IN204 - Programmation Orientée Objet
Projet Final : *Ray Tracing*

David Velasquez Ospina
João Pedro Araújo Ferreira Campos

Janvier 2020

1 Introduction

Le *Ray Tracing* est une technique utilisée pour le rendu d'images en ordinateur. Elle consiste à simuler des rayons de lumière, qui partent de la caméra vers les objets de la scène, et ensuite des objets vers les sources de lumière. En calculant l'intersection des rayons avec les objets, on peut obtenir la couleur de chaque pixel de l'image que l'on veut créer. Le *Ray Tracing* est largement utilisé dans la génération d'images pour les jeux vidéo, ainsi que pour la recherche en physique optique.

Ce projet présente une implémentation en C++ de cette technique. L'idée générale de l'algorithme, les fonctions principales et la performance du programme sont présentése.

2 L'algorithme principal

Le programme principal lit une archive qui décrit la scène, et implémente l'algorithme dont le pseudo-code est présenté en 1. Un rayon de lumière est créé à partir de la position de la caméra, et suit la direction dans laquelle elle est pointée. Si le rayon ne touche aucun objet dans sa trajectoire, le pixel courant est coloré noir. Si au contraire il y a un objet dans son chemin, le point d'intersection est calculé et le pixel courant reçoit la couleur de l'objet trouvé. Cette couleur dépendra de couleur originale de l'objet, si sa superficie est réflexive ou pas, et de l'existence d'ombres causés par d'autres objets. Ce calcul est fait par la fonction *getColorAt()*. Elle prend en compte si l'objet est réflecteur, et fait un appel récursif le cas échéant, ayant comme argument le rayon reflété. Son comportement en haut niveau est décrit dans 2. Pour trouver la direction d'un rayon R reflété, on fait un calcul qui prend en compte le vecteur normal au point d'intersection, N. $\vec{R}_{reflet}$ est donné par :

$$\vec{R}_{reflet} = (\vec{S} - \vec{R}) \cdot 2$$

où

$$\vec{S} = \vec{N} \times (-\vec{R} \cdot \vec{N})$$

Le schéma de ce calcul est montré dans la figure 1 où $\vec{A2}$ est le rayon que l'on cherche, $\vec{R}_{reflet}$.

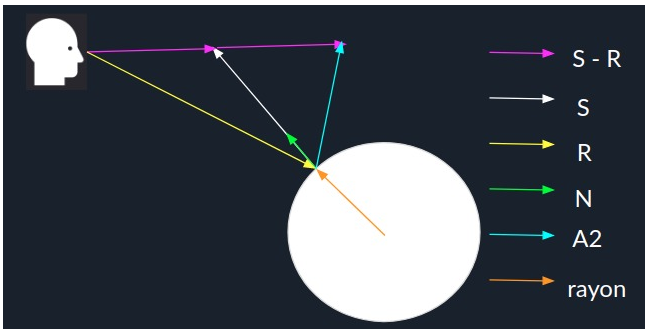


Figure 1: Calcul du rayon reflété

Pour arriver à faire ce calcul, il est important de calculer l'intersection des rayons avec les sphères. Une droite peut être représentée par des équations paramétriques :

$$x = x_1 + (x_2 - x_1)t$$
$$y = y_1 + (y_2 - y_1)t$$
$$z = z_1 + (z_2 - z_1)t$$

La sphère dont le centre est le point $c = (x_c, y_c, z_c)$ peut être représentée par :

$$(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 = r^2$$

En combinant les équations, on obtient l'équation quadratique de la forme $at^2 + bt + c = 0$ avec les coefficients :

$$a = (x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2$$
$$b = -2[(x_2 - x_1)(x_c - x_1) + (y_2 - y_1)(y_c - y_1) + (z_c - z_1)(z_2 - z_1)]$$
$$c = (x_c - x_1)^2 + (y_c - y_1)^2 + (z_c - z_1)^2 - r^2$$

Si $b^2 - 4ac > 0$, la droite intercepte la sphère et la valeur de t donnée par la solution donnera le point d'interception.

Les algorithmes principaux, décrits en haut niveau, sont donc :

```
Data : pixels[width][height], vector<Object*>, vector<Source*>
begin
    for chaque pixel de l'image do
        for i to nombre d'objets do
            intersections[i] ← findIntersection(objet, rayon);
        end
        indexClosestObject ← winningObjectIndex(intersections)
        if aucun objet trouvé then
            pixel ← BLACK
        end
        else
            pixel ← getColor()
        end
    end
end
```

Algorithme 1 : Ray tracing

```
Data : objet plus proche de la caméra, rayon, point d'intersection
begin
    if objet réflécteur then
        cherche un objet dans le chemin du rayon réfléchi
        getColor()
    end
    for i to nombre de lumières do
        crée rayon du point d'intersection vers la source lumineuse
        vérifie s'il rencontre d'autres objets pour créer des ombres
        met à jour finalColor
    end
    return finalColor
end
```

Algorithme 2 : Get Color

Il est donc prévu une complexité de $O(P \times N)$ pour l'algorithme principal, où P est la quantité de pixels et N le nombre d'objets. Cela a été vérifié en faisant augmenter l'un des deux paramètres, avec le deuxième constant. Le résultat est présenté dans les figures 2 et 3, dans lesquelles on voie bien que le temps d'exécution augmente de manière linéaire pour P et N.

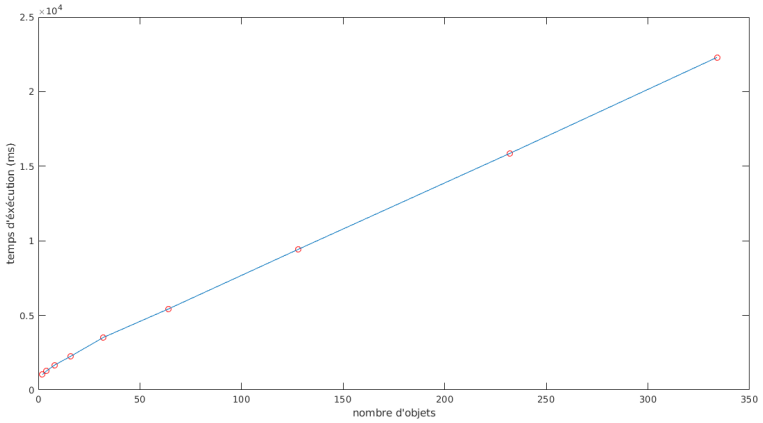


Figure 2: Temps d'exécution en fonction du nombre d'objets

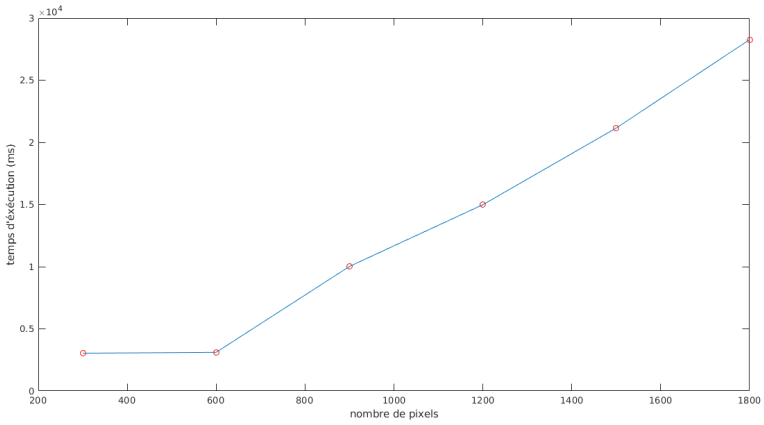


Figure 3: Temps d'exécution en fonction du nombre de pixels

3 Les Classes

Les classes qui composent le programme sont présentées ci-dessous. Leurs attributs et relations peuvent être visualisés sur la figure 7.

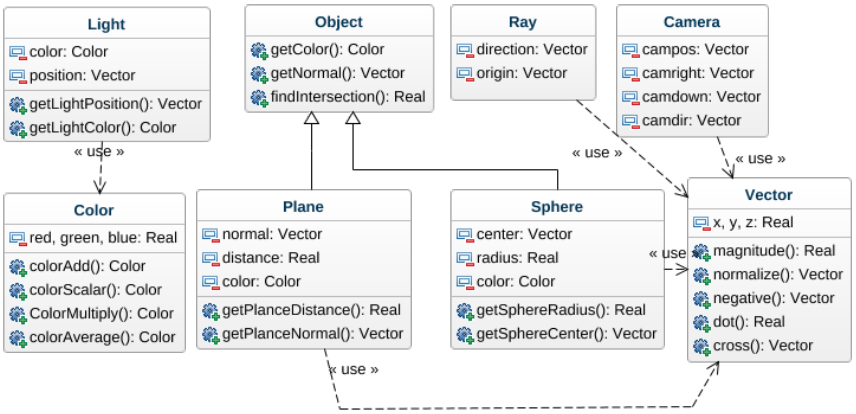


Figure 4: Diagramme de classes

- **Les Objets**
  La classe base pour les objets est la classe Object. Elle définit des fonctions virtuelles qui sont implementées dans les classes filles : virtual Color getColor (), virtual Vect getNormalAt(Vect intersection_position), virtual double findIntersection(Ray ray), virtual void ShowMe(). Deux classes héritent donc de *Object* : *Sphere*, défini par un centre et un rayon, et *Plane*, défini par un vecteur normal et la distance à la caméra.

- **Vect**
  Classe qui définit un vecteur dans un espace de trois dimensions, avec trois coordonnées réelles. Implémente sous forme de méthodes les opérations que l'on peut faire avec les vecteurs : produit scalaire, produit vectoriel, multiplication par scalaire et somme de vecteurs.

- **Ray**
  Cette classe représente un rayon de lumière. Le rayon est composé de deux vecteurs : un définissant son origine, et l'autre sa direction.

- **Camera**
  La caméra est définie par un vecteur position et un vecteur direction (dans laquelle elle est pointée). La direction est calculée dans le programme comme la différence entre sa position et le point vers lequel on souhaite que le regard soit dirigé.

- **Color**
  La classe Color utilise RGB pour créer les couleurs des pixels. Un quatrième attribut, appelé *Special*, a été ajouté à la classe pour que le programme soit capable de générer des reflets. Si cette valeur est comprise entre zéro et un, la surface de l'objet est réfléchissante. Si la valeur vaut 2, l'objet obtient un motif de carrés.

4 Parallélisation

Le programme a été parallélisé par MPI, en répartissant les pixels dans les différentes tâches. MPI a été choisi car le programme comporte des données différentes qui poseraient des problèmes d'accès à la mémoire en cas de parallélisation avec mémoire partagée. Cependant, comme un seul ordinateur est utilisé pour faire les calculs, le nombre de pixels qui peut être calculé sont limités. Pour le calcul du Speed Up, l'ordinateur utilisé a les spécifications suivantes:

Architecture:: x86_64
CPU op-mode(s): 32-bit, 64-bit
Byte Order: Little Endian
CPU(s): 12
On-line CPU(s) list: 0-11
Thread(s) per core: 2
Core(s) per socket: 6
Socket(s): 1
NUMA node(s): 1

La scène rendue pour les tests est représentée sur la figure 7. Elle est composée de deux sphères réfléchissantes et d'un plan avec un motif. La taille de l'image a été fixée à 600 de large par 400 de haut et le dpi à 72. L'antialiasing a été utilisé avec une profondeur de 2. Le résultats des tests se trouvent dans le tableu 1.

Nombre de processus	**Temps d'exécution(s)**	**Temps de calculs de la tâche plus lente(s)**	**Speed Up**
1	4.24122	4.22223	1
2	2.52337	2.5069	1.681
4	1.35328	1.33651	3.135
8	1.29287	1.207	3.514
12	0.943565	0.926658	4.420

Table 1: Speed Up du programme parallélisé par MPI

5 Les Bibliothèques

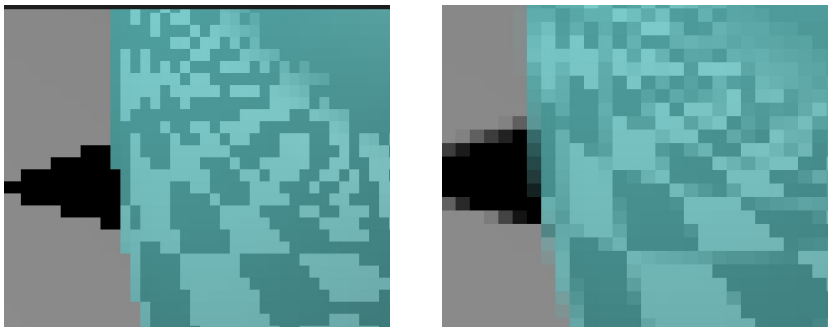
Le programme utilise deux bibliothèques préconçues : *json11* et *argh*. La première est utilisée pour faire la lecture d'une archive du type json, qui contient les objets de la scène à être rendue. La deuxième sert à processer les arguments passés lors de l'éxécution. Si aucun argument n'est fourni, le programme ouvre par défaut le fichier "scene.json". Ce fichier contient les sources de lumière et les objets qui feront partie de la scène, et fournit toutes les informations nécessaires pour que les classes soient instanciées. Il est donc possible de choisir les

positions et couleurs de chaque objet. Le format de ce fichier est montré dans la figure 5.

Figure 5: Fichier .json

6 Antialiasing

La fonction *main* contient la variable *aadepth*, responsable pour définir la profondeur du Antialiasing. Cette variable est un nombre entier, et plus sa valeur est importante plus les détails de l'image finale seront visibles. L'idée du antialiasing est de créer des pixels virtuels pour chaque pixel réel. Par exemple, si *aadepth* vaut 4, pour chaque pixel quatre nouveaux pixels seront crées. Ensuite, le pixel original reçoit la moyenne des quatre couleurs. L'effet obtenu peut être visualisé sur la figure 6.



(a) Sans anti aliasing (b) Avec anti aliasing de profondeur 20

Figure 6: Effet de l'antialiasing

7 Résultats

Ici deux exemples d'images sont présentés. Le premier est une scène avec deux sphères et deux sources de lumière, sans antialiasing. Le deuxième a été créé avec anti aliasing de profondeur 16, dans lequel on constate la meilleure qualité de l'image.

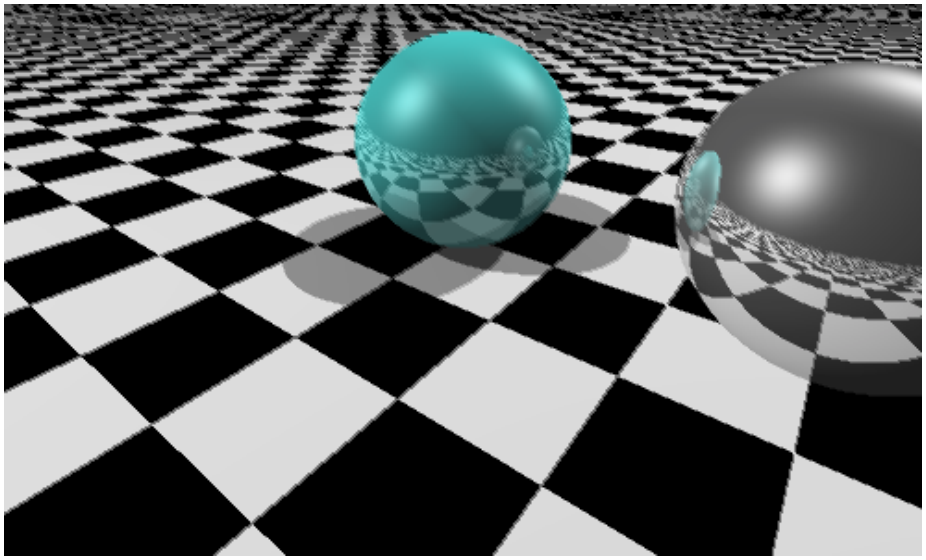


Figure 7: Scène rendue avec deux sources lumières

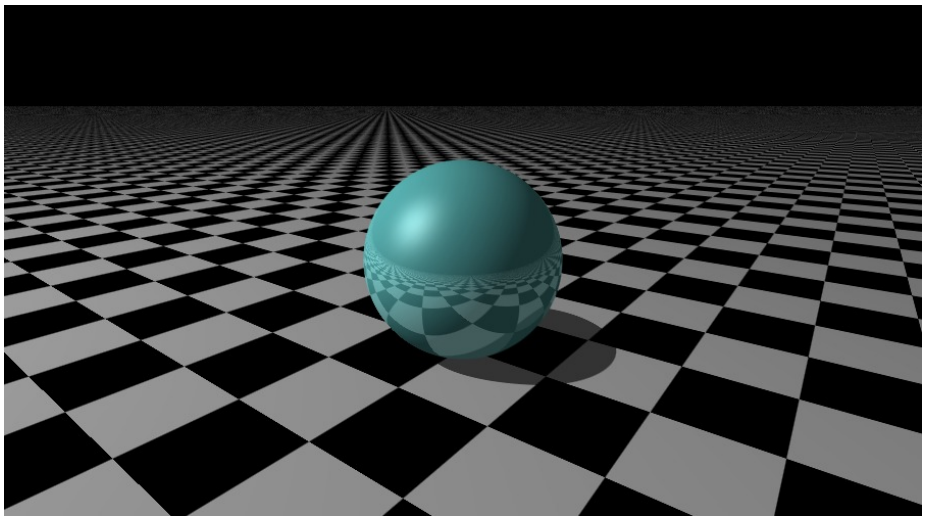


Figure 8: Scène rendue avec anti-aliasing

8 Conclusion et discussion

Le *Ray Tracing* est une technique dont la complexité d'implémentation est compensée par les résultats très réalistes que l'on peut obtenir. Ce projet a largement contribué à développer les habilités de programmation orientée objet, ainsi que l'usage du langage C++. Les principales difficultés rencontrées ont lien avec l'algèbre du problème, c'est-à-dire la définition des systèmes de coordonnées et le calcul de direction des rayons. Malgré des résultats satisfaisants au niveau de la qualité des images rendues, il y a beaucoup de fonctionnalités qui peuvent y être ajoutées.

Il est possible de constater que ce projet laisse de la marge pour plusieurs améliorations, qui pourraient être implémentées lors d'une nouvelle version du programme. Notamment les objets supportés pourraient être plus variés, de façon à faire le rendu d'images plus complexes. Pour que l'utilisation du programme soit plus simple pour un utilisateur, il peut être intéressant d'implémenter une interface dans laquelle on pourrait définir toutes les caractéristiques de la scène que l'on souhaite créer. Cela permettrait de définir les scènes de manière plus simple que d'avec un fichier json à modifier. Un programme plus sophistiqué permettrait aussi de déplacer la caméra en temps réel, c'est-à-dire changer le point de vue de la scène sans avoir à le réinitialiser avec d'autres paramètres.

References

[1] *Sphere and line intersection* disponible sur *http://www.ambrsoft.com/TrigoCalc/Sphere/SpherLineIntersectio*

[2] *Introduction to Ray Tracing: a Simple Method for Creating 3D Images* disponible sur *https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-ray-tracing*

[3] Juvigny, X. : notes de cours, Programmation Parallèle. (2019)

[4] Monsuez, B. : notes de cours, Programmation Orientée Objet. (2019)