitions plus douces entre les surfaces qui font disparaitres les pixels.

3 Maillage et structures d'accélération

Dans cette partie on ajoute le support des maillages ainsi qu'un certain nombre d'améliorations du code, comprenant des refactorisations facilitant la maintenabilité et le déboguage.

Notamment l'ajout d'un parser de fichier de configuration modelé sur le parser de fichier OBJ permet de spécifier tout les paramètres de la simulation sans recompiler le projet. De cette façon il était plus simple de faire varier certains paramètres pour comprendre la nature des bogues rencontré au cours de l'implémentation des maillages. Les paramètres modifiables

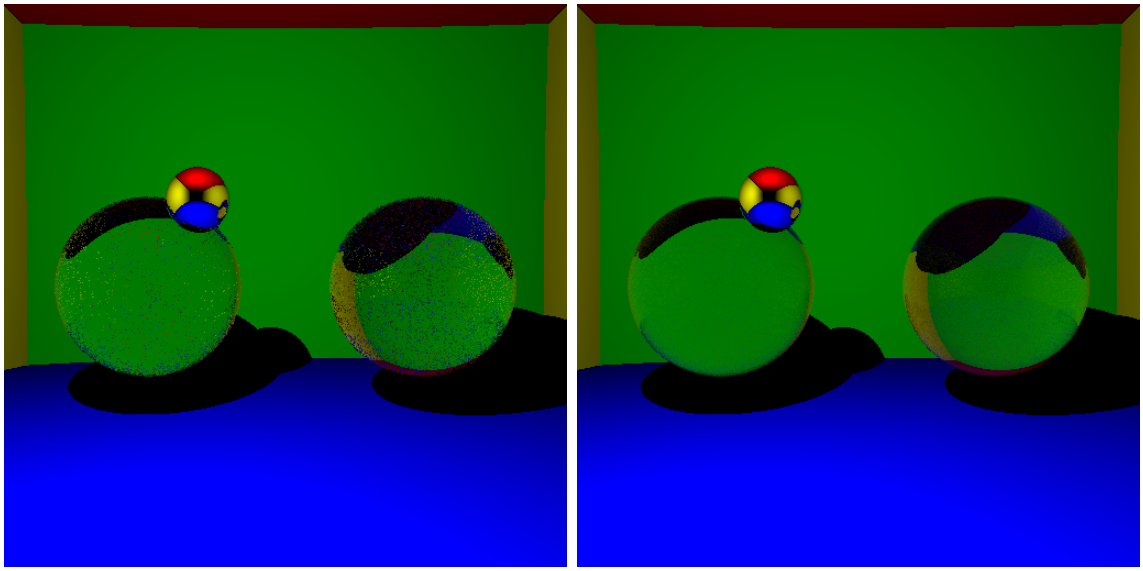

FIGURE 7 – Rendu avec une sphère transparente en lançant 2 rayons par pixel (droite) et 30 rayons par pixel (gauche)

comprenent entre autres :

— la taille de l'image
— le nombre de rayon lancé par pixel
— la position et l'orientation de la caméra
— la composition de la scène

Un exemple de fichier de configuration est présenté en annexe dans la figure 11.

En ce qui concerne l'implémentation des maillage, la première étape est de rendre un maillage simple, chargé depuis un fichier OBJ avec un le parser fourni en cours.

Le rendu basique consite à parcourir la liste des triangles et à déterminer une éventuelle intersection. L'intersection la plus proche de la caméra est alors considéré comme l'intersection avec le maillage.

L'intersection avec un triangle consiste à déterminer une intersection P avec le plan incluant le triangle, par la donnée d'un sommet du triangle est de la normale géométrique du triangle déterminé par produit vectoriel de deux cotés du triangle. L'orientation de cette normale est détérminé d'après les normales d'ombrages fournies par le ficher OBJ. Une fois l'intersection dans le plan définie on calcule les coordonées barycentriques en résolvant le système vectoriel 1. Si les inconnues t, u et v sont dans l'intervale $[0, 1]$ alors le point P est compris dans le trangle IJK. Ces paramètres peuvent ensuite être utilisé plus tard pour extrapoler les coordonées UV pour la texture et la normale d'ombrage.

$$\begin{cases} t + u + v & = & 1 \\ tI + uJ + vK & = & P \end{cases} \tag{1}$$

Cet ajout permet d'afficher des maillages simples comme le singe de la figure 9. On peut observer dans cette figure l'effet de l'interpolation de normales sur un même maillage.

On ajoute ensuite une structure d'accélération qui permet de discriminer plus rapidement quels triangles doivent être testé de sorte que le calcul de l'intersection entre un maillage

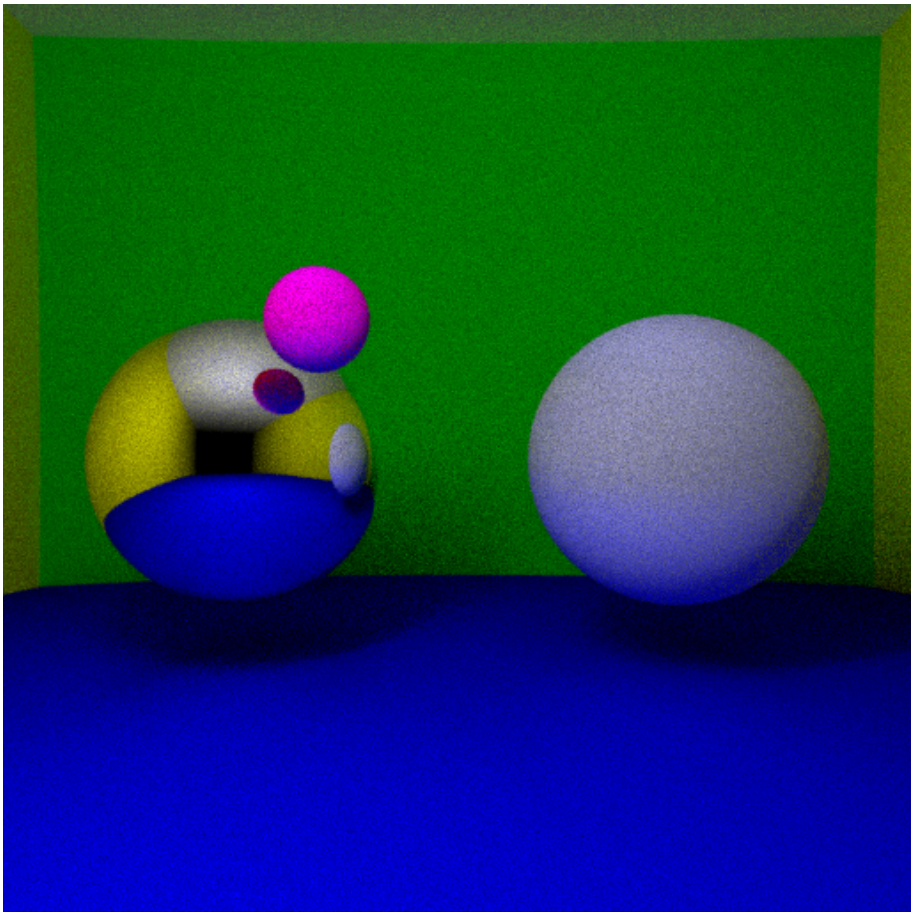

FIGURE 8 – Ajout de sources étendu et d'éclairage indirecte

et un rayon ne soit plus une fonction linéaire du nombre de triangle mais une fonction logarithmique. Pour cela on divise au préalable les triangles du maillages en deux ensembles disjoints ayant des boites englobantes non confondu. Puis pour chaque sous ensemble on répète l'opération de partition jusqu'à avoir des ensembles de un ou deux triangles seulement. Le calcul de l'intersection rayon/maillage consiste alors à intersecter le rayon avec les boites englobantes et à ne tester que les triangles qui sont dans les plus petites boites intersectées. On utilise pour cela un algorithme récursif qui déterminer les boites intersecté en descendant et qui construit l'ensemble des triangles à tester en remontant.

Pour l'opération de partition on cherche à avoir des boites équilibrés. On cherche donc la direction (aligné sur les axes) qui offre la plus grande variance dans la position des triangles puis on partitione les triangles autour de la médiane des positions sur cet axe. Ainsi on divise à chaque étape le nombre de triangle par boites par deux. Cette opération est possible en temps linéaire (à chaque itération, donc avec une complexité en $O(n \log n)$ pour le maillage totale, n étant le nombre de triangle) grace au calcule de variance (opération linéaire) et à la fonction standard `std::nth_element` qui permet de faire la partition et le calcul de la médiane en même temps et en temps linéaire (en moyenne, grâce a une variation de l'algorithme *quickselect*).

Enfin on ajoute le support des textures. Cela consiste à créer à la volé un materiaux

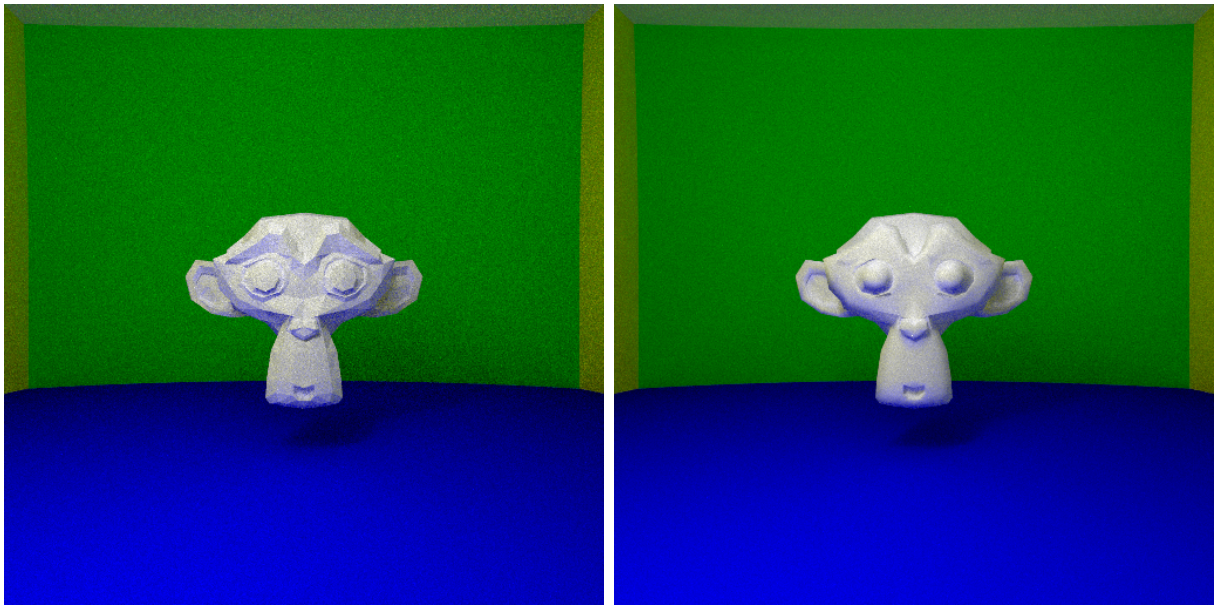

FIGURE 9 – Rendu de maillages simple avec normales géométriques (gauche) et normales interpolé (droite)

diffusif pour l'intersection avec le triangle dont la couleur corespond à celle d'un pixel de l'image texture. La position du pixel est interpolé d'après les coordonées UV associé à chaque sommets du triangle. On obtient alors la figure 10.

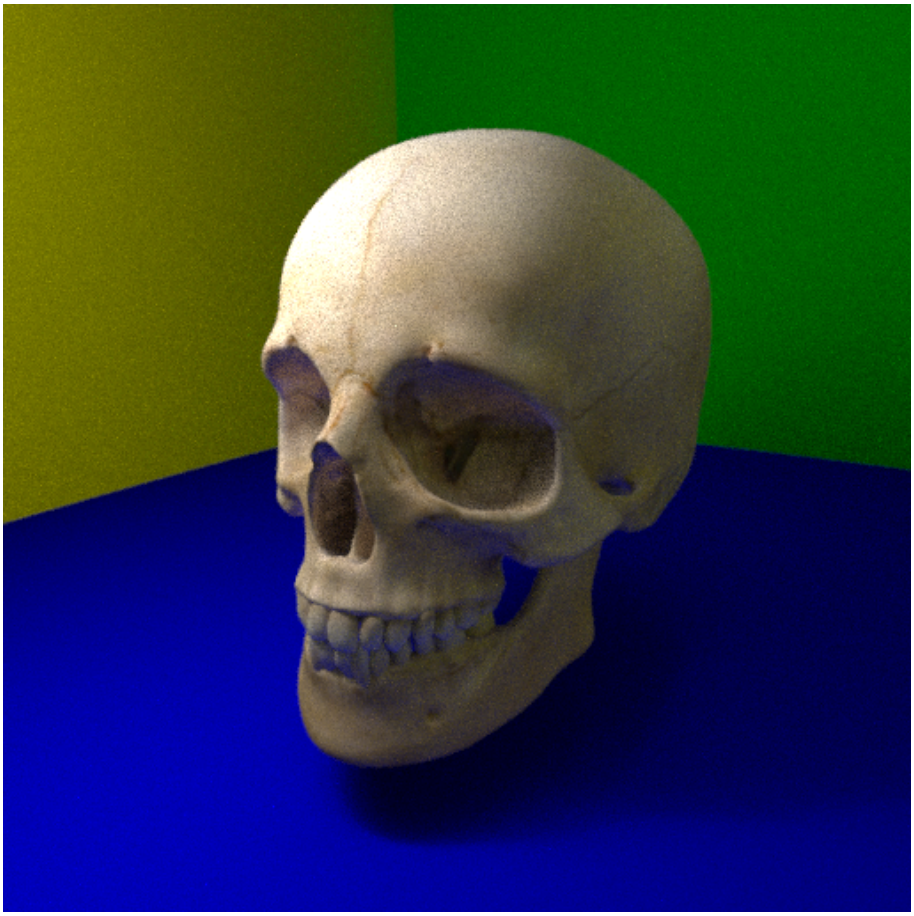

FIGURE 10 – Rendu d'un maillage complexe (40728 faces dont 39288 quads soit 80016 triangles) avec une texture 1024×1024

Annexes

```
## Rendering config
ray_number 20
size 512 512
field_depth 55.0
focal_opening 0.0
fov 1.0472
recursive_depth 5
# antialiasing level; level 0: no antialiasing,
# level 10: too strong
antialiasing 5
# camera x y z dx dy dz;
# (x, y, z): camera center,
# (dx, dy, dz): camera direction
camera 20 10 55 -0.6 -0.3 -1
light_intensity 5e8
## Scene setup
# sphere mat_id radius x y z
# ceiling (0: white)
sphere 0 940 0 1000 0
# floor (1: blue)
sphere 1 985 0 -1000 0
# front wall (3: green)
sphere 3 940 0 0 -1000
# left and right wall (4: yellow)
sphere 4 940 -1000 0 0
sphere 4 940 1000 0 0
# tmesh mesh_file texture_file scale x y z
tmesh ./misc/skull.obj ./misc/Skull.jpg 1 0 0 20
```

FIGURE 11 – Exemple de fichier de configuration pour la version finale du programme