

Rapport projet 2 : Lancer de rayons

Lauric Desauw, Alexandre Drewery, Yan Garito

7 décembre 2018

Table des matières

1	Implémentation	2
1.1	Les Classes	2
1.1.1	Classes standard	2
1.1.2	Classe d'objets	2
1.1.3	Classe d'outils	2
1.2	Modules	2
2	Premiers lancers : modèle simple	3
2.1	Éclairage simple	3
2.2	Réflexion	4
2.3	Colorimétrie	4
3	Diffusion	4
3.1	Première approche : cône de diffusion	4
3.1.1	Idée	4
3.1.2	Problème de la méthode	4
3.2	Seconde approche : diffusion ciblée	4
3.2.1	Idée	4
3.2.2	Résultat	5

Introduction

Ce projet traite du lancer de rayons, une simulation des effets de la lumière sur une scène pour en créer un rendu réaliste. Ce rendu peut ensuite être utilisé pour produire des animations, par exemple dans le cadre d'un film d'animation.

Nous avons dans un premier temps réalisé une implémentation minimale, où seuls l'éclairage et la réflexion sont pris en compte. Nous avons ensuite ajouté une gestion de la colorimétrie et de la diffusion pour un rendu plus réaliste.

1 Implémentation

1.1 Les Classes

Pour mener à bien ce projet nous avons eu besoin de définir différentes classes dont nous expliciterons l'utilité dans la suite du rapport.

1.1.1 Classes standard

Tous d'abord nous avons utilisé des classes de point et de vecteurs, des couleurs ainsi qu'un affichage graphique à l'aide de la bibliothèque SFML 2.5.1. Ces classes sont standard et ne seront pas détaillées ici.

1.1.2 Classe d'objets

L'objectif du lancer de rayon étant de faire le rendu graphique d'une scène nous avons défini des classes d'objets afin de créer cette scène.

Nous avons donc une classe abstraite `Shape` qui nous permet de manipuler un objet sans avoir besoin de connaître sa forme et ses caractéristiques. Puis nous avons des classes filles qui définissent des objets tels que des plans finis (`Plane`) et des boules (`Ball`).

1.1.3 Classe d'outils

Nous avons défini une classe de rayon de lumière contenant un point d'origine, un vecteur directeur et une couleur.

Comme cela demanderait trop de calcul de lancer tous les rayons partant depuis une source nous utiliserons le retour inverse de la lumière. Les objets de `Lightray` seront donc des rayons inverse (allant vers la source de lumière).

Nous avons aussi défini une classe `Hit` contenant un point où un rayon de lumière à intersecter un objet ainsi que la normale de l'objet en ce point et l'objet touché (l'objet touché est donné afin d'avoir accès à sa texture et sa couleur).

1.2 Modules

Nous avons séparés nos fonctions de sorte à avoir différents modules ayant un objectif clair.

Dans un premier temps nous avons fait un module `colorization` qui contient les fonctions suivantes.

```
Color reflected_color(Lightray ray,
                      Hit& real_hit,
                      const std::vector<Shape*>& shapes,
```

```

        const std::vector<Shape*>& sources);

Color diffuse_color(Lightray ray,
                    Hit& real_hit,
                    const std::vector<Shape*>& shapes,
                    const std::vector<Shape*>& sources);

Color compute_color(Lightray ray,
                    Hit& real_hit,
                    const std::vector<Shape*>& shapes,
                    const std::vector<Shape*>& sources);

```

Où `reflected_color` calcule la couleur en un point de la scène d'après les règles de la réflexion (voir 2.2), `diffuse_color` calcule la couleur en un point de la scène d'après les règles de diffusion (voir 3) et `compute_color` calcule la couleur en tenant compte des deux couleurs précédentes.

Ensuite nous avons fait un module `rendering`, utilisant le module `colorization`, qui à partir d'un monde en donne un rendu graphique.

2 Premiers lancers : modèle simple

2.1 Éclairage simple

Dans un premier temps nous allons afficher tous les points accessibles depuis la caméra. Pour cela on lance des rayons depuis la caméra pour chaque pixel de l'écran et on regarde si oui ou non cela touche un objet de la scène. Si le rayon part dans le vide alors le pixel est affiché noir, sinon il est affiché de la couleur de l'objet atteint.

Pour faire cela nous avons définie une méthode virtuelle pur dans la classe `Shape` :

```
Hit check_intersection(Lightray, bool);
```

Pour la classe `Ball` cette méthode consiste à résoudre selon la variable t l'équation :

$$\|\mathbf{AC} - t \times \mathbf{u}\|_2 = 0$$

où A est l'origine du rayon, C le centre de l'objet de classe `ball` et \mathbf{u} le vecteur directeur du rayon de lumière. Si cette équation n'a pas de solution alors le rayon n'a pas intersecté la boule, si elle a une solution réelle alors le rayon est tangent à la boule et on va considérer qu'il ne l'intersecte pas et si elle a deux solutions réelles on va considérer t_1 la plus petite des deux solutions. On a ensuite que le point d'intersection est :

$$I = A + t_1 \times \mathbf{u}$$

Pour la classe `Plane` nous commençons par regarder l'intersection du rayon avec le plan infini contenant notre objet de classe `Plane`. Si le rayon n'est pas confondu au plan le point d'intersection est :

$$I = \frac{\langle \mathbf{AP} | \mathbf{n} \rangle}{\langle \mathbf{n} | \mathbf{u} \rangle} \times \mathbf{u} + A$$

où A est l'origine du rayon, P un point quelconque du plan et \mathbf{n} la normale du plan. Il reste ensuite à vérifier si ce point appartient bien au plan fini de classe `Plane` et cela se fait facilement en vérifiant quatre produit scalaire.

2.2 Réflexion

Nous souhaitons maintenant ajouter un effet de réflexion à notre image. Pour cela lorsque l'on lance un rayon depuis la caméra et qu'il intersecte un objet on va s'intéresser au rayon réfléchi selon les lois de Descartes. Si ce rayon touche une source de lumière alors nous allons utiliser les informations de cette source pour en déduire la couleur du pixel. Si le rayon ne touche rien alors nous en déduisons que le pixel est noir. Enfin si le rayon touche un autre objet nous allons appliquer récursivement cet algorithme en sachant que si au bout de 3 étapes de récursions aucune source n'a été rencontré on considérera que le rayon a été totalement absorbé et donc le pixel sera affiché noir.

Pour une scène avec quatre plans lumineux formant un couloir et deux boules réfléchissantes au centre on obtient le résultat suivant (voir figure 1).

2.3 Colorimétrie

3 Diffusion

3.1 Première approche : cône de diffusion

3.1.1 Idée

Pour représenter le cône de diffusion, on part du vecteur normal \vec{n}_1 que l'on obtient via les lois de Descartes. On obtient ensuite $P = \vec{n}_1^\perp = \text{Vect}\{\vec{n}_2, \vec{n}_3\}$ puis on fait tourner \vec{n}_1 selon l'axe \vec{n}_2 puis selon l'axe \vec{n}_3 pour obtenir un vecteur \vec{n} . En effectuant « toutes »¹ les rotations autour de chacun de ces axes, on génère tout le cône de diffusion. On lance ensuite tous les rayons du cône de diffusion généré :

- initialement, on a trois variables entières red, green, blue qui sont initialisées à 0;
- si le rayon considéré atteint une source de couleur codée par le triplet red_source, green_source, blue_source, on incrémente red, green, blue respectivement par red_source, green_source, blue_source et on incrémente un compteur c de 1;
- sinon, on ne fait rien.

Une fois tous les rayons lancés, on divise red, green et blue par c et on renvoie Color(red, green, blue).

3.1.2 Problème de la méthode

Le problème de cette méthode est, comme montré en figure ??, que la précision n'est pas suffisante, ce qui mène à des artefacts importants. Néanmoins, même si ce problème avait été réglé, la méthode est beaucoup trop gourmande pour le gain obtenu. En effet, la méthode de diffusion ciblée crée des images de toute aussi bonne qualité avec un temps de calcul quasiment nul.

3.2 Seconde approche : diffusion ciblée

3.2.1 Idée

Plutôt que de calculer toutes les rotations, on n'utilise qu'un rayon pour déterminer la couleur du pixel. Comme montré en figure ??, s'il n'y a qu'une seule source de lumière :

- on trace la demi-droite d partant du point d'impact du rayon inverse vers la source de lumière;

1. A précision d'un degré, ce qui donne une précision de 10^{-2} sur les radians et en ignorant les rotations inutiles (de plus de 90)

- on cherche le premier objet que cette d intersecte ;
- si c'est la source ciblée, elle éclaire le point ;
- sinon, elle n'éclaire pas le point.

Notons k_a le coefficient d'absorption de l'objet et $k_d = 1 - k_a$. Notons également s_c la couleur de la source et o_c la couleur de l'objet. Enfin, notons c le cosinus de l'angle entre le vecteur donné par les lois de Descartes et un vecteur directeur de d .

- Si le point est éclairé par la source, on décrète que sa couleur est $(k_a + k_d \times c) \times s_c \times o_c$.
- Sinon, le point est dans l'obscurité et on décrète alors que sa couleur est $k_a \times s_c \times o_c$.

Dans le cas où il y a plusieurs sources, on itère ce processus pour chacune des sources et on somme les couleurs obtenues afin de calculer la couleur du point.

3.2.2 Résultat

On peut voir certains des résultats obtenus en figure ?? et figure ??.

Le problème principal de cette méthode est l'obtention de la demi-droite d . Pour plus de simplicité, nous avons décidé que cette demi-droite relierait le point touché par le rayon inverse et l'isobarycentre de la source. Or comme vu en figure ??, cela pose certains problèmes : le dégradé est trop simpliste du fait que d traverse l'objet lui-même et les pixels se considèrent comme étant dans l'ombre alors qu'en vérité, ils devraient être éclairés. Une amélioration possible serait alors de trouver le point de la source le plus proche du point touché P qui pourrait éclairer ce dernier. On pourrait ensuite altérer la couleur selon le point P_s de la source qui éclaire effectivement le pixel. Par exemple, si O est l'isobarycentre de la source, on pourrait décréter que la couleur du pixel dépend également du cosinus de l'angle entre \overrightarrow{PO} et $\overrightarrow{PP_s}$ en multipliant s_c par ce facteur avant de décider de la couleur du pixel.

Conclusion

Documents Annexes



FIGURE 1 – Affichage d'une scène avec réflexion totale de la lumière