Rapport du projet de programmation avancée et projet
Sujet 4 : lancer de rayon

Lisa ACHARD, Audrey SANCHEZ

Table des matières

1 Introduction

L'objectif est de représenter une scène 3D sur un écran avec **le tracé de rayon**. Ce procédé est intéressant puisqu'il permet d'avoir un résultat plus réaliste qu'une simple rastérisation. Son principe est le suivant : il s'agit de représenter une scène 3D sur un écran avec le lancer de rayons. Il s'agit en effet de **lancer des rayons qui partent de l'oeil ou de la caméra, vers les pixels de l'écran**. Si un rayon se coupe avec un objet de la scène, alors la couleur de l'objet est appliquée au pixel. À l'inverse, si le rayon ne se coupe avec aucun objet, alors le pixel sera noir. Ce procédé peut être amélioré en prenant en compte les différents éclairages de la pièce, et ainsi, le résultat sera d'autant plus réaliste que les sources de lumière primaires et secondaires seront prises en compte.

Nous avons ainsi appliqué le lancer de rayons afin de réaliser une scène 3D contenant une sphère, un cube et une source de lumière accrochée au plafond de la scène.

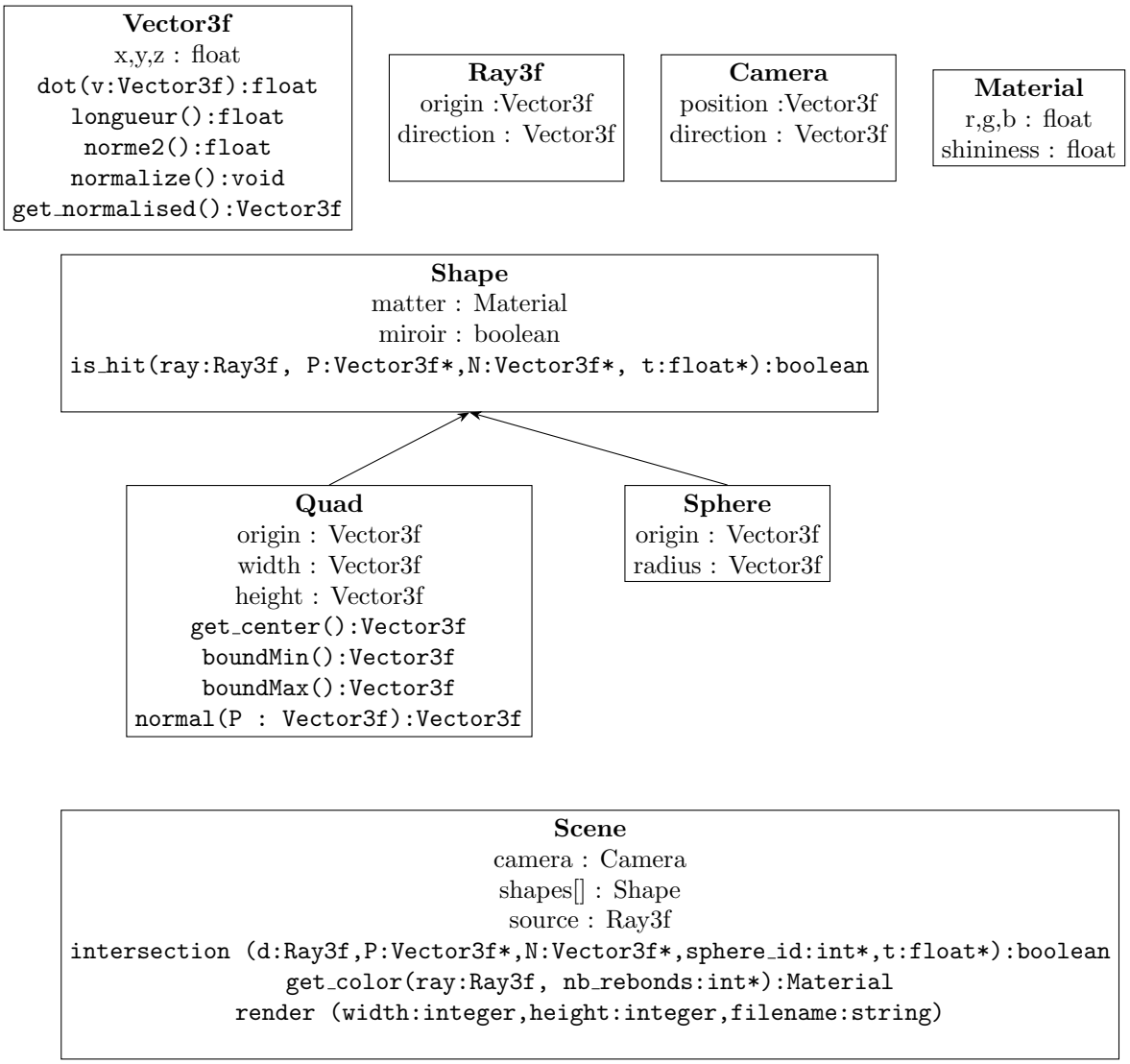
Pour cela, nous avons codé en C++ et utilisé la librairie SDL. Le code devra être compilé avec :

```
g++ -g -Wall -Wextra -o projet *.cpp `pkg-config --cflags --libs sdl2`
```

.

2 Diagramme UML

Le diagramme UML du projet est le suivant. Par un soucis de visibilité, les constructeurs, getters, setters et operators, n'ont pas été transcrits dans ce diagramme. Par ailleurs, comme le code a été commenté en utilisant la syntaxe de Doxygen, des informations sur les fonctions sont disponibles dans le code.



Les **méthodes principales** de notre projet sont :

- **Dans Shape :**
La méthode virtual **is_hit** permet de vérifier si le rayon intersecte l'objet et de récupérer le point P d'intersection du rayon, la normale $\vec{N}$ à ce point et la valeur de t à ce point (cf plus loin pour la définition de t).

- **Dans Sphere :** La méthode **is_hit** de la classe mère Shape est implémentée pour une sphère.

- **Dans Quad :** La méthode **is_hit** de la classe mère Shape est implémentée pour un cube grâce aux méthodes get_center(), boundMin(), boundMax() et normal(P:Vector3f).

- **Dans Scene :** La méthode **intersection** permet de connaitre l'objet intersecté le plus proche pour un rayon donné. La méthode **get_color**, quant à elle, utilise la fonction intersection, et détermine la couleur du pixel défini pour un rayon donné. Enfin, la méthode **render**, permet notamment d'afficher et d'enregistrer l'image de la scène.

3 Objets

Dans la suite, nous allons utiliser les notations suivantes. La caméra est placée à un point C et la source de lumière à un point S de l'espace. Soit alors un rayon, d'origine C et de direction $\vec{u}$, choisi unitaire. Soit P un point sur la demi droite créée par ce rayon. P peut alors être défini tel que :

$$\exists\, t \in \mathbb{R}^+, P = C + tu$$

3.1 Sphères

3.1.1 Intersection rayon-sphère

Posons O le centre d'une sphère, R le rayon de cette sphère et $\vec{OP} = P - O$. Nous cherchons à trouver le point d'intersection entre le rayon et la sphère.

Le système à résoudre est donc le suivant :

$$\begin{cases} P = C + tu & \text{(équation du rayon)} \\ \|\vec{OP}\|^2 = R^2 & \text{(équation du cercle)} \end{cases} \tag{1}$$

On remplace P dans la deuxième équation :

$$||C + t\vec{u} - O||^2 = R^2$$

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

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

car u est choisi unitaire

Nous obtenons donc une équation du second degré à résoudre.

Ainsi, si le discriminant est négatif, il n'y pas d'intersection. De plus, si la solution t est négative, alors il n'y a pas d'intersection également.

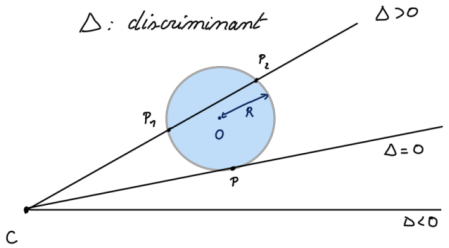


Figure 1: Intersection rayon-sphère en fonction du signe du discriminant

La résolution de ce polynôme de second degré est réalisée dans la fonction Sphere::is_hit, qui permet également d'obtenir le vecteur normal $\vec{N}$ et le vecteur dirigé vers la source de lumière $\vec{L}$, qui seront utiles pour l'éclairement.

3.1.2 Utilisation de plusieurs sphères

Dans notre code, nous avons au total **sept sphères**. Pour cause, nous avons utilisé tout d'abord cinq sphères pour réaliser les trois murs (gauche, droite, et fond), le plafond et le sol. En effet, en prenant des rayons très grands, ces sphères ressemblent à des plans et le résultat est satisfaisant. Par ailleurs, nous avons deux autres sphères, une de couleur rouge, et une qui est un miroir.

3.2 Cube

3.2.1 Intersection rayon-cube

On considère un cube que l'on définit par trois vecteurs de dimension 3 : O l'origine du cube, h dont la coordonnée y correspond à la hauteur du cube et w dont la coordonnée x correspond à la largeur et la profondeur du cube.

On définit alors les bornes min et max du cube comme les points qui sont au plus en haut et plus proche de la caméra, dans notre cas, l'origine O ; et au plus bas et plus loin, le point B_{max} sur le schéma.

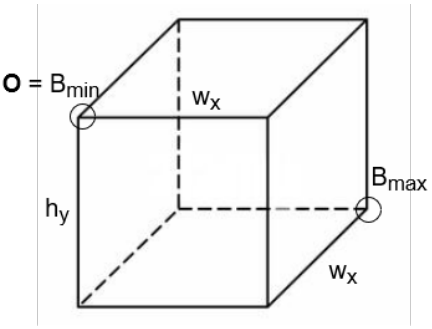


Figure 2: Représentation d'un cube dans le projet

Les limites B_{min} et B_{max}, qu'on appelera plus tard $B0$ et $B1$ définissent un ensemble de lignes parallèles à chaque axe du système de coordonnées. On peut donc exprimer n'importe quel point du cube en utilisant des équations de droite sur chaque axe. Exprimons par exemple ce point sur l'axe x (nous reprenons les notations introduites au début de la 2ème partie) :

$$C_x + tu_x = B0_x$$

On trouve l'équation suivante :

$$t0_x = \frac{(B0_x - C_x)}{u_x}$$

On fait ensuite la même chose pour chaque axe, et avec la borne $B1$. On obtient alors 6 valeurs de t :

$$t0_x = \frac{(B0_x - C_x)}{u_x}$$

$$t0_y = \frac{(B0_y - C_y)}{u_y}$$

$$t0_z = \frac{(B0_z - C_z)}{u_z}$$

$$t1_x = \frac{(B1_x - C_x)}{u_x}$$

$$t1_y = \frac{(B1_y - C_y)}{u_y}$$

$$t1_z = \frac{(B1_z - C_z)}{u_z}$$

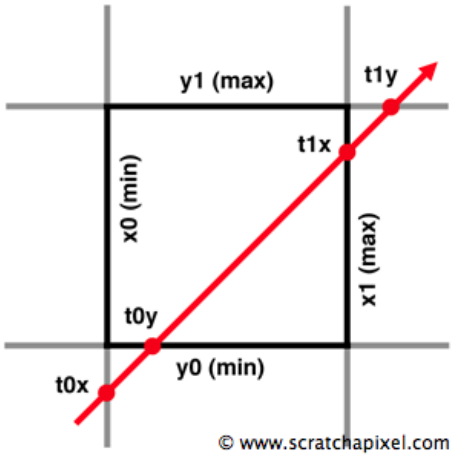


Figure 3: Intersection avec le cube en 2D

Pour savoir s'il y a intersection, il suffit alors de récupérer les points le plus loin t_{max} et le plus proche t_{min} de la caméra et de vérifier que $t_{max} > t_{min}$, sinon il n'y a pas d'intersection, et que $t_{max} > 0$, sinon le point est derrière la caméra.

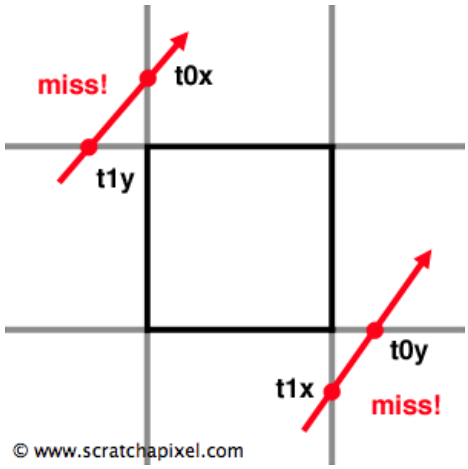


Figure 4: Pas d'intersection avec le cube en 2D

3.2.2 Normale au point

Après avoir déterminé si le point intercepte le cube ou non, il s'agit de trouver la normale en ce point, utile au calcul de l'éclairement. À la différence de la sphère, presque chaque point a une normale unitaire (en dehors des coins et des arêtes). On peut alors penser que le calcul de la normale est plus simple que pour la sphère, mais c'est en fait plus compliqué, puisqu'il s'agit de trouver la normale en fonction de son emplacement, ce qui est coûteux, à l'inverse de la calculer, comme pour la sphère.

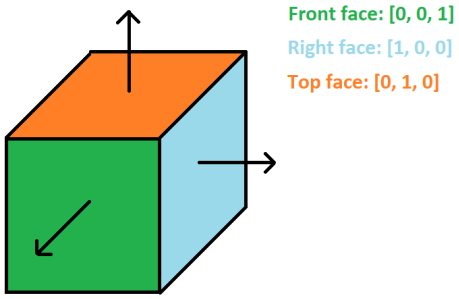


Figure 5: Exemple de Cube avec des normales

3.3 Relation entre objets

La fonction **Scene::intersection** permet, pour un rayon donné, de connaitre l'objet touché le plus proche. Pour cela, elle utilise la fonction **Shape::is_hit**,

implémentée dans les deux classes filles Sphere et Quad, qui renseigne l'existence d'une intersection entre un objet et le rayon, et le cas échéant le premier point d'intersection entre l'objet et le rayon. Ainsi, en **parcourant l'ensemble des points d'intersection**, la fonction choisit celui **le plus proche**, et renseigne alors les informations nécéssaires sur cette intersection.

4 Éclairage

4.1 Modèle d'éclairage lambertien

4.1.1 Éclairage direct

L'éclairage direct prend en compte les sources de lumières **primaires**, qui produisent leur propre lumière, comme l'ampoule que nous avons fixé au plafond de la scène.

Soit c une des 3 composantes de couleur d'un pixel. Soient ρ l'albedo pour cette composante, comprise entre 0 et 1, I l'intensité de la source de lumière, $\vec{N}$ la normale au point d'intersection P avec l'objet, L le vecteur partant du point d'intersection vers la source de lumière, et d la distance entre P et la source de lumière S. Le modèle d'éclairage lambertien nous indique alors que la valeur de la composante c vaut $\frac{\rho}{\pi}\frac{I<\vec{N},\vec{L}>}{d^2}$.

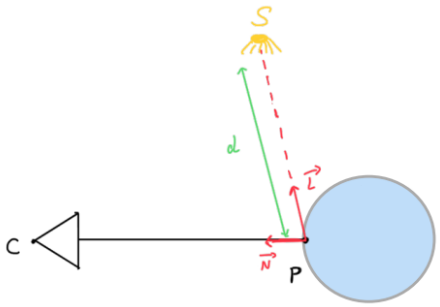


Figure 6: Modèle lambertien

Les **ombres** sont réalisées en prenant un rayon qui part de P et de direction $\vec{L}$. Si ce rayon intersecte un objet entre le point P et la source de lumière S, alors il est à l'ombre de cet objet. Le pixel prend alors la couleur noir.

L'éclairage direct est implémenté dans la fonction **Scene::get_color**.

4.1.2 Éclairage indirect

- Principe

L'éclairage indirect, quant à lui, prend en compte les sources de lumière **secondaires**, qui renvoient une partie de la lumière qu'elles reçoivent. Afin d'avoir un résultat toujours plus réaliste, il est ainsi utile de s'y intéresser.

La fonction de réflectance est une fonction qui décrit comment la lumière se réfléchit sur les objets. Elle est, à un facteur près, la probabilité qu'un photon se propage dans une direction donnée :

$$f : S^2 \times S^2 \times \mathbb{R} \to \mathbb{R}$$

Soient L_0 l'éclairage qui sort d'un objet et L_e l'éclairage émis par l'objet. Soient ω_i une direction incidente et ω_0 une direction reflétée. L'équation du rendu, qui décrit le comportement de la lumière, est alors la suivante :

$$L_0(x, \vec{\omega_0}, \lambda) = L_e(x, \vec{\omega_0}, \lambda) + \int L_i(x, \vec{\omega_i}, \lambda) f(x, \vec{\omega_i}, \vec{\omega_0}, \lambda) < \vec{N}, \vec{\omega_i} > d\vec{\omega_i}$$

Résoudre cette équation est compliqué, puisqu'en théorie, le nombre de rebonds de la lumière est infini, et ces rebonds se fond dans de multiples directions.

Cette équation doit respecter plusieurs principes :

- **1er principe** : conservation de l'énergie

$$\int L_i(x, \vec{\omega_i}, \lambda) f(x, \vec{\omega_i}, \vec{\omega_0}, \lambda) < \vec{N}, \vec{\omega_i} > d\vec{\omega_i} \leq 1$$

- **2nd principe**

$$f(\omega_i, \omega_0, \lambda) = f(\omega_0, \omega_i, \lambda)$$

- **3ème principe**

$$f \geq 0$$

Cette équation peut être résolue avec la **méthode de Monte-Carlo**, avec un échantillonage aléatoire, et en prenant un nombre de rebonds fini.

Nous avons admis la solution, qui est de générer 2 nombres aléatoires r1 et r2 tels que les coordonnées du rayon tranmis sont :

$$x = cos(2\pi r_1)\sqrt{1 - r_2}$$

$$y = sin(2\pi r_1)\sqrt{1 - r_2}$$

$$z = \sqrt{r2}$$

Figure 7: Éclairage indirect

Sur la figure ci-dessus sont illustrées deux méthodes possibles pour la réflexions des rayons. La méthode que nous avons utilisé est la **méthode de gauche,**

linéaire. En effet, nous lançons *nrays* rayons de la caméra et nous choisissons un nombre n de rebonds. Alors, pour chaque rayon lancé, nous réalisons n rebonds, en choisissant à chaque intersection un unique rayon. La **méthode de droite**, quant à elle, est **exponentielle**, et n'est donc pas préférable. Pour cause, elle lance un rayon de la caméra, et à chaque rebond, elle lance plusieurs rayons réfléchis, qui eux-mêmes lanceront plusieurs rayons réfléchis au prochain rebond.

- **Implémentation et problèmes rencontrés**

L'éclairage indirect aurait ainsi permis d'avoir un résultat plus réaliste, en prenant en compte les rayons réfléchis. Nous avons implémenté sa contribution dans **Scene::get_color**.

Le principe est le suivant : dans **Scene::render**, pour chaque pixel, on appelle plusieurs fois la fonction get_color sur le rayon caméra-pixel avec un certain nombre de rebonds. À chaque rebond, la fonction **get_color** va **choisir un rayon réfléchi** "aléatoirement" et va **appeler récursivement la fonction get_color** avec un nombre de rebonds en moins.

Pour chaque pixel est réalisée la moyenne des différents résultats de get_color. Par exemple, si on appelle 15 fois get_color pour chaque pixel, cela signifie que, pour un pixel donné, on va s'intéresser à 15 rayons réfléchis issus de l'éclairage indirect.

Cependant, le problème suivant s'est posé : avec un nombre de rayons réfléchis trop faible, l'**image** obtenue est **bruitée** et le résultat n'est alors pas satisfaisant.

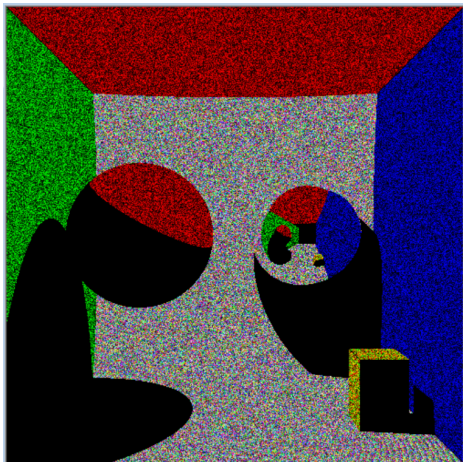


Figure 8: Image bruitée avec l'éclairage indirect

Par ailleurs, prendre un nombre trop grand de rayons réfléchis **augmente** fortement le **temps d'exécution du programme**, qui dépend, qui plus est, de la rapidité de l'ordinateur utilisé. Une solution à ce problème aurait été d'utiliser la **programmation parallèle** lors de la réalisation de notre boucle for qui parcourt chaque pixel un à un. Dans cette idée, nous avons ainsi essayé d'utiliser la commande

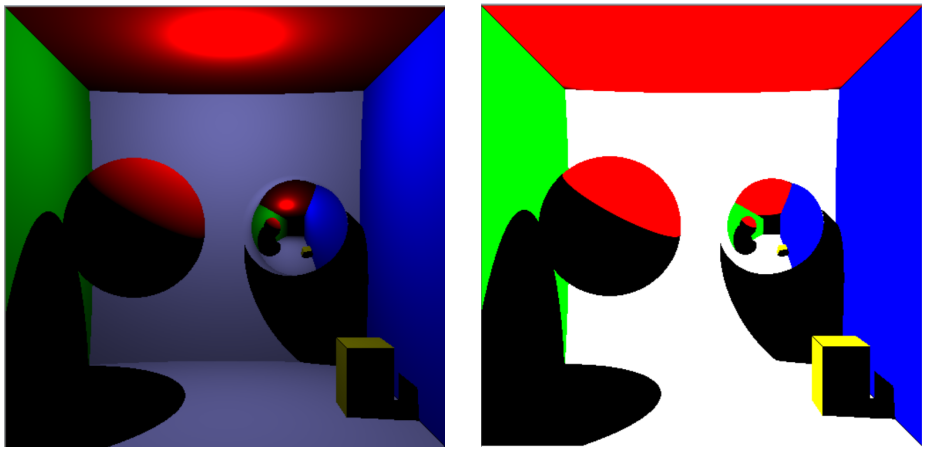
```
#pragma omp parallel for
```

de OpenMP, qui permet de réaliser les itérations sur différents threads en même temps, et ainsi d'utiliser différents processeurs. Néanmmoins, nous n'avons pas réussi à faire marcher cette commande, et sans cette commande, le temps d'exécution du programme est trop long avec un nombre de rayons réfléchis satisfaisant.

C'est pourquoi, dans notre code, **l'implémentation de l'éclairage indirect n'est pas utilisée et est commentée**. Ainsi, finalement, dans la méthode Scene::render, la méthode Scene::get_color est appelée une unique fois pour chaque pixel, et get_color s'occupe uniquement de l'éclairage direct.

4.2 Correction Gamma

Une fois les composantes de couleur récupérées avec la méthode Scene::get_color, à chaque composante de couleur est appliquée la **puissance 1/2.2** afin de rendre le résultat plus réaliste. Il s'agit de la correction Gamma.



(a) Résultat final (b) Résultat sans correction gamma

Figure 9: Comparaison des résultats

De plus, il est important de noter, qu'après avoir appliqué le modèle d'éclairage lambertien ainsi que la correction Gamma aux composantes de couleur, ces

dernières peuvent ne plus être comprises entre 0 et 255. On s'assure alors bien de les ramener dans cet intervalle.

4.3 Miroir

Dans la classe **Shape**, nous avons ajouté **l'attribut boolean miroir**. S'il vaut true, alors l'objet est un miroir.

Soient $\vec{I}$ le rayon lumineux incident au miroir, $\vec{R}$ le rayon réfléchi sur le miroir et $\vec{N}$ la normale au point d'incidence. Le schéma suivant permet de déterminer la formule donnant le rayon réfléchi :

$$\vec{R} = \vec{I} - 2 < \vec{I}.\vec{N} > \vec{N}$$

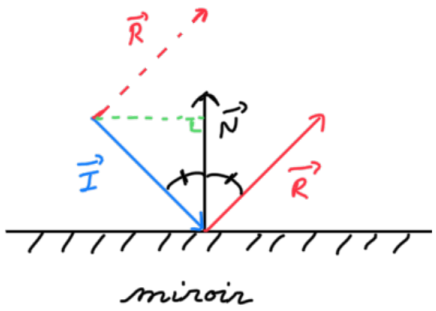


Figure 10: Rayon incident sur le miroir

Le caractère miroir de l'objet est pris en compte dans la fonction **Scene::get_color**, appelée sur le rayon incident et avec un nombre de rebonds égal à 5.

En effet, dans la fonction **Scene::get_color**, nous cherchons le **premier objet qui coupe le rayon incident**, et si cet objet est un **miroir**. Si c'est le cas, alors nous calculons le rayon réfléchi d'après la formule ci-dessus, et appelons la fonction get_color sur ce nouveau rayon et avec un rebond en moins. Lorsque le nombre de rebonds atteint 0, on s'arrête.

Le résultat ainsi obtenu est très satisfaisant : 5 rebonds suffisent pour retourner un beau miroir. En revanche, si l'on commence à mettre plusieurs miroirs dans la scène, il faudrait un nombre infini de rayons pour avoir le même exact résultat que dans la réalité. On pourrait alors imaginer augmenter le nombre de rebonds pour avoir un résultat un peu plus réaliste. Néanmmoins, ceci ne semble pas poser tant de problèmes que cela puisque les reflets d'un miroir dans un autre miroir diminuent au fur et à mesure des rayons réfléchis, et ainsi à partir d'un certain nombre de rayons, les reflets seront très petits et ainsi peu importants.

5 Résultat final

Lors de l'exécution du programme, l'image suivante est affichée, et produite au format BNG (à ouvrir avec IMDisplay par exemple). L'utilisateur peut choisir la taille de l'image en ligne de commande afin d'avoir une taille d'image qui corresponde à ses goûts.

Nous avons donc affiché une scène, avec le mur gauche vert, le plafond rouge, le mur de droite bleu, le sol blanc, et le mur du fond d'une teinte de gris.

La source de lumière est placée au plafond, on la voit au rond de lumière au plafond. Nous n'avons en revanche pas cherché à la représenter physiquement.

Au premier plan, nous avons mis une sphère rouge. Au second plan à droite, nous pouvons apercevoir un cube jaune. Enfin, au fond à droite, nous pouvons voir une sphère miroir.

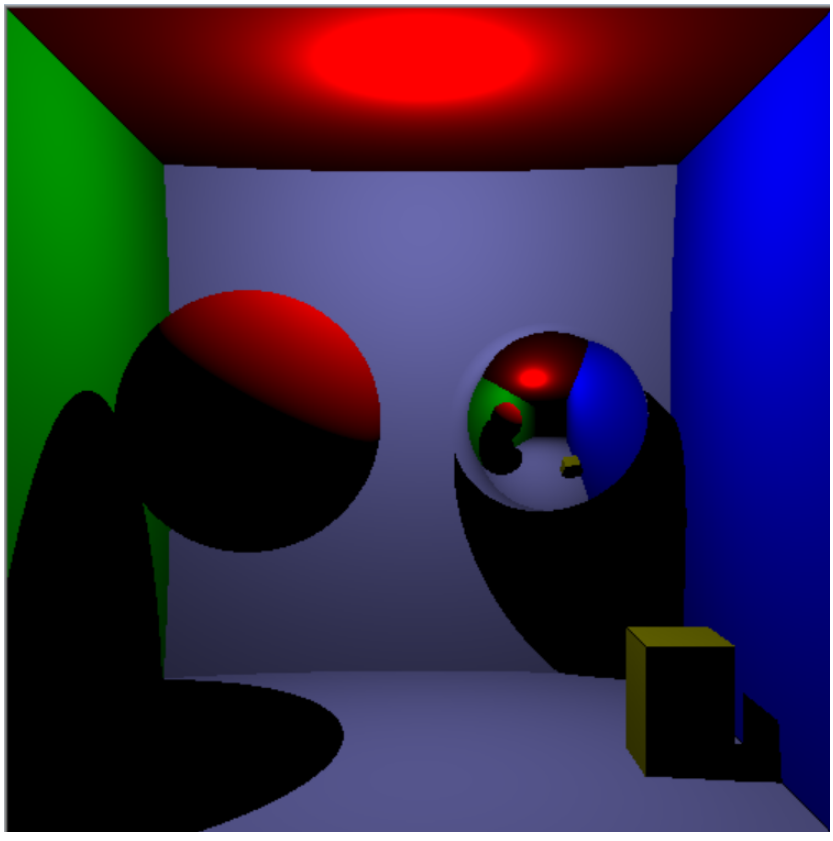


Figure 11: Résultat final

6 Bibliographie

- **Pour les cubes :**
https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-box-intersection.html

- **Pour les sphères et l'éclairage :**
https://perso.liris.cnrs.fr/nicolas.bonneel/teaching.html