ENSIIE

Renaud Caron – Valentin Buchon

Rapport Projet PAP

Lancer de rayon

Année 2022–2023

Table des matières

1 Présentation du projet

1.1 Introduction

Le lancer de rayon est une technique de plus en plus utilisée en informatique pour simuler la propagation de la lumière dans un environnement 3D en utilisant des rayon lumineux partant de la caméra et rebondissant sur les divers objets de la scène. Ce projet est une première implémentation de cette technique en C++ en utilisant la bibliothèque standard et la SDL.

1.2 Utilisation du programme

Une fois compilé, si le programme est lancé sans arguments, il affichera, pour répondre au cahier des charges, cette fenêtre :

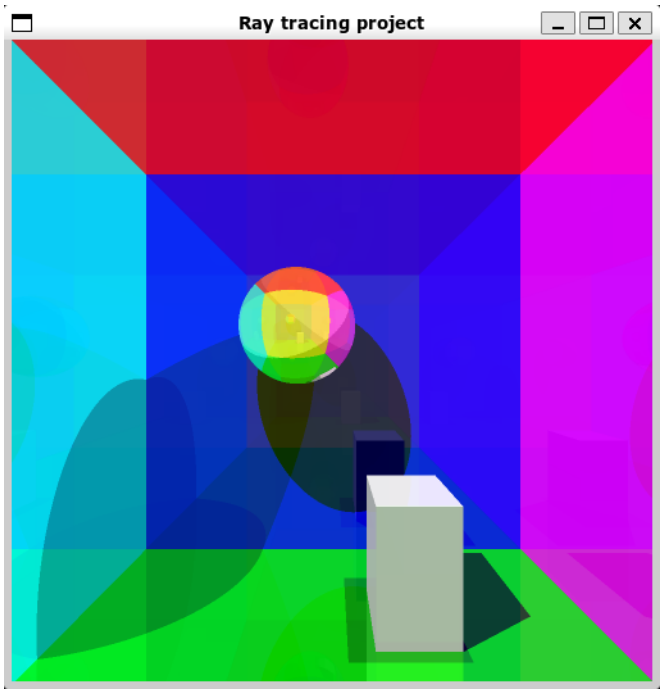


FIGURE 1 – Lancement du programme sans arguments

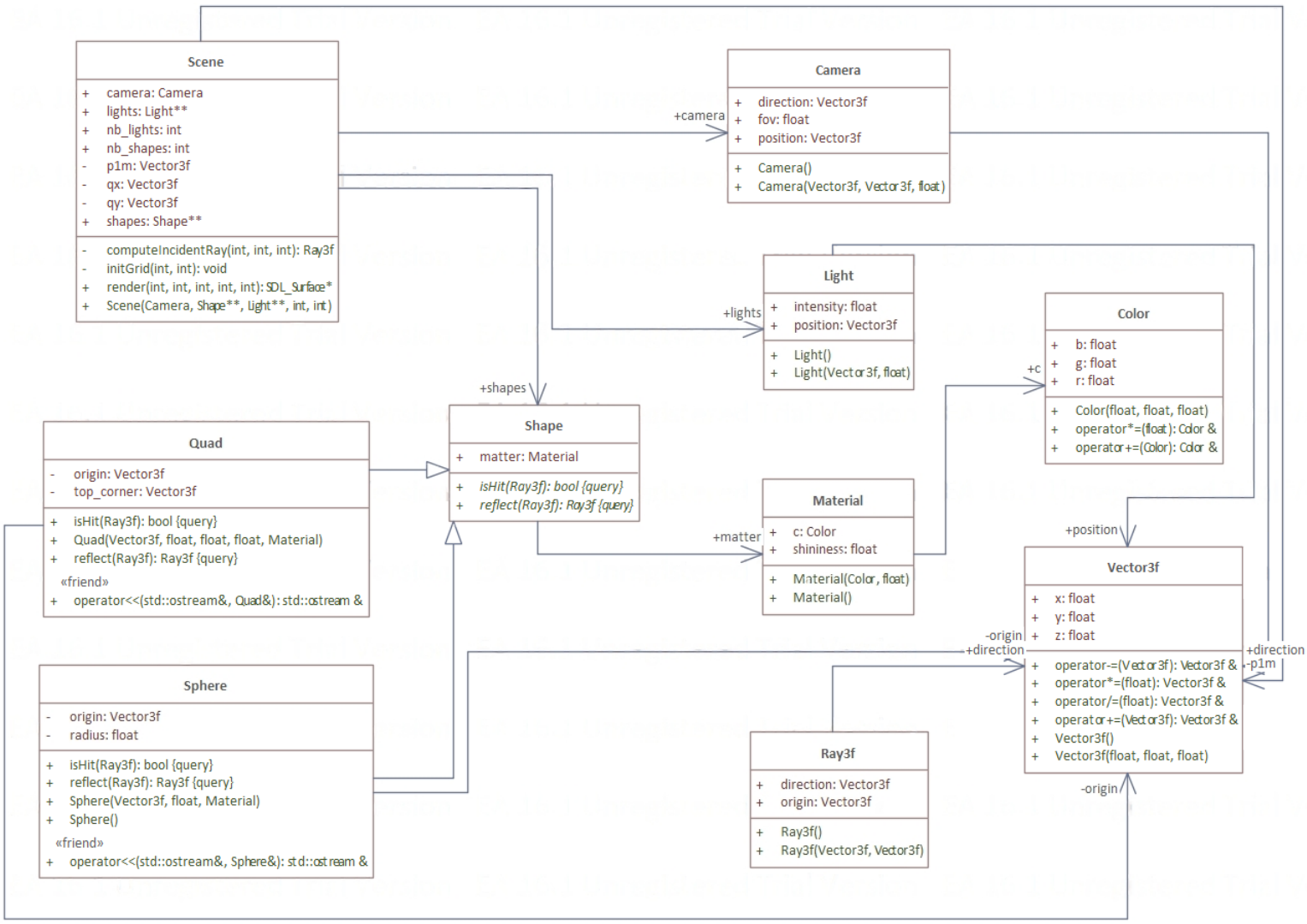
Il peut également être lancé avec les arguments suivants :

1. –**help** : affiche les différents arguments possibles.
2. –**threads n** : permet de préciser le nombre de threads à utiliser (voir 4.1). Si non utilisé, le programme essaiera de déterminer le nombre de threads disponibles seul.
3. –**render-all** : génère toutes les images utlisées dans ce rapport.

2 Choix de conception

2.1 Diagramme UML

En partant du diagramme UML proposé dans le sujet, nous sommes arrivés à celui-ci :



2.2 Color

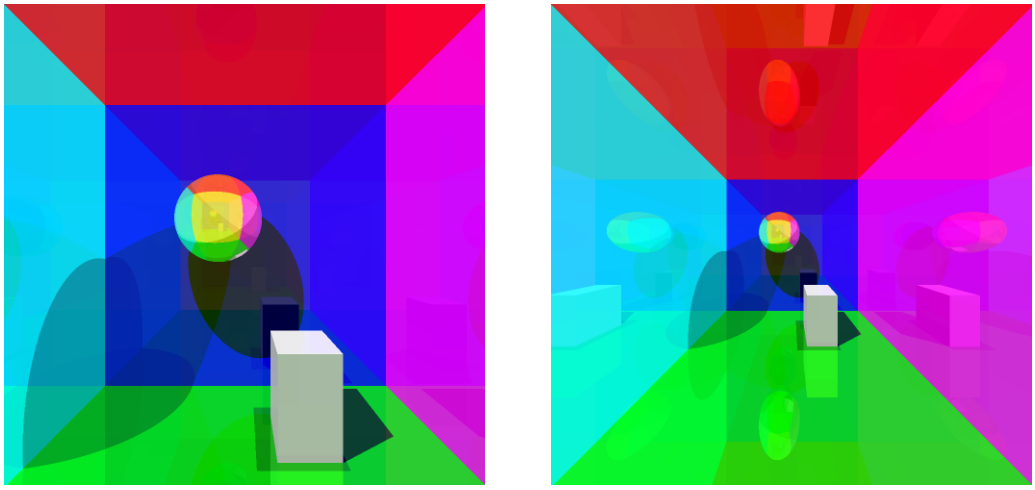
Le diagramme UML du sujet suggérait de stocker les couleurs directement dans Material mais une classe Color supplémentaire a été introduite pour permettre d'effectuer les opérations possibles sur un espace de couleur linéaire directement avec les opérateurs + et *.

2.3 Light

Les sources lumineuses sont implémentées avec une classe et non uniquement avec un Vector3f, pour leur permettre d'avoir une intensité lumineuse (voir 4.3).

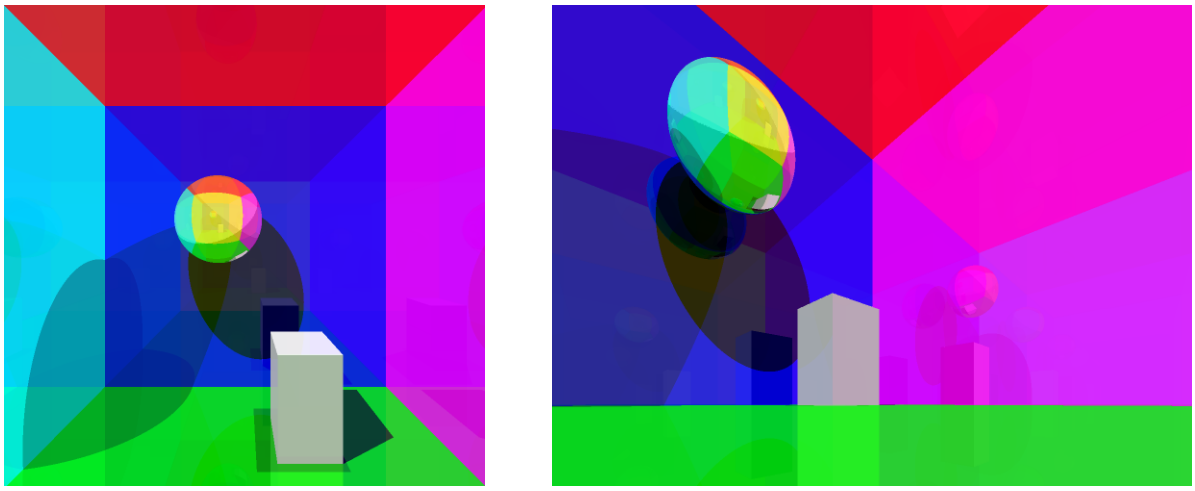
2.4 Camera

En plus de la position et la direction, la classe Camera a un FoV (Field of view - champ de vue), qui est un angle :



(a) FoV de 90° (∼ humain) (b) FoV de 130° (∼ objectif grand angle)

FIGURE 2 – Différents FoV



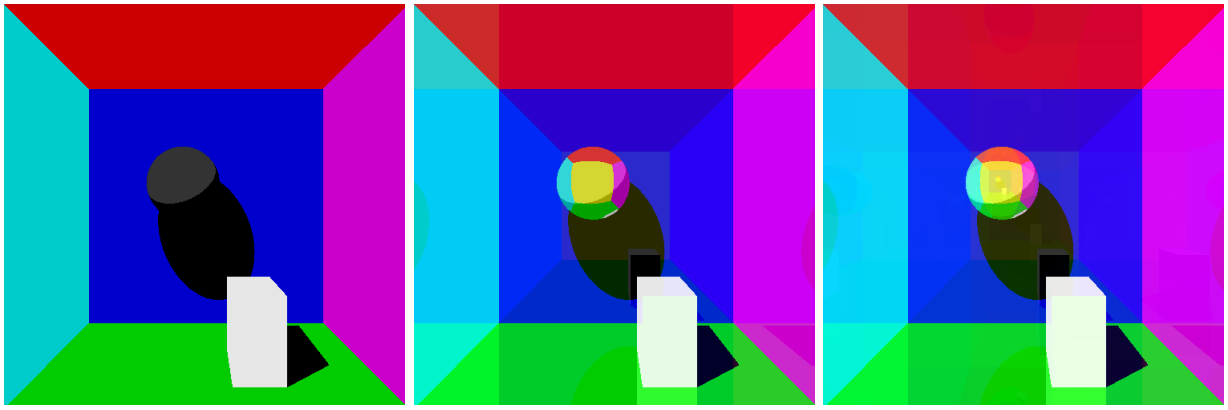
(a) Axe à niveau (b) Contre-plongée, format d'image 4/3

FIGURE 3 – Différentes orientations de la caméra

2.5 Scene

Notre classe Scene possède plus de variables d'instance que celle proposée dans le sujet : on y trouve les nombres de lumières et de formes, ainsi que 3 variables privées utilisées pour le calcul des rayons partant de la caméra et passant par chaque point de la grille correspondant à l'image. Ces dernières sont fixées par la fonction `initGrid()`. Une attention particulière a été portée à l'optimisation de la fonction `render()`, car elle très couteuse en calcul. Cette dernière renvoie une

`SDL_Surface`*, pouvant être affichée ou sauvegardée, et prend en argument les dimensions de l'image souhaitée (voir Figure 3), un nombre de threads et un facteur SSAA (voir 4.1 et 4.2 respectivement), et le nombre de rebonds souhaité :



(a) 0 rebond (b) 1 rebond (c) 5 rebonds

FIGURE 4 – Différents nombre de rebonds maximal

Il est intéressant de constater que la contribution des rebonds successifs à la couleur de chaque pixel décroit rapidement : par exemple, si toutes les figures avaient un indice de réfléxion de 0.5, les 5 premiers rebonds aurait une contribution de $\sum_{k=1}^{5} 0.5^i \approx 97\%$ de la valeur totale, ce qui justifie de limiter le nombre de rebonds maximum, autrement infini.

3 Aspects mathématiques

3.1 Collision

3.1.1 Sphère[1]

Pour les sphères, il faut déterminer si une droite intercte une sphère. Pour cela, posons quelques notations :

— x : points sur la sphère
— c : centre de la sphère
— r : rayon de la sphère

Ainsi l'équation de la sphère est :

$$\|x - c\|^2 = r^2$$

De même, pour une droite représentant un rayon, nous avons :

— x : points sur la droite
— o : origine du rayon lumineux
— d : distance depuis l'origine à la droite
— u : direction du rayon lumineux

Ainsi :

$$x = o + du$$

De ces faits chercher un point qui est sur la droite et sur la sphère devient :

$$\|o + du - c\|^2 = r^2 \Leftrightarrow (o + du - c) \cdot (o + du - c) = r^2$$

Nous avons donc :

$$d^2(u \cdot u) + 2d[u \cdot (o - c)] + (o - c) \cdot (o - c) - r^2 = 0$$

Nous obtenons donc un polynome de degré 2 en d : $ad^2 + bd + c = 0$ avec :

— $a = u \cdot u = \|u\|^2$
— $b = 2d[u \cdot (o - c)]$
— $c = (o - c) \cdot (o - c) - r^2 = \|o - c\|^2 - r^2$

Par conséquent nous obtenons :

$$d = \frac{-[u \cdot (o - c)] \pm \sqrt{(u \cdot (o - c))^2 - \|u\|^2(\|o - c\|^2 - r^2)}}{\|u\|^2}$$

Il est important de noter que la lumière a une direction, et par conséquent que, si d existe, nous ne somme intéressés que par ses valeurs positives. Nous pouvons donc déterminer si un rayon intersecte une sphère par l'existance d'une valeur de d positive, et, dans l'affirmative, que cette intersection est en $x = o + du$ avec le plus petit d positif.

3.1.2 Pavé droit[2]

Concernant les pavés droits, de la même manière que pour la sphère nous posons pour la droite :

— x : points sur la droite
— o : origine du rayon lumineux
— d : distance depuis l'origine à la droite
— u : direction du rayon lumineux

Ainsi :

$$x = o + du$$

Définissons notre pavé droit par les 6 plans engendrés par ses faces, F_{xi}, F_{yi} et F_{zi} $\forall i \in [1, 2]$. Sous l'hypothèse que notre droite n'est parallèle à aucun de ces plans, on obtient 6 valeurs de d_{ai} pour que la droite intersecte chaque plan. Prenons c_1 et c_2 2 coins opposés du pavé du droit, on a alors :

$$\forall a \in \{x, y, z\},\ d_{a1} = \frac{c_{1a} - o_a}{d_a}\ et\ d_{a2} = \frac{c_{2a} - o_a}{d_a}$$

Il suffit alors que les trois premiers plans traversés par la droite ne soient pas parallèles 2 à 2 pour que cette droite entre dans le pavé droit. Autrement dit, si $d_{a1} < d_{a2}$ $\forall a \in \{x, y, z\}$ (il suffit d'inverser les plans si nécessaire), la droite intersecte le pavé droit si et seulement si $\max_{a\in\{x,y,z\}} d_{a1} < \min_{a\in\{x,y,z\}} d_{a2}$.

La lumière ayant une direction, l'intersection existe si et seulement si $\min_{a\in x,y,z} d_{a2}>0$ et, dans ce cas, l'intersection est en $x = o+du$ avec le plus petit d positif appartenant à $\max_{a\in\{x,y,z\}} d_{a1}$,$\min_{a\in\{x,y,z\}} d_{a2}\}$. Enfin, traiter les cas parallèles à part ne semble empriquement pas utile, ce qu'on pourrait expliquer par le fait que le C++ autorise les divisions par 0.0 en renvoyant inf ou -inf selon le signe du numérateur, qui sont compatible avec min, max et les opérateurs de comparaison.

3.2 Réflexion

Une fois que l'on a déterminé sur quelle figure la lumière allait être réfléchie, il faut déterminer le rayon réfléchi. Si l'on pose :

— V : le rayon incident
— N : le vecteur normal à la surface sur lequel V est réfléchi
— R : le rayon réfléchi recherché

On obtient, après, après quelques transformations géométriques classiques[3] :

$$R = 2(V \cdot N)N - V$$

La seule information manquante est donc le vecteur N. Dans notre implémentation, les rebonds à l'intérieur d'une figure ne sont pas pris en charge, ce qu'il faudrait régler implémenter la réfraction par exemple, mais cela simplifie légèrement la situation. Soit I le point d'intersection et O le centre de la sphère :

$$N_{sphere} = O - I$$

Pour le pavé droit, N ne dépendent que de la face sur laquelle est I. Si l'on choisi c_1 comme le coin 'inférieur' (coordonnées minimales) et c_2 comme le coin 'supérieur' (coordonnées maximales), et f la fonction qui vaut 1 en 0 et 0 ailleurs :

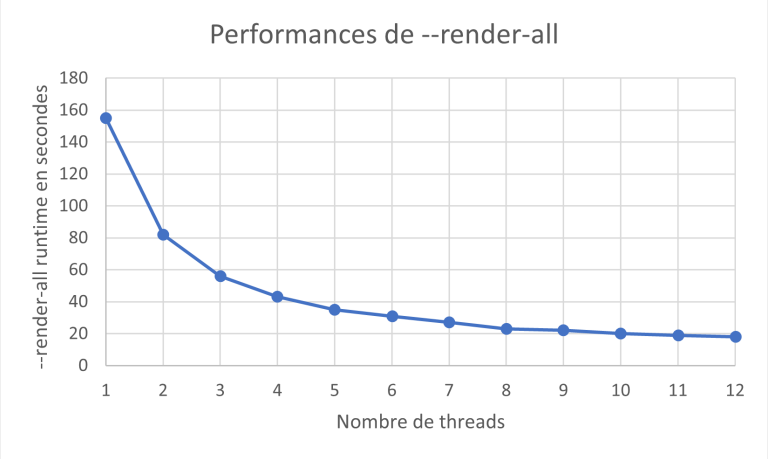
$$N_{pave} = (-f(d_{x1}) + f(d_{x2}), -f(d_{y1}) + f(d_{y2}), -f(d_{z1}) + f(d_{z2}))$$

En pratique, l'égalité de flottants n'ayant pas de sens, on place le 1 ou le -1 au premier d_{ai} suffisament petit rencontré, ce qui élimine également le problème limite des arrêtes et des coins.

4 Fonctionnalités supplémentaires

4.1 Multithreading

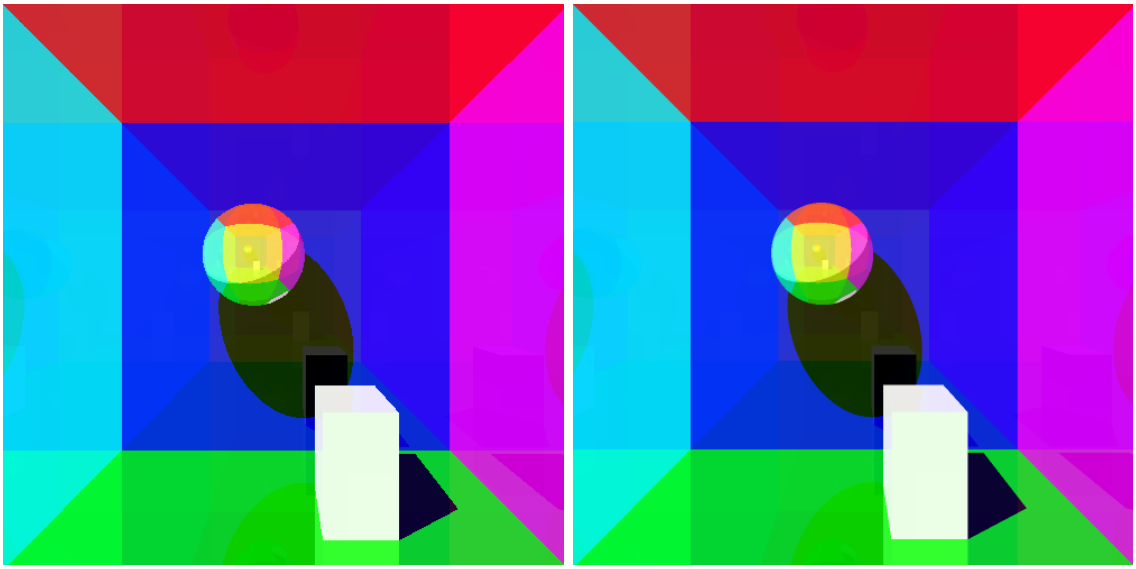
Le lancer de rayon étant très couteux en calculs, nous pouvons diviser le travail nécessaire pour obtenir une image en utilisant le multithreading : chaque thread du processeur peut se charger d'une tranche de l'image finale, qui sont indépendantes. Le nombre de thread souhaité est un argument de la fonction `render()`, et peut-être choisi au lancement du programme, avec l'argument –thread n. Tous les threads sont utilisés par défaut, ou un seul si le programme ne parvient pas à déterminer leur nombre.



Le multithreading réduit bien comme souhaité le temps d'exécution du programme : on passe de 155 secondes avec un unique thread à 18 secondes avec 12 threads (AMD Ryzen 5 5600H, 6C/12T). On peut de plus expliquer les bénéfices dégressifs de plusieurs façons : le système d'exploitation et les applications en fond utilisent une partie du processeur, la fréquence moyenne du processeur baisse légérement sous la charge pour des raisons thermiques et/ou énergétiques (4,20 à 4,05 Ghz observé ici), et seul le rendu bénéficie du multithreading : la conversion en SDL_Surface* et l'écriture sur le disque se font en temps constant par exemple.

4.2 Supersampling Anti-Aliasing

On peut observer un effet d'escalier sur le bord des figures, car un rayon lumineux touche ou ne touche pas une figure, et il n'y a pas d'entredeux. Pour masquer ce problème, on peut utiliser de l'anticrénelage : nous avons choisis d'implémenter le supersampling anti-aliasing. Son principe est très simple : chaque pixel est la moyenne de n sous-pixels, qui peuvent être répartis de différentes façon sur l'image. Le motif le plus simple pour ces sous-pixels est une grille uniforme : il suffit de rendre l'image dans une résolution k fois supérieure, et chaque pixel sera alors la moyenne de k^2 sous-pixels, pour un retour à la résolution initiale.

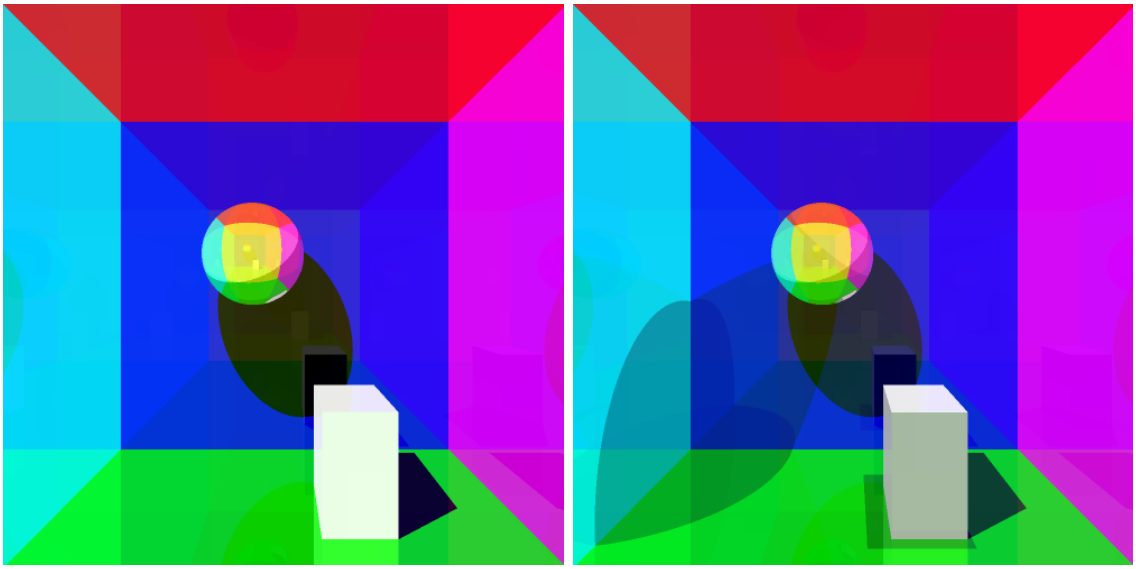


(a) Sans SSAA (b) SSAA avec $k = 2$

FIGURE 5 – Différents facteur de SSAA

4.3 Sources lumineuses multiples

On peut constater en observant l'algorithme décrit dans le sujet qu'ajouter d'autres sources lumineuses, en gardant notre espace de couleur linéaire, est relativement simple : il suffit d'associer à chaque lumière une intensité lumineuse (avec $\sum_{n=1}^{k} intensite_i = 1$), et d'ajouter "la même quantité de couleur" à chaque pixel et à chaque rebond, mais pour chaque lumière et pondéré par ces intensités. Cela ne multiplie pas le nombre de calcul à effectuer par le nombre de lumières, car le premier objet touché par le rayon incident est indépendant des sources lumineuses.



(a) Une source lumineuse (b) Deux sources lumineuses

FIGURE 6 – Différentes sources lumineuses

5 Pour aller plus loin

5.1 L'utilsation du GPU

Afin de rendre les scènes plus rapidement, nous pourrions paralléliser le calcul sur carte graphique (GPU). Les GPU sont très efficaces pour ce type de calcul redondant, en effet, ceux-ci possèdent des milliers de cœurs conçus pour traiter rapidement un grand nombre de tâches mathématiques simples.

5.2 L'ajout de figures

L'ajout de figures supplémentaires serait également possible afin d'améliorer l'application. L'étape logique dans cette direction serait de faire les calculs d'intersection et de reflexion pour les triangles dans l'espace, car on peut approximer toute figure 3D par une collection de triangles.

5.3 Réfraction

L'ajout de matériaux translucides ou transpararents, avec des indices de réfraction, peut très bien s'imaginer. Cela serait plus couteux en calcul : à chaque rebond sur un objet translucide, 2 rayons existeraient : le réfléchi et le réfracté. De plus, il faudrait utitliser un espace de couleur avec des longeurs d'ondes, comme l'espace HSL, si l'on souhaite prendre en compte que la réfraction dépend de la longeur d'onde, ce qui nous permettrait par exemple de simuler un prisme.

5.4 Anti-aliasing adaptatif

Peu importe le motif choisi, le SSAA est extrêmement couteux car il multiplie la quantité de calcul par le nombre de sous-pixels. On pourrait imaginer utiliser un nombre de sous-pixels pour régler ce variable : si, entre 4 pixels adjacents, la couleur varie trop, on calcul pour chaque pixel 4 sous-pixels, et on répète cela jusqu'à ce que la variation s'estompe ou qu'une profondeur maximale

soit atteinte. Cela serait moins direct à mettre en place, mais devrait avoir de meilleurs performance tout en lissant d'avantage les bords.

Références

[1] "Line-Sphere Intersection." Wikipedia, Wikimedia Foundation, 23 novembre 2022, https://en.wikipedia.org/wiki/Line-sphere.

[2] "Ray-Box Intersection." Scratchapixel, https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-box-intersection.

[3] "Reflecting a Vector." 3dkingdoms, 18 janvier 2006, https://3dkingdoms.com/weekly/weekly.php?a=2.