Université de Strasbourg

Rapport Projet 150h

Lancer de rayons en temps réel

Nathan ROTH

Projet encadré par M. Sylvain THERY et M. Pascal GUEHL

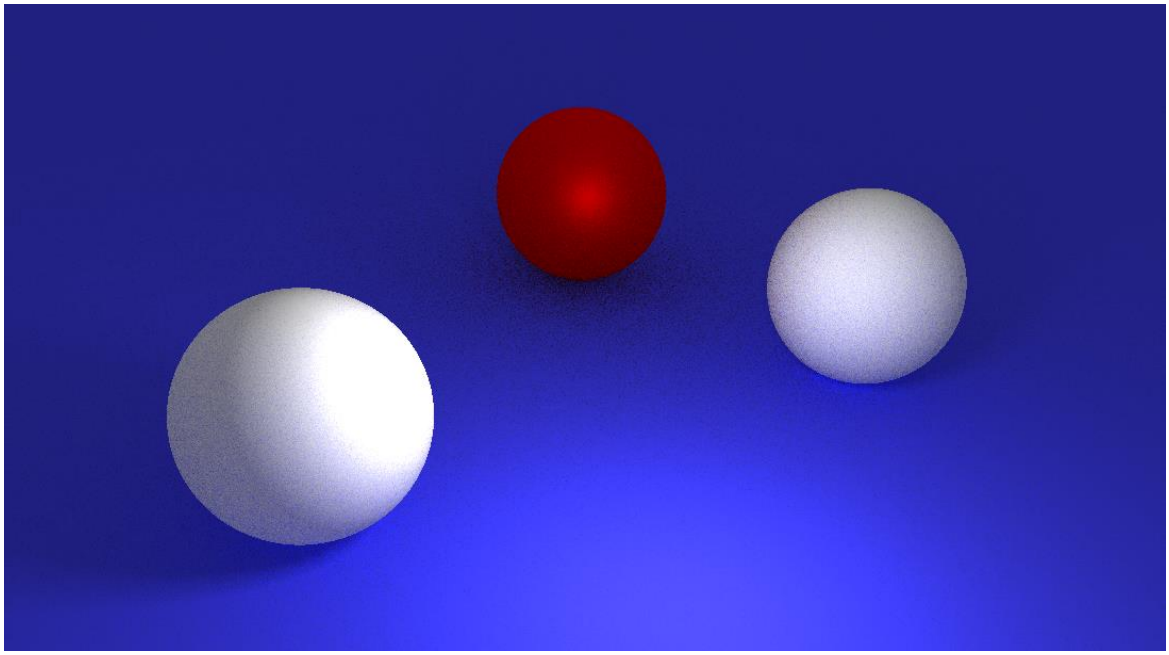
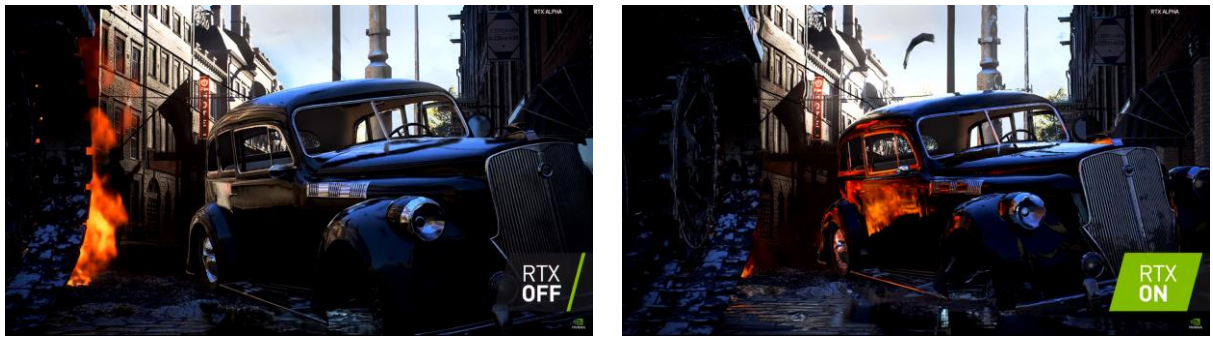


Table des matières

Introduction

Contexte

Le lancer de rayon (*raytracing*) est une méthode de synthèse d'image 3D permettant d'obtenir des rendus de grande qualité. Mais cette qualité se paye au prix du temps de calcul. En effet, un rendu photoréaliste réalisé avec cette méthode, pour un film par exemple, peut nécessiter l'équivalent de plusieurs heures de travail pour un bon ordinateur de bureau. Il est alors évident que son usage est inadapté à des applications nécessitant un affichage à fort taux de rafraîchissement, comme les jeux vidéo par exemple. Cependant, la puissance des récentes cartes graphiques, notamment les cartes de la série RTX de *nVidia*, ont permis l'introductions de ces techniques dans certains jeux. Ces dernières offrent désormais la possibilité de traiter des effets de lumière jusqu'alors fortement approximés et souvent peu convaincants comme les reflets sur un miroir ou l'illumination globale.



Images du jeu Battlefield 5 (Electronic Arts). On constate qu'un élément absent du champ de vision de la caméra ne peut être réfléchis avec une méthode comme Screen Space Reflection *en* deffered shading*. RTX, l'API de* raytracing *de* nVidia*, en est capable.*

L'objectif de ce projet est d'implémenter un lancer de rayon en temps réel afin d'observer l'impact d'un tel algorithme sur les performances de l'ordinateur.

Pour ce faire, nous avons choisi une implémentation en WebGL, permettant d'utiliser le GPU directement via les *vertex* et *fragment shaders*. Cependant, WebGL ne servira que de canevas à notre application en affichant uniquement le rendu final de notre moteur.

La librairie utilisée est EasyWebGL, crée par M. Sylvain Théry. Elle permet d'utiliser plus simplement WebGL et offre une API d'interface 2D pour un contrôle en direct de l'application.

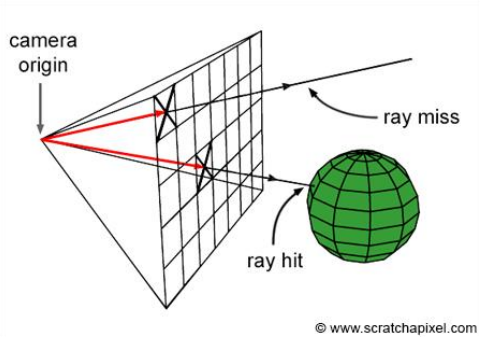
Architecture du programme

Comme expliqué plus haut, OpenGL nous sert ici principalement d'interface avec le GPU. L'idée est d'écrire tout le moteur en un seul *shader* GLSL et d'afficher le résultat sur un simple quad couvrant le champ de vision de la caméra. Le *fragment shader* utilisé écris directement dans le *framebuffer* final.

Il est important de noter qu'en OpenGL, le *fragment shader*, lancé une fois par pixel de l'image, est le programme qui décide de la couleur du pixel (ou du texel s'il écris dans un *buffer* et non directement sur l'écran). Ces appels sont parallélisés sur le GPU, permettant à l'ordinateur de calculer plusieurs pixels à la fois (le nombre variant en fonction du GPU). Il n'existe donc pas de communication directe entre les différents processus, et l'ordre d'appel n'est pas garanti.

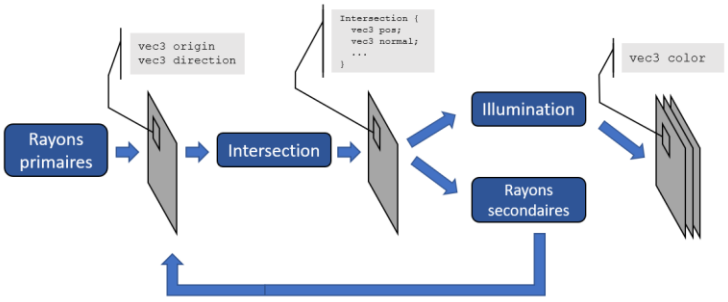
Fonctionnement général de l'algorithme

La première étape dans la conception du moteur est le lancer des rayons primaires. On envoie un rayon par pixel de l'image, avec comme origine le foyer de la caméra.



Une fois lancé, le rayon intersecte (ou non) la scène. A partir de cette intersection, plusieurs autres rayons sont envoyés dans la scène afin de calculer les différents effets de lumières. Ces rayons secondaires sont susceptibles de générer d'autres rayons, puis d'autres, et ainsi de suite jusqu'à atteindre une certaine limite prédéfinie.

L'aspect récursif du lancer de rayon le rend peu adapté à une implémentation sur GPU. Une parade intéressante a été proposée par Radeon avec son API Radeon-Rays. Le principe est de diviser les différentes étapes du lancer de rayon classique en plusieurs *shaders* différents, en transmettant les données via des *buffers* puis en accumulant progressivement le résultat final. Chacun de ces *shader* s'occupe d'une partie précise de l'algorithme, comme la génération des rayons primaires ou le calcul de la BRDF. Certaines étapes peuvent alors être répétées afin de faire converger normalement l'algorithme. Chaque texel d'un *buffer* représente un pixel, on a donc une valeur (flottant, vecteur…) par texel par *buffer* par étape.



Cette particularité va poser un problème dès le premier rebond, à partir duquel l'algorithme doit générer plusieurs sous chemins. En effet, pour augmenter la qualité d'une image générée par « lancer de rayon », il faut augmenter le nombre de rayons à chaque rebond afin de mieux « sonder » l'environnement. A lieu d'augmenter

le nombre de rayons par passage, on augmente le nombre de boucles, une boucle représentant un chemin différent. On génère alors séquentiellement (par *thread* de GPU) tous les chemins nécessaires.

Une architecture similaire a été implémenté pour ce projet, mais seul le principe général à été retenu (un chemin par boucle).

Lancer de rayon

Calcul d'un rayon primaire

La direction du rayon est calculée en fonction du pixel étudié et de l'orientation de la caméra. Soient la taille (w, h) de la fenêtre de rendu, la position (x, y) du pixel courant où $x \in [0, w[$ et $y \in [0, h[$ et les matrice camera C et projection P . On cherche $\mathbf{o}$ l'origine du rayon et $\mathbf{d}$ sa direction.

On transpose tout d'abord x et y en coordonnées réelles dans l'intervalle $[-1,1]$, où 0 est le centre de la surface de rendu.

$$P_x = \frac{2x}{w} - 1 \qquad P_y = -\left(1 - \frac{2y}{h}\right)$$

On obtient un vecteur $\mathbf{r}$ dans l'espace de projection de la forme $\left(P_x, P_y, -1,1\right)$.

Pour obtenir $\mathbf{d}$:

$$\mathbf{d} = C^{-1}P^{-1}\mathbf{r}$$

Pour obtenir $\mathbf{o}$, on utilise la propriété de la matrice selon laquelle la position de la caméra est le vecteur correspondant à la dernière colonne de C, d'où

$$\mathbf{o} = C^{-1} * (0,0,01)$$

Les matrices C et P effectuent pour nous les opérations de changement de base en fonction de la position et du champ de vision de la caméra, et des paramètres de perspective. [1]

Intersecter un objet

Une fois le rayon lancé, on doit tester s'il existe une intersection entre lui et un objet de la scène. Le moteur développé pour ce projet gère certains objets paramétriques, comme les sphères ou les plans. Les intersections avec les triangles sont aussi gérées, en prévision de l'importation de maillages.

A chaque intersection, une structure est renvoyée par effet de bord. Cette structure contient la position de l'intersection dans le repère global, la normale de la surface au point d'intersection, la distance entre l'intersection et l'origine du rayon, et l'indice du matériau de la face intersecté. Toutes ces informations sont nécessaires au calcul d'illumination et à la génération de nouveaux rayons. Les formules ci-après sont inspirées du site Scratchpixel [2].

Soit un rayon $\mathbf{o} + \mathbf{d}t$ où $\|\mathbf{d}\| = 1$, $\mathbf{o}$ l'origine du rayon et $\mathbf{d}$ sa direction. Soit $\mathbf{p} = \mathbf{o} + \mathbf{d}t$ le point d'intersection avec un objet. L'objectif des routines d'intersection est de trouver t pour calculer $\mathbf{p}$.

Sphère

Soit une sphère ce centre $\mathbf{c}$ et de rayon r.

$$|\mathbf{p} - \mathbf{c}|^2 + r^2 = 0$$

$$|\mathbf{o} + \mathbf{d}t + \mathbf{c}|^2 + r^2 = 0$$

$$a = 1, \qquad b = 2\mathbf{d}(\mathbf{o} - \mathbf{c}), \quad c = |\mathbf{o} - \mathbf{c}|^2 + r^2$$

Il ne reste qu'à résoudre $t = \min\left(\frac{-b \pm \sqrt{\Delta}}{2a}\right)$ si pour $\Delta = b^2 - 4ac$ on a $\Delta \geq 0$ pour trouver l'intersection la plus proche. La normale au point d'intersection est le vecteur unitaire $\mathbf{p} - \mathbf{c}$.

Plan

Soit un plan défini par sa normale $\mathbf{n}$ et passant par un point $\mathbf{p_0}$. On a alors

$$(\mathbf{p} - \mathbf{p_0}) \cdot \mathbf{n} = 0$$

$$(\mathbf{o} + \mathbf{d}t - \mathbf{p_0}) \cdot \mathbf{n} = 0$$

$$t = \frac{(\mathbf{p_0} - \mathbf{o}) \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}}$$

Triangle

Pour déterminer si nous intersectons un triangle $[\mathbf{v_0}, \mathbf{v_1}, \mathbf{v_2}]$, nous déterminons d'abord la position du point d'intersection $\mathbf{p_t}$ sur le plan formé par ce triangle. Puis au moyen de produits scalaire successifs (sens trigonométrique) entre les arêtes $[\mathbf{v_i}, \mathbf{v_{i+1}}]$ et le vecteur vers le point d'intersection $[\mathbf{v_i}, \mathbf{p_t}]$, nous déterminons si $\mathbf{p_t}$ est à l'intérieur du triangle. Il y est si tous les produits scalaires étaient de signe positif. La normale du triangle est $\mathbf{v_1} - \mathbf{v_0} \wedge \mathbf{v_2} - \mathbf{v_0}$.

Génération des rayons secondaires

Après chaque intersection, et si la limite de rebond n'a pas été atteinte, un nouveau rayon est envoyé depuis ce point. On choisit une direction au hasard dans l'hémisphère orientée selon $\mathbf{n}$, la normale au point d'intersection $\mathbf{p}$. Cette étape est problématique lorsque l'on travaille uniquement sur GPU. En effet, ces derniers ne peuvent générer de nombres aléatoires. Il faut donc soit les importer, soit trouver une méthode de génération. Afin de réduire au maximum le nombre de calculs à effectuer, la première méthode a été utilisée. Ces vecteurs sont calculés sur CPU avant le démarrage de l'application et envoyés sur le GPU dans une texture 3D contenant $w \times h \times k$ vecteurs unitaires aléatoires, où k est une valeur fixée arbitrairement.

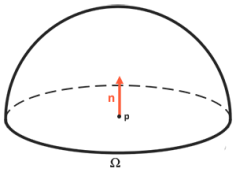
Soit un couple de nombres aléatoires u et v,

$$\phi = 2\pi u$$

$$\theta = \mathrm{acos}(1 - 2v)$$

$$\mathbf{r_{rand}}(\sin(\phi)\sin(\theta), \cos(\phi)\sin(\theta), \cos(\theta))$$

Une fois dans le *shader*, on teste si $\mathbf{r_{rand}}$ est du « même coté » que $\mathbf{n}$, c'est-à-dire si $\mathbf{r_{rand}} \cdot \mathbf{n} > 0$. Si ce n'est pas le cas, $\mathbf{r_{rand}} = -\mathbf{r_{rand}}$. On a alors un nouveau rayon d'origine $\mathbf{p}$ et de direction $\mathbf{r_{rand}}$.



Hémisphère Ω centrée sur p et orienté selon n

En pratique, à chaque nouveau vecteur, le *shader* pioche dans la texture le vecteur i suivant aux coordonnées (x, y, i), où $i \equiv k$. Cette fonction est également utilisée pour l'échantillonnage des sources lumineuses [voir Soft Shadows].

Cette méthode peut être largement optimisée, notamment en réduisant la taille de la texture et en la répétant. Une autre méthode possible serait l'utilisation d'un bruit de Perlin à très haute fréquence, permettant son exécution sur GPU. Tous les calculs y seraient cependant également transférés.

Illumination

Equation de rendu

On rappelle l'équation de rendu qui définit la radiance émise depuis un point $\mathbf{p}$ dans l'espace,

$$L_o(\mathbf{p}, \mathbf{w_o}, \mathbf{w_i}) = L_e + \int_\Omega L_i(\mathbf{p}, \mathbf{w_i}) f_r(\mathbf{w_o}, \mathbf{w_i}) \cos\theta \, d\omega$$

où $\mathbf{p}$ est le point d'intersection, $\mathbf{w_i}$ est le vecteur de $\mathbf{p}$ vers la source lumineuse, $\mathbf{w_o}$ le vecteur vers l'observateur ou vers la surface captant la lumière réfléchie par f_r. Le terme L_e représente la lumière émise par l'objet intersecté, qui nous ignorerons ici car non supporté par le moteur. Le terme $\cos\theta$ vaut $\mathbf{n} \cdot \mathbf{w_i}$. Ω représente l'hémisphère orienté selon $\mathbf{n}$. f_r est la BRDF (*Bidirectional Reflection Distribution Function*), la fonction qui définit l'effet de la lumière sur l'objet ainsi que le comportement de cette dernière lorsqu'elle est réfléchie. Pour résoudre cette intégrale, on utilise l'estimateur de Monte-Carlo [3](Section 14.1.3).

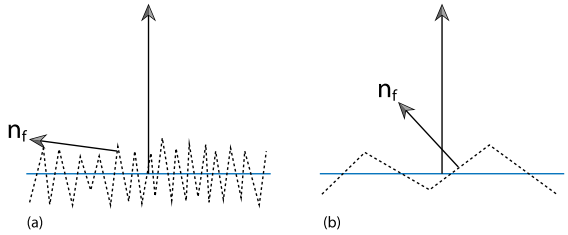
$$L_o(\mathbf{p}, \mathbf{w_o}) = \frac{1}{N} \sum_k^N \frac{L_i(\mathbf{p}, \mathbf{w_k}) f_r(\mathbf{w_o}, \mathbf{w_k}) \cos\theta}{P(\mathbf{w_k})}$$

Le principe est d'échantillonner N fois l'hémisphère et de pondérer ces échantillons à l'aide d'une PDF (*Probability Distribution Function*). En effet, chaque rayon secondaire n'a pas la même influence sur l'illumination du point d'intersection. P est dépendant de f_r : si f_r est une surface peu rugueuse, seuls les rayons proches du vecteur de réflexion de la lumière contribuerons.

Cook-Torrance BRDF

Notre moteur utilise le modèle Cook-Torrance [4] qui émule la théorie des micro-facettes. Cette dernière, physiquement réaliste et largement utilisée en imagerie 3D, considère que toute surface est composée de plus petites faces dont la moyenne des normales vaut $\mathbf{n}$. L'écart d'orientation entre les normales défini la rugosité de

la surface. Plus l'écart est grand, plus la surface est rugueuse. Dans les figures si dessous, les surfaces à gauche sont plus diffuses (rugueuses), celles de droites plus réfléchissantes (lisses).



PBR [3](*Figure 8.12*)

LearnOpenGL [5]

En se basant sur le facteur de rugosité du matériau, il est possible de calculer la distribution de normales alignées avec $\mathbf{h}$. Plus cette distribution sera grande (la rugosité faible) plus la surface sera réfléchissante(lisse). $\mathbf{h}$ est le vecteur halfway (à mi-chemin) entre $\mathbf{w_o}$ et $\mathbf{w_i}$, qui vaut $\frac{\mathbf{w_o}\cdot\mathbf{w_i}}{|\mathbf{w_o}+\mathbf{w_i}|}$. Si $\mathbf{w_o}$ est en direction de $\mathbf{p}$ et $\mathbf{w_i}$ en direction de la lumière, $\mathbf{w_o} + 2\mathbf{h} = \mathbf{w_i}$. $\mathbf{h}$ est donc le vecteur représentant la normal d'une surface qui réfléchirait l'un des deux vecteur en un autre.

Le modèle Cook-Torrance est calculé selon 3 termes principaux D, F et G:

$$f_r(\mathbf{w_i}, \mathbf{w_o}) = \frac{DGF}{4(\mathbf{n}\cdot\mathbf{w_i})(\mathbf{n}\cdot\mathbf{w_o})}$$

Distribution des normales

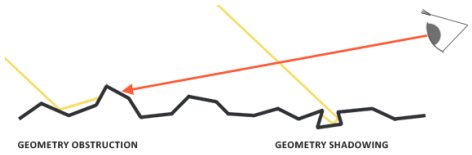
D est la fonction de distribution des normales alignées avec $\mathbf{h}$. C'est elle qui influence en majeur partie l'aspect du matériau. Il existe plusieurs formulations pour ce terme [6], la formule ci-après est celle de Trowbridge-Reitz [7]. α est le facteur de rugosité.

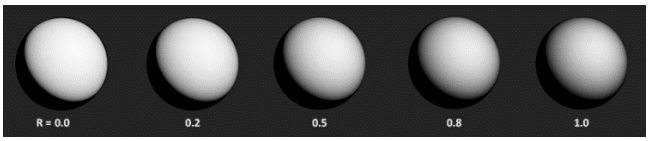
$$D(\mathbf{n}, \mathbf{h}, \alpha) = \frac{\alpha^2}{\pi((\mathbf{n}\cdot\mathbf{h})^2(\alpha^2-1)+1)^2}$$

Plus le nombre de normales alignées avec $\mathbf{h}$ est grande, plus la surface est lisse. Un plus grand nombre de faces réfléchissent la lumière dans une même direction, à la façon d'un miroir.

Fonction géométrique

G approxime la proportion des faces qui en occultent d'autres. Une surface très rugueuse peut voir ses micro-facettes obstruer la lumière réfléchie par ses voisines et donc réduire la quantité de lumière totale réfléchie.





La formulation ici est une combinaison de la méthode de Smith pour l'expression de G et de Schlick-Beckmann [8](Eq. 19) pour G_1.

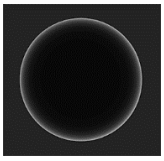
$$G(\mathbf{n}, \mathbf{w_o}, \mathbf{w_i}, k) = G_1(\mathbf{n}, \mathbf{w_o}, k)G_2(\mathbf{n}, \mathbf{w_i}, k)$$

$$G_1(\mathbf{n}, \mathbf{w}, k) = \frac{\mathbf{n} \cdot \mathbf{w}}{(\mathbf{n} \cdot \mathbf{w})(1-k)+k}$$

$$k = \sqrt{2\alpha^2/\pi}$$

Equation de Fresnel

F est le terme de Fresnel. Cette équation, en fonction de l'indice de réfraction du matériau (IOR), donne le ratio entre la lumière réfléchie et la lumière réfracté selon l'angle depuis lequel on observe la surface. La loi de Fresnel implique que pour tout matériau, plus on l'observe à un angle rasant, plus il réfléchit la lumière.



Intensité du ratio de Fresnel sur une sphère. Il est très important aux angles rasants où la quasi-totalité de la lumière y est réfléchie, et non diffractée.

Calculer très précisément ce ratio est complexe, nous utilisons alors l'approximation de Schlick [8](Eq. 15)

$$F(F_0, \mathbf{w_i}, \mathbf{h}) = F_0 + (1 - F_0)(1 - \mathbf{w_i} \cdot \mathbf{h})^5$$

F_0 est la réflectance de base du matériau. Il est possible de la calculer facilement pour des matériaux diélectriques (isolant électrique, comme le plastique ou le verre) à l'aide de leur indice de réfraction. Mais pour des matériaux non-diélectrique (comme des métaux), une différente méthode, plus complexe, doit être utilisée. Dans ce second cas, on préfèrera utiliser une base de données préexistante [9], compilant les indices de réfraction de nombreux matériaux.

PDF

Dans le cas d'un matériaux diffus ($\alpha > 0.2$), la PDF lambertienne est suffisante.

$$P = \frac{1}{2\pi} \qquad L_o(\mathbf{p}, \mathbf{w_o}) = \frac{\pi}{2}\sum_k^N L_i(\mathbf{p}, \mathbf{w_k})\left(\frac{DFG}{\mathbf{n} \cdot \mathbf{w_o}}\right)$$

Cependant, le modèle Cook-Torrance est capable de représenter des surfaces réfléchissantes. Ce cas n'est cependant pas géré par le moteur, et la PDF correspondante n'a pas été calculée.

Soft shadows

Les sources lumineuses du moteur sont uniquement ponctuelles. Cela signifie qu'elles peuvent être soit totalement cachées, soit totalement visibles, provoquant des ombres parfaites (hard shadows). Cependant, une ombre est douce lorsque la source lumineuse partiellement occulté. On ajoute alors un paramètre « rayon », leur donnant un volume, et donc une forme sphérique.

Lors du sampling de la source lumineuse, on choisit un point $\mathbf{s}$ au hasard sur la surface de la source, et on note $\mathbf{w_s}$ le vecteur de $\mathbf{p}$ vers $\mathbf{s}$. On applique le calcul d'illumination classique pour M échantillons vers la source. En tirant un nombre suffisant de rayons, on obtient une ombre douce.

$$L_o(\mathbf{p}, \mathbf{w_o}) = \frac{\pi}{2} \sum_{k}^{N} \left(\frac{1}{S} \sum_{s}^{M} L_s(\mathbf{p}, \mathbf{w_s}) \left(\frac{DF(\mathbf{w_s})G(\mathbf{w_s})}{\mathbf{n} \cdot \mathbf{w_o}} \right) \right)$$

Il est cependant incorrect d'échantillonner la sphère. En effet, on constate une plus forte concentration des points d'échantillonnage sur les bords que sur le centre car ils sont visuellement plus proches. Il serait plus juste de répartir les points d'échantillonnage sur un disque centré sur la source lumineuse et orientée vers $\mathbf{p}$.

Il serait aussi souhaitable de gérer la puissance de la lumière en fonction de sa taille. En effet, pour une puissance données, si sa taille augmente, sa radiance émise en une portion de surface ne change pas, augmentant de fait sa puissance réelle.

Path tracing et transport de lumière

Cette partie de l'algorithme est le cœur d'un *raytracer*. C'est aussi l'une des plus complexe. Jusqu'ici, notre moteur de rendu peut être considéré comme ne gérant que l'illumination locale, c'est-à-dire en ne prenant en compte que les informations locales de la surface intersectée. Un moteur gérant l'illumination globale permet à une surface, rouge par exemple, de réfléchir une partie de sa lumière émise sur une surface avoisinante.

L'ombre de cette sphère a une teinte rouge car la lumière rouge réfléchi par la sphère éclaire le sol.

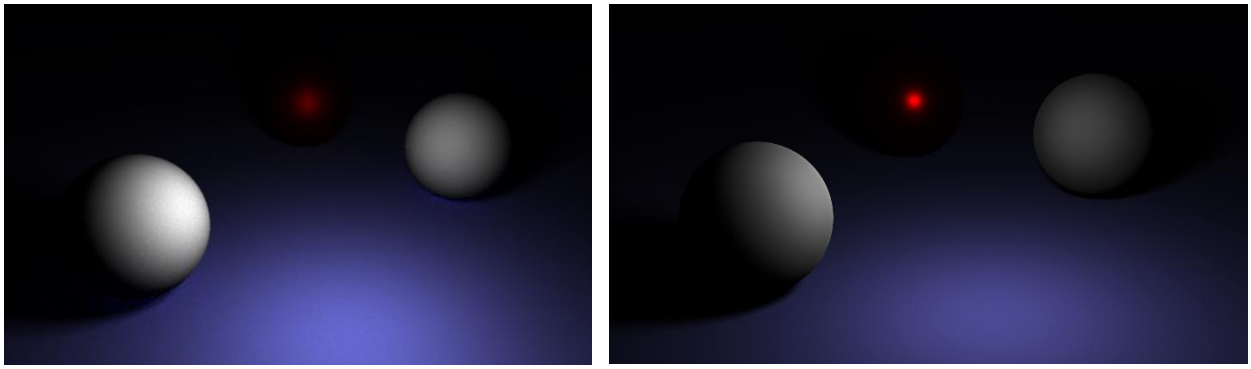
Malheureusement, cette étape n'a pu être menée à bien dans sa totalité. Sa complexité fait qu'elle n'a pu donner de résultats satisfaisant à temps. Cependant, une ébauche du transport de lumière est présente dans notre moteur. A chaque rebond, le résultat de l'évaluation de la BRDF est sauvegardé puis accumulé par multiplication aux radiances précédentes. Pour un chemin composé des sommets $\mathbf{p_0}, \mathbf{p_1}, \dots, \mathbf{p_n}$ où p_0 est le foyer de la caméra et $\mathbf{p_n}$ la dernière intersection, on obtient

$$L_o(\mathbf{p_1}, \mathbf{w_o}) + \prod_{i=2}^{n-1} L(\mathbf{p_i}) f_r(\mathbf{p_{i+1}} \rightarrow \mathbf{p_i} \rightarrow \mathbf{p_{i-1}}) \cos\theta_i$$

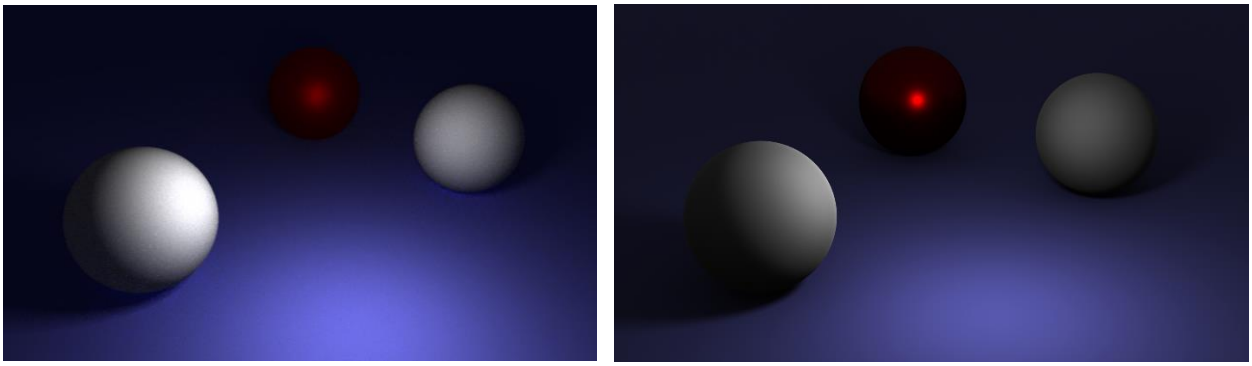
Performances

Comparaison avec un moteur de référence

Le rendu final de notre moteur a été comparé avec le moteur Cycles de Blender. Une scène de test identique à la nôtre a été créée dans le logiciel. Elles sont présentées ici sans rebond de lumière.



A gauche, notre moteur, à droite, Blender.



Ici avec une ambiant light.

Pour des paramètres identiques, on remarque des différences certaines entre les deux moteurs, notamment sur la sphère rouge, et l'intensité du dégradé aux angles extrêmes d'éclairage.

Images par secondes

Pour la scène présentée, application lancée sous Chrome 79 avec une carte graphique nVidia GTX 970

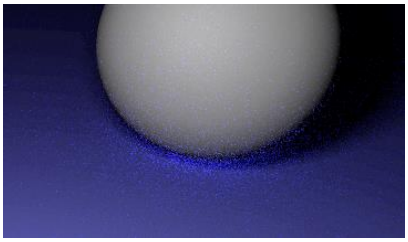
Rayons	Rebonds	Lumière	IPS
5	1	1	**>60**
10	1	1	**>60**

15	1	1	**>60**
20	1	1	**>60**
10	*5*	1	**>60**
15	*5*	1	**30**
20	*10*	1	**27**
10	5	*5*	**30**
15	5	*10*	**20**
15	5	*15*	**14**

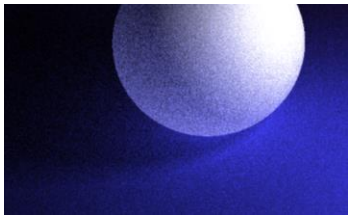
On remarque qu'augmenter le nombre de rayons d'échantillonnage de la lumière affecte grandement les performances. Notre architecture demande en effet une évaluation de la BRDF à chaque échantillonnage de la source lumineuse, ce qui multiplie le nombre de calculs. Il serait préférable d'essayer une optimisation ne demandant qu'une unique évaluation de la BRDF, pondérée par un facteur d'occultation de la lumière.

Résultats visuels

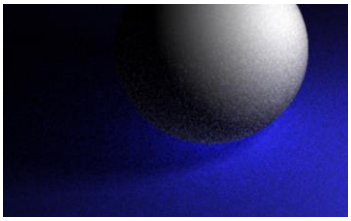
Transport de lumière



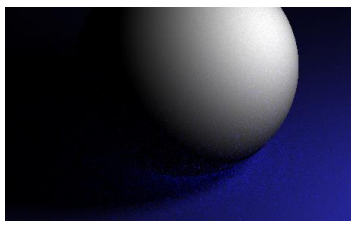
Le transport de lumière fonctionne en partie. Cependant, si l'augmentation du nombre de rayon augmente effectivement la qualité de l'image, augmenter le nombre de rebonds fais disparaitre l'éclairage indirecte. C'est un problème lié à une implémentation incorrecte de l'équation de transport de lumière qui ne termine pas correctement.



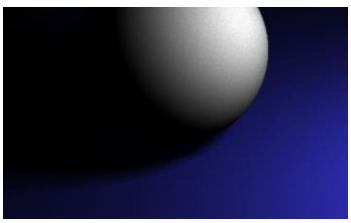
1 rebond



2 rebonds



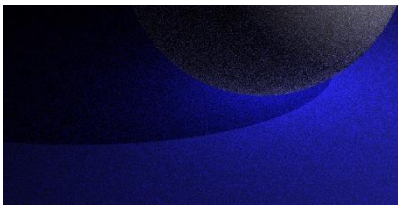
5 rebonds



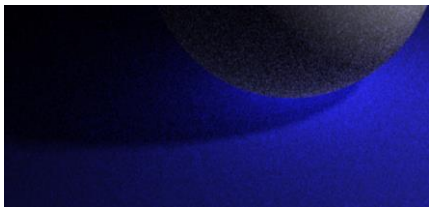
10 rebonds

Ombres

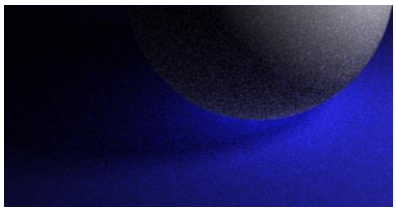
Ici, nous avons modulé la taille de la lumière présente dans la scène. A gauche, la lumière a une taille nulle, l'ombre est donc « parfaite ». Sur l'image au centre et à gauche, nous lui avons donné une taille. L'ombre est de bonne qualité.



Rayon : 0.0



Rayon : 0.2



Rayon : 0.5

Conclusion

Malgré certains choix d'implémentation, la base actuelle de notre programme est assez satisfaisante en l'état, et est facilement extensible. Le lancer de rayons primaires, l'évaluation de la BRDF à la première intersection ainsi que les ombres douces donnent des résultats satisfaisants pour des performances acceptables (>60 ips).

Néanmoins, utiliser une telle technologie sans optimisations est encore hors de portée des machines actuelles sans cartes graphiques puissantes. Pour une très simple scène ne contenant que 3 sphères, on passe en dessous de 60 *ips* dès les 50 intersections par pixels. Il est impératif de réduire le plus possible le nombre de rebonds nécessaires à la génération de l'image.

La création d'un moteur de rendu par lancer de rayon est une tâche longue et complexe. Déjà familier des moteurs de rendu, j'ai eu l'occasion d'étudier ce sujet plus avant. J'ai découvert de nombreux concepts et domaines dont j'ignorais la contribution à ces types de programmes, comme les systèmes d'échantillonnage ou la radiométrie. Je me suis également heurté à des difficultés que je n'ai pu surmonter à temps, comme la compréhension du transport de lumière. Si ce projet était très intéressant à réaliser, il faudrait encore beaucoup de temps et d'apprentissage de ma part pour qu'il soit aboutis, du moins pourvu de toutes les fonctionnalités basiques. Ce projet sera très surement continué dans le cadre personnel.

Références

[1] A. Gerdelan, «Mouse Picking with Ray Casting,» 2 Octobre 2016. [En ligne]. Available: http://antongerdelan.net/opengl/raycasting.html. [Accès le Décembre 2019].

[2] Scratchpixel, [En ligne]. Available: https://www.scratchapixel.com/index.php?redirect. [Accès le Décembre 2019].

[3] M. Pharr, W. Jakob et G. Humphreys, Physically Based Rendering:From Theory To Implementation.

[4] R. L. Cool et K. E. Torrance, «A Reflectance Model for Computer Graphics,» *ACM Transaction on Graphics,* vol. 1, n° %11, pp. 7-24, 1982.

[5] J. d. Vries, «Learn OpenGL,» [En ligne]. Available: https://learnopengl.com/.

[6] B. Karis, «Specular BRDF Reference,» 3 Août 2013. [En ligne]. Available: http://graphicrants.blogspot.com/2013/08/specular-brdf-reference.html. [Accès le Janvier 2020].

[7] B. Walter, S. R. Marschner, H. Li et K. E. Torrance, «Microfacet Models for Refraction through Rough Surfaces,» *Eurographics Symposium on Rendering,* 2007.

[8] C. Schlick, «An Inexpensive BRDF Model for Physically-Based Rendring,» *Proc. Eurographics,* vol. 13, n° %13, pp. 233-246, 1994.

[9] M. Polyanskiy, «RefractiveIndex,» 2020. [En ligne]. Available: https://refractiveindex.info/.