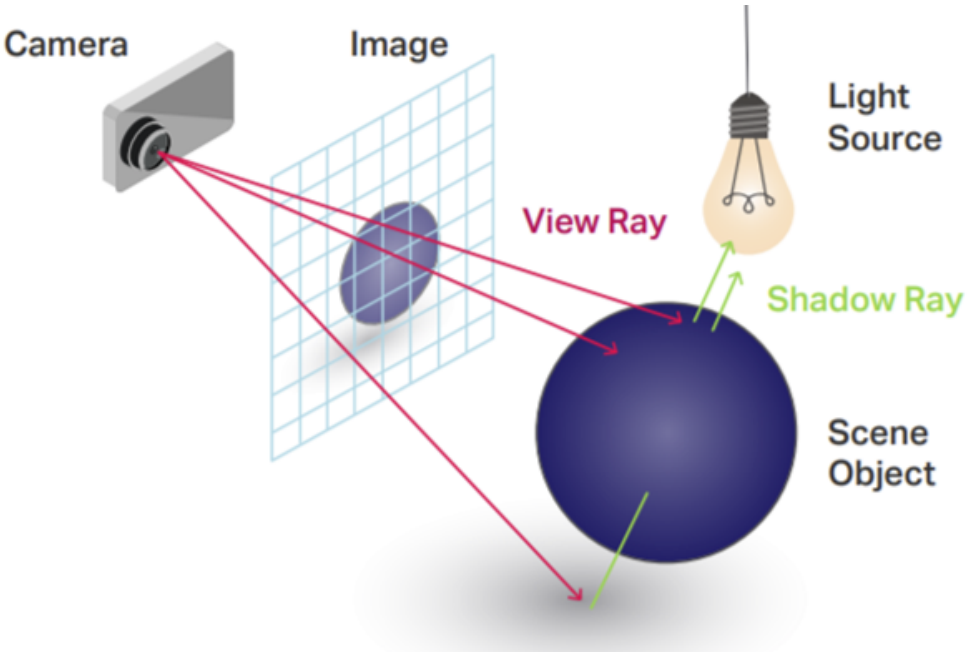
Projet de Programmation Avancée

Lancer de rayons

Teddy ALEXANDRE, Arthur BABIN



Décembre 2022

Table des matières

1 Introduction

Le sujet de notre projet de programmation avancée est le **lancer de rayons**. Il s'agit d'une technique de rendu de synthèse d'images 3D, c'est-à-dire d'obtention d'une scène d'objets en trois dimensions sur un écran en deux dimensions. Cette technique est très utilisée dans de nombreux domaines, notamment celui du jeu vidéo pour obtenir de bons rendus graphiques.

Plusieurs techniques d'approches peuvent être employées, comme la **rastérisation**, qui consiste en l'application de formules de projection de l'image 3D en 2D. Le procédé est en général rapide mais parfois peu réaliste, car les ombres doivent être simulées.

On utilise donc souvent une deuxième technique, plus réaliste, qui est le **tracé de rayons** (raytracing), et qui repose sur le principe du retour inverse de la lumière. En effet, au lieu de lancer des rayons à partir des diverses sources lumineuses pour voir où ils rebondissent, on part du point d'arrivée (la caméra dans le cadre du projet) et pour chaque pixel de l'écran, on observe la trajectoire du rayon lumineux dirigé vers ce pixel. La méthode est plus coûteuse en termes de temps d'exécution néanmoins, d'autant plus si on tient compte des diverses propriétés des matériaux des objets présents dans la scène (réflexivité, réfraction, transmission, etc.).

C'est donc cette technique que nous avons dû mettre en place dans le cadre du projet. Ce dernier a été écrit en C++, commenté avec Doxygen, et utilise la bibliothèque externe SDL pour effectuer les affichages à l'écran.

2 Choix effectués

2.1 Modélisation du problème

Nous avons fixé quelques conventions pour résoudre ce problème. On se place dorénavant dans l'espace (cf Figure 6).

La boîte englobante correspondant aux "murs" dans notre scène est de dimensions $(2000, 1000, 1000)$ et son centre est en $(0, 500, 0)$. Nous avons de plus placé la caméra en $(100, 600, -400)$.

En ce qui concerne la grille d'affichage de la scène, celle-ci se situe dans la direction $(0, 0, 200)$ par rapport à la caméra. De plus, l'origine du repère se situe en haut à gauche de la grille, avec l'axe des abscisses orienté vers la droite, et l'axe des ordonnées orienté vers le bas, conformément à la SDL qui fonctionne ainsi.

Il y a également une source de lumière omnidirectionnelle presque au centre de la scène en $(100, 500, 0)$. Nous avons fait le choix de ne pas prendre en compte la direction de cette lumière pour un meilleur rendu mais on aurait également pu faire en sorte de pouvoir gérer une liste de lumières directionnelles et il aurait ainsi fallu prendre en compte pour chaque lumière le produit scalaire entre sa direction avec le vecteur unitaire partant du point d'intersection vers son origine.

La scène dispose également d'une sphère et d'autres parallélépipèdes rectangles (alignés ou non par rapport aux axes) que l'on place librement dans la boîte, afin de vérifier le bon fonctionnement de l'algorithme de tracé de rayons.

2.2 Diagramme UML / Ajout de méthodes

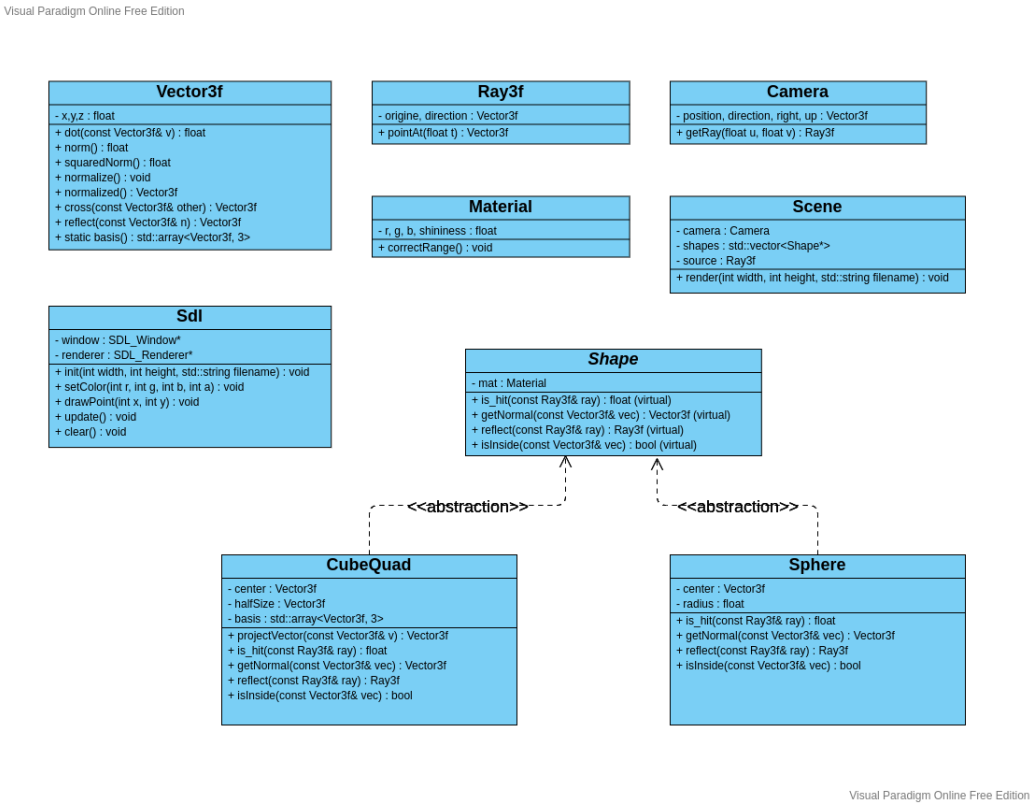


Figure 1: Diagramme UML du projet

Nous avons fait le choix d'ajouter et de modifier quelques méthodes afin de faciliter certaines portions du code final.

Dans la classe *Vector3f*, des méthodes sur les calculs avec les vecteurs ont été ajoutées (produit scalaire, produit vectoriel, calcul de norme, normalisation d'un vecteur, vecteur réfléchi par rapport à un vecteur incident), elles servent essentiellement pour déterminer les intersections entre les objets et les rayons lumineux.

Dans la classe *Ray3f*, justement, nous avons ajouté une méthode pointAt qui détermine le point paramétré par un flottant t sur la droite engendrée par le rayon lumineux (l'équation d'un rayon lumineux d'origine O et de direction $\vec{d}$ étant $\overrightarrow{r(t)} = O + t\vec{d}$.

Dans la classe abstraite *Shape* (et donc ses classes filles *CubeQuad* et *Sphere*), on a surchargé la méthode *is_hit* en renvoyant la distance entre l'origine du rayon et la Shape si il y a intersection, -1 sinon. Nous avons également défini les méthodes *getNormal* et *isInside* qui déterminent respectivement le vecteur normal à la Shape en un point, et si un point se situe à l'intérieur de la Shape ou non. Ces deux méthodes virtuelles pures sont implémentées dans les classes filles *CubeQuad* et *Sphere*.

Dans *CubeQuad*, nous avons modélisé les figures un peu différemment de l'énoncé : un cube est défini par son centre, sa **demi-taille** (sous forme de vecteur 3D), et la **base** dans laquelle le *CubeQuad* est aligné par rapport aux axes. Cela nous permet ainsi d'afficher par exemple des cubes orientés. Nous avons aussi implémenté la base canonique associée aux *CubeQuad* non orientés.

On a enfin écrit une classe *SDL* qui implémente des méthodes utiles de la bibliothèque SDL, afin d'en faciliter l'utilisation (masquage de l'utilisation des pointeurs sur la fenêtre et le rendu).

3 Calcul de la couleur d'un pixel

Nous avons décidé d'utiliser un modèle de réflexion (additionné au modèle de Phong)(cf Figure 2) :

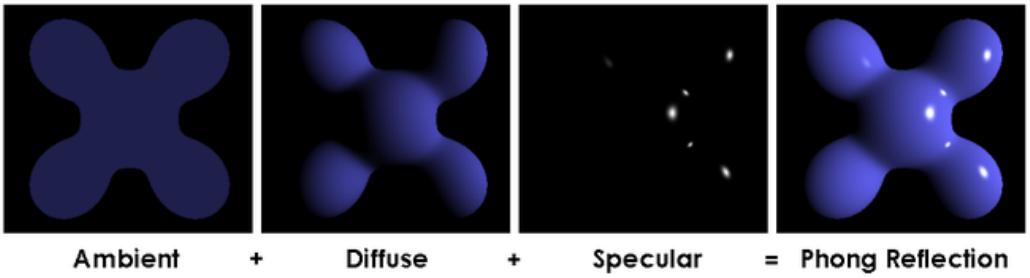


Figure 2: Illustration du modèle de Phong : l'intensité totale est la somme des trois composantes décrites ci-dessus

$$c_{\text{total}}^{\text{ray}} = \begin{cases} 0 & \text{s'il n'y a pas d'intersection} \\ c_{\text{a}} & \text{si le point d'intersection n'est pas éclairé} \\ c_{\text{a}} + c_{\text{d}} + c_{\text{s}} + c_{\text{reflected}} & \text{sinon} \end{cases}$$

$$\text{avec} \begin{cases} s & \text{la shininess du matériau} \\ \vec{n} & \text{la normale au point d'intersection} \\ \vec{r} & \text{le vecteur unitaire intersection-lumière} \\ c_{\text{material}} & \text{la couleur du matériau du point d'intersection} \\ c_{\text{a}} = 0.2 * c_{\text{material}} & \text{sinon} \\ c_{\text{d}} = (\vec{n} \cdot \vec{r}) c_{\text{material}} & \text{sinon} \\ c_{\text{s}} = 0.1(\vec{n} \cdot \vec{r})^s c_{\text{material}} & \text{sinon} \\ c_{\text{reflected}} = s c_{\text{total}}^{\text{reflected ray}} & \text{la couleur réfléchie} \end{cases}$$

4 Algorithmes et problèmes rencontrés

Nous avons donc suivi l'algorithme fourni dans l'énoncé : la détermination de la grille n'a pas posé de problème particulier.

Concernant le calcul de l'image, plusieurs problèmes se sont posés :

- Comment gérer les intersections pour calculer le rayon réfléchi ? Comment alors gérer le cas d'une réflexion (objet avec coefficient de luminosité strictement positif) ?

En effet, afin de déterminer entièrement le rayon réfléchi, nous avons besoin de connaître le point d'intersection entre le rayon incident et notre objet (cube ou sphère). En fonction du type de primitive, on peut alors déterminer le vecteur normal et en déduire le rayon incident. C'est donc ainsi que nous avons implémenté pour chaque primitive une méthode reflect qui calcule ce rayon réfléchi r, dont la formule est donnée grâce aux lois de Descartes sur la réflexion et la réfraction :

$\vec{r} = \vec{i} - 2(\vec{i} \cdot \vec{n})\vec{n}$, avec $\vec{i}$ le rayon incident et $\vec{n}$ le vecteur normal.

On peut donc implémenter les réflexions pour calculer les couleurs, en écrivant un programme récursif pour le faire de la manière la plus naturelle possible.

- Comment savoir si un pixel est caché ou pas ? Comment calculer alors la couleur finale d'un pixel ?

Afin de déterminer comment colorier le pixel, on a besoin de savoir si le point associé dans la scène est exposé à la source de lumière. Nous avons donc parcouru l'ensemble des primitives de la scène, et vérifié si chacune des primitives était frappé par le rayon qui part du point d'intersection entre le rayon incident et l'objet, et qui se dirige vers la source de lumière. Si c'est le cas, le pixel est caché et l'on affiche avec un gris sombre (afin de distinguer du noir complet, de la couleur de fond). Sinon on utilise les propriétés de notre Shape, et notamment de son Material pour renvoyer une couleur au format RGB. On peut bien évidemment complexifier l'ensemble et introduire un coefficient de réflexion pour tenir compte des réflexions sur l'objet, qui contribuent aussi à la couleur finale du pixel.

En ce qui concerne les pavés droits orientés c'est-à-dire non alignés par rapport aux axes nous avons ajouté un attribut *basis* qui correspond à la base de vecteurs dans laquelle le *CubeQuad* est aligné. Cet attribut nous permet ainsi d'utiliser les algorithmes d'intersection d'une droite avec des plans lorsqu'ils sont alignés par rapport aux axes après une projection du rayon dans la base correspondante. Cependant nous avons rencontré un problème que nous n'avons pas su résoudre: la position des *CubeQuad* orientés subit une translation non désirée et non constante (elle dépend de l'angle de rotation) mais cela n'a pas de répercussion sur la qualité ou le réalisme du rendu.

5 Réalisation et rendu final

Le rendu final est fonctionnel, avec un programme qui compile et un affichage sur la fenêtre créée avec la classe SDL. Nous avons réussi, par le biais d'une fonction récursive, à rajouter les effets de la réflexion en fonction de la luminosité. Bien que le temps d'exécution soit rallongé à cause des calculs récursifs (de l'ordre d'une dizaine de secondes), les rendus restent très appréciables de ce point de vue là.

Néanmoins, tout n'est pas parfait : la présence de petits points au niveau des figures donne un rendu parfois un peu brouillon, ce qui est aussi accentué par la présence d'aliasing à l'écran (les images sont crénelées). Il était donc possible d'aller plus loin dans le projet à ce niveau-là.

En effet, nous n'avons pas pu consacrer suffisamment de temps au projet, qui aurait pu être encore plus travaillé et offrir donc un meilleur rendu. L'ensemble était néanmoins difficile à conjuguer avec les cours en parallèle et les autres projets à rendre.

Nous sommes néanmoins satisfait de pouvoir obtenir un rendu d'image 3D avec l'ensemble des classes implémentées, car cela a tout de même requis un certain temps avant de pouvoir afficher des rendus intéressants à la fenêtre.

6 Conclusion sur le projet

Le projet de tracé de rayons s'est révélé très intéressant, et nous a permis d'en apprendre davantage sur les méthodes de rendu d'images 3D grâce aux théorèmes d'optique. Il nous a permis également de renouer avec quelques théorèmes et fondements de l'optique, qui ont été abordés antérieurement dans notre formation d'ingénieur (en prépa scientifique plus précisément), ainsi que quelques calculs géométriques.

Ce projet nous a également permis de nous convaincre de l'intérêt certain que possède la programmation orientée objet, qui s'est révélée très utile ici. Nous ne sommes pas certains que ce projet aurait été plus simple sans l'apport de ce paradigme de programmation.

Figure 3: Premier rendu obtenu, sans les réflexions et sans dégradé de couleur

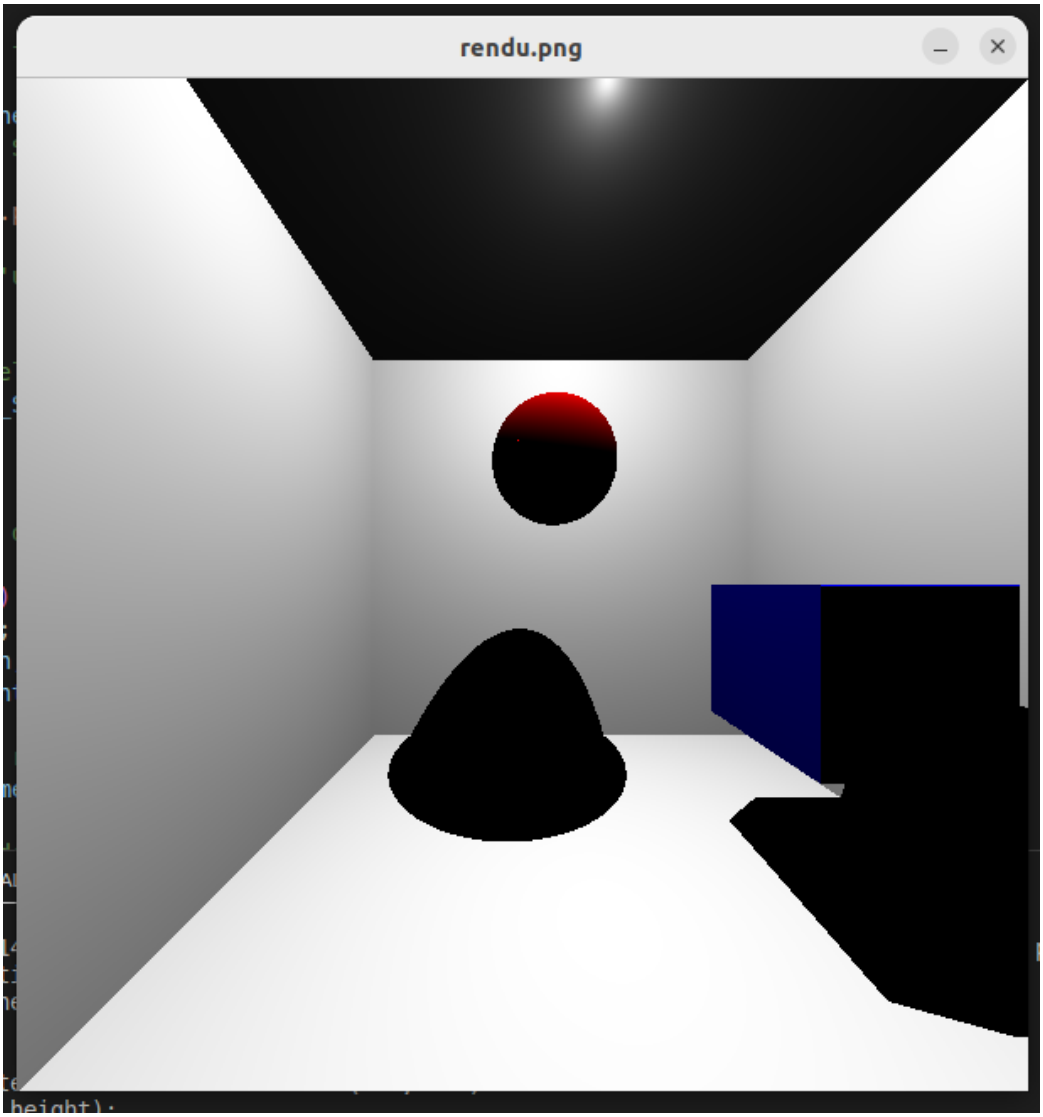


Figure 4: Second rendu obtenu, avec un dégradé de couleur suivant l'angle d'inclinaison avec la source de lumière

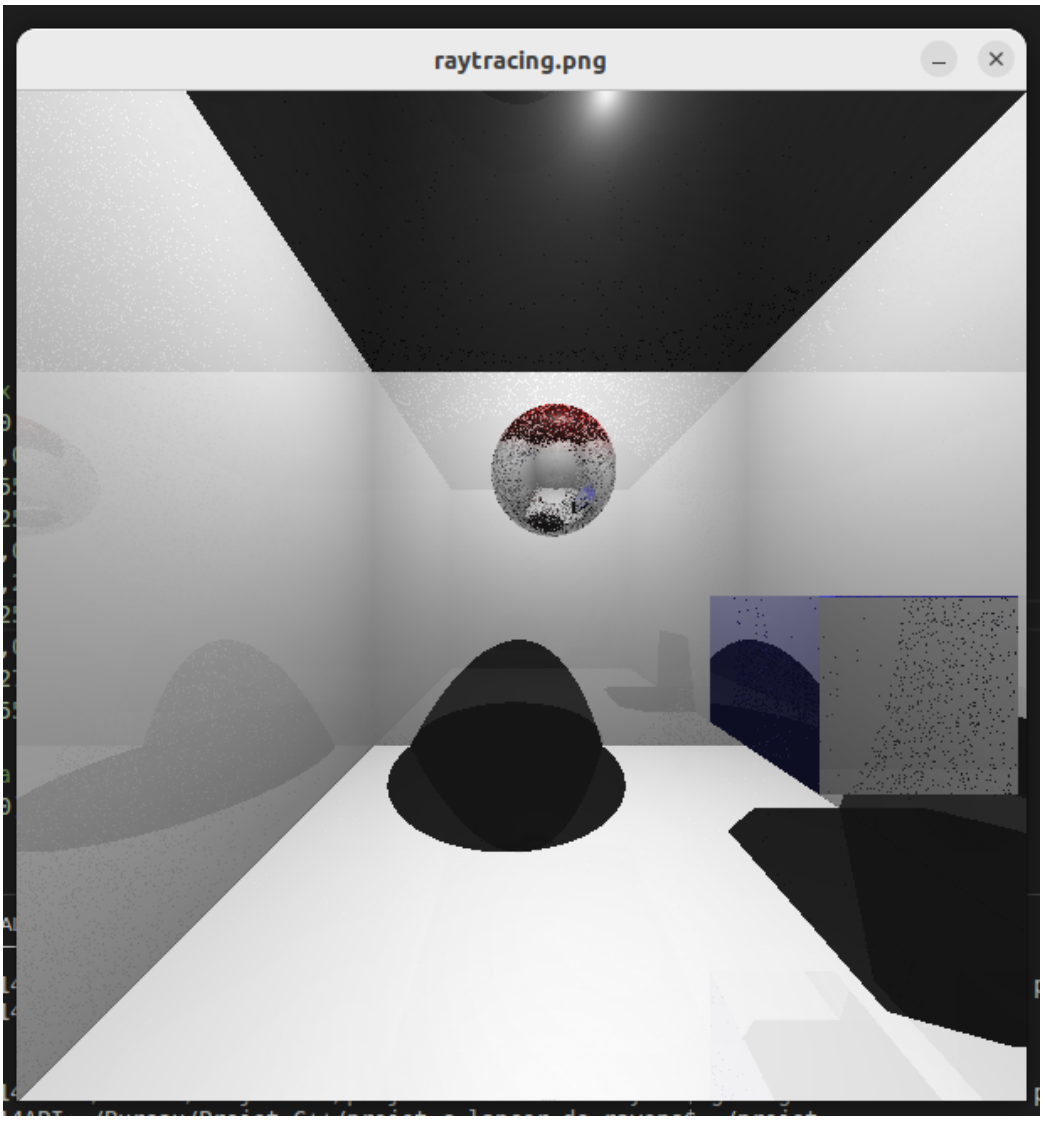


Figure 5: Rendu avec des effets de réflexion sur les objets

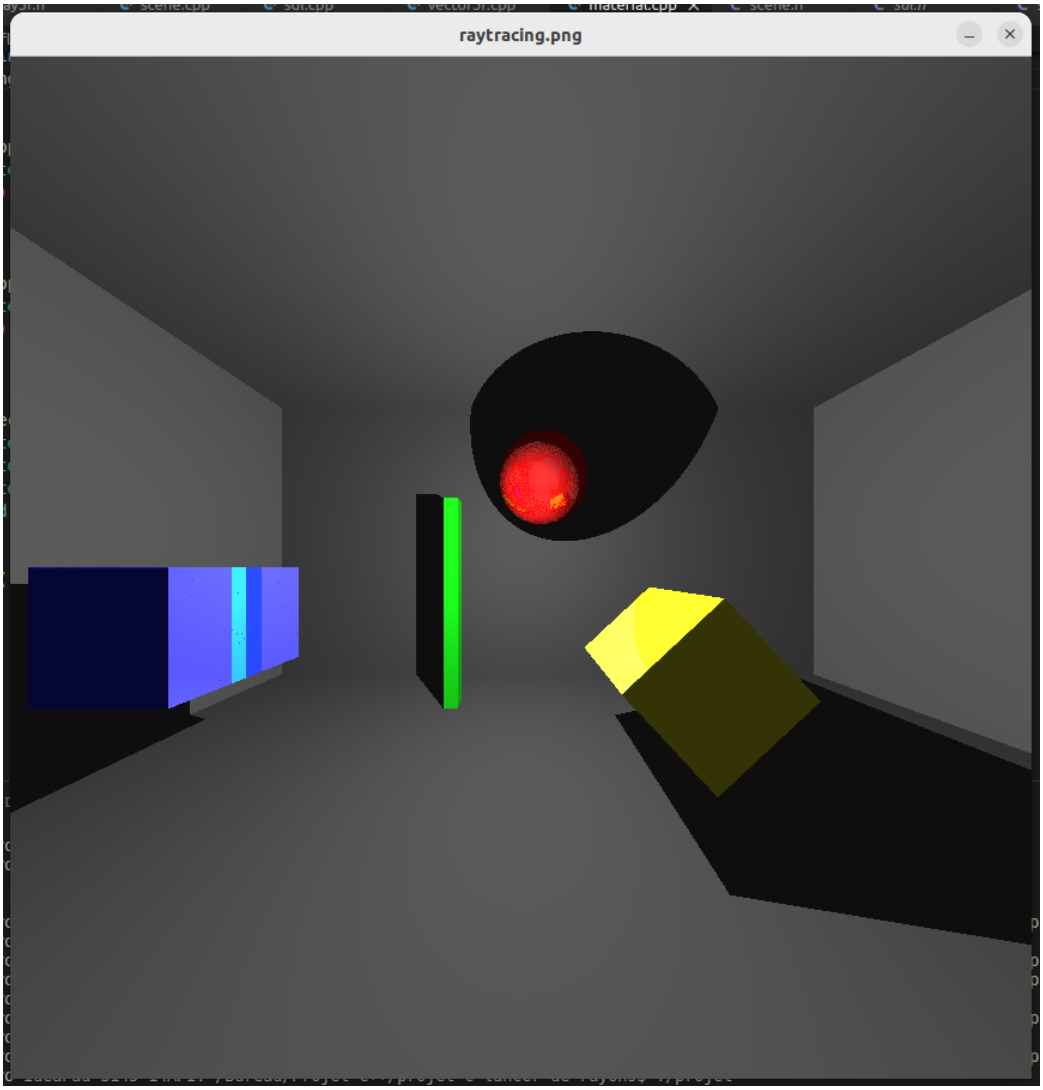


Figure 6: Rendu obtenu en appliquant le modèle de Phong

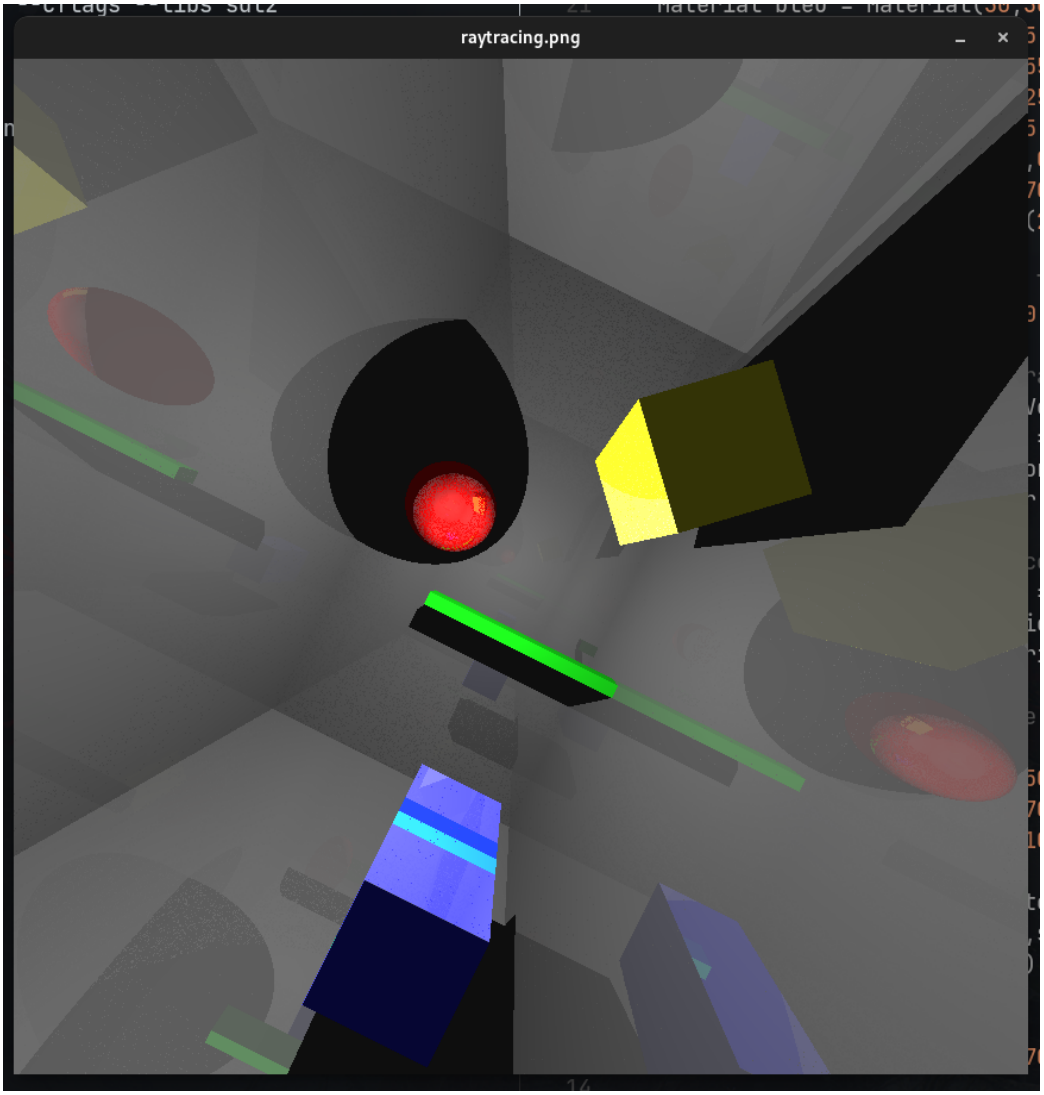


Figure 7: Rendu obtenu en appliquant le modèle de Phong (avec rotation de la caméra et des murs légèrement réfléchissants)