Screen Space Directional Occlusion

INF585 – Projet de Computer Animation

Cédric JAVAULT – Master AI Year 1

Introduction

Le cours de INF584 a pour ambition de nous faire découvrir les briques essentielles de la **synthèse d'image**. Le terme anglais de « **Rendering** » exprime probablement mieux l'enjeu : passer d'une scène virtuelle composée notamment de sources de lumière, de meshs 3D (avec topologie, géométrie, texture, et propriétés exprimant le comportement par rapport à la lumière), et d'une caméra (position, direction, distance focale et taille de l'écran) … à une image 2D faite de w*h pixels.

Les **applications et les enjeux sont multiples**. Les attentes sont bien différentes que ce soit dans les bureaux d'études pour l'industrie (outils de CAO de plus en plus perfectionnés), chez les architectes, chez les concepteurs de jeux vidéo et, bien sûr, dans la production cinématographique, laquelle tire aujourd'hui incontestablement la recherche, et où les images de synthèse ont depuis fort longtemps dépassé le seul cadre des dessins animés.

D'un point de vue physico-mathématique, calculer une image de synthèse, c'est résoudre **l'équation du rendu,** proposée simultanément par by David Immel et al. and James Kajiya en 1986, et qui exprime la manière dont la lumière se propage dans l'espace…

Intensité renvoyée *Emission intrinsèque* *Lumière Reçu* *Réflectance*

$$L_o(\omega_o) = L_e(\omega_o) + \int_0^{2\pi} \int_0^{\pi/2} L_i(\omega_i) f(\omega_i, \omega_o) \cos\theta_i d\theta_i d\phi_i$$

… mais cette équation est for compliquée à résoudre ! Elle exprime L en tout point de l'espace et dans chaque direction ce qui fait cinq degrés de libertés. Pour être puriste, il faudrait de plus l'intégrer longueur d'onde par longueur d'onde, et prendre le temps en considération si les objets sont en mouvement !

Sauf peut-être dans des cas extrêmement, l'équation du rendu n'a bien sûr pas de solution explicite. Nous ne savons que calculer des **approximations :**

- **plus ou moins exactes**
- **plus ou moins réalistes**
- **et plus ou moins rapidement !**

Trouver le meilleur compromis entre ces paramètres a occupé, et occupe encore une part importante de la recherche opérationnelle en rendering.

J'ai travaillé sur le papier « *Approximating Dynamic Global Illumination in Image Space* »[1] qui a posé les bases de la SSDO « Space Sceen Directionnal Occlusion » en 2007 qui propose d'**améliorer la technique de SSAO** « Screen Space Ambiant Occlusion » : au prix d'un temps de calcul légèrement supplémentaire, la SSDO approxime la direction de l'occlusion et peut même simuler un premier rebond de lumière pour amener, par exemple, un effet de color bleeding, qui est typique de la Global Illumination « GI », laquelle est beaucoup plus coûteuse en temps de calcul.

On le voit d'emblée avec le paragraphe précédent, pour bien comprendre la SSDO et ses apports, il faut commencer par présenter d'autres techniques. J'ai donc structuré ce rapport en trois parties : dans un premier temps, je présente la théorie et les résultats de mon implémentation pour la GI, l'AO et la SSAO. Dans un second temps, je présente la SSDO et mes résultats. Enfin, dans une troisième partie, je donne quelques détails d'implémentation, et discute les résultats obtenus avant d'esquisser les développements possibles pour enrichir mon code en conclusion.

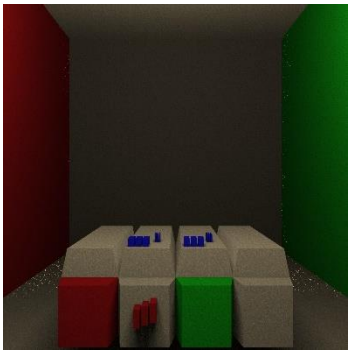
[1] [RGS09] RITSCHELT., GROSCHT., SEIDELH. H.-P.: Approximating Dynamic Global Illumination in Image Space. In Proceedings of the 2009 symposium on Interactive 3D graphics and games -I3D '09(2009), pp. 75–82.

I. « Previous Work »

1°/ Construction d'une scène et Direct Illumination

En première année de Master IA à l'X, j'ai suivi cette année 8 cours d'informatique / maths appliquées dont 5 comprenaient des développements significatifs. Au contraire de tous les autres enseignements, vous nous faites partir sans aucun code de départ. J'ai trouvé cette méthodologie très intéressante et tout à fait complémentaire des autres enseignements ; elle m'aura énormément fait progresser[2].

Pour ce projet, et par cohérence, j'ai donc choisi de générer entièrement la scène à afficher de manière procédurale[3]. Pour mon travail de TD, j'avais construit une scène très proche de l'image ci-dessous à gauche (qui est elle-même tirée de vos slides). J'ai ajouté quelques éléments de petits détails inspirés de l'image du milieu (qui vient de l'article) et ainsi **fabriqué la scène à droite**.



Cette image à droite correspond à la scène que j'ai générée, calculée avec une **Direct Illumination** (« DI ») et dans laquelle j'ai positionné deux sources de lumière type AeraLight :

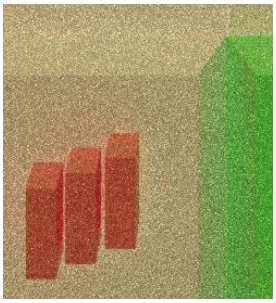
- Une assez petite, en haut de l'image (un peu comme ci-dessus à gauche), même si elle n'est pas visible dans mon image à gauche
- Une grande source en avant de l'image pour que la GI et la SSDO captent plus de lumière.

A noter : j'ai passé un temps certain à « régler » l'éclairage pour afficher des images aussi proches que possible dans les différentes routines (DI GI AO SSAO et SSDO), sans y parvenir de manière vraiment satisfaisante : j'aurai pu passer plus de temps, mais ce n'était pas l'enjeu essentiel. Au moins fut-ce l'occasion de bien comprendre le rôle central joué par la lumière en Rendering…

2°/ Résultats en GI

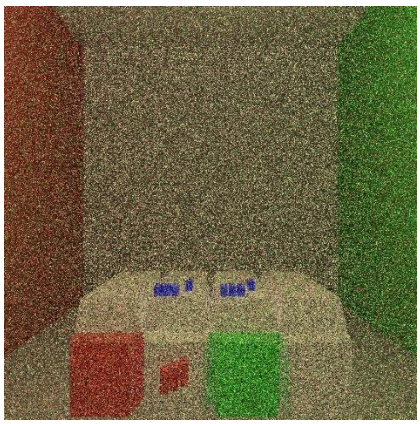
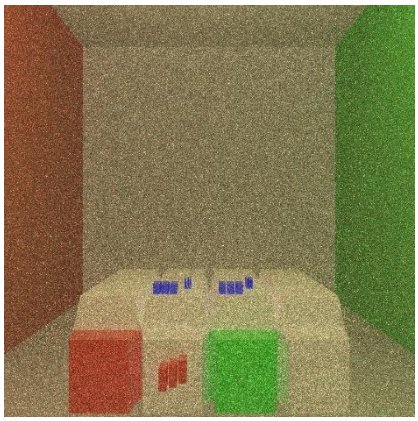
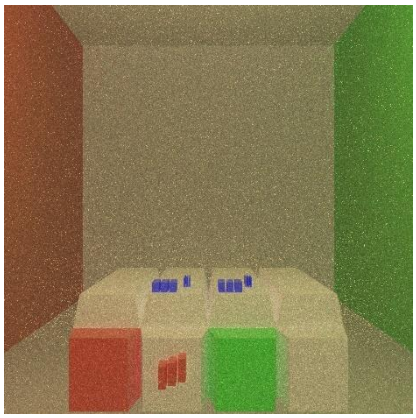
En TD, j'avais programmé une routine de RayTracing en illumination globale, routine que j'ai finalisée dans le cadre du présent projet pour avoir une image à laquelle comparer les différentes approximations (AO, SSAO, SSDO) qui j'allais produire.

J'ai eu des difficultés pour obtenir un éclairage et un contraste satisfaisant. Le résultat, même avec 256 samples reste assez bruité. Si mon travail en GI est probablement très perfectible, il a au moins le mérite de bien faire apparaître l'effet de color bleeding comme sur ces deux détails de l'image à 256 samples de la page suivante.



[2] Je partais de loin : je croyais avoir compris la théorie pour passer de C à C++ mais n'avais jamais créé une classe et ses fonctions membres ; je n'étais pas au clair avec les passages par référence ; j'ai pu testé step by step de petites améliorations pour gagner à chaque fois quelques pourcents en vitesse d'exécution.

[3] Je ne me sers tout de même d'un cube unitaire « cube_tri.off » dont je charge la topologie pour éviter une erreur bête.

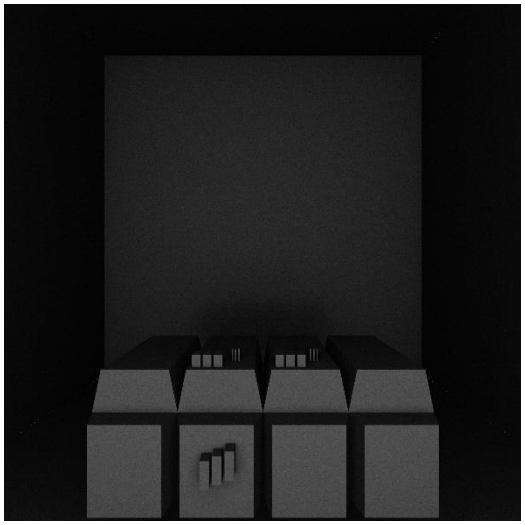




Ci-dessus, le résultat avec 1, 16 puis 256 samples en GI.

3°/ Principe et résultats en AO

Le calcul complet en GI est long et fortement bruité. **L'AO offre une alternative en calculant assez rapidement une approximation** qui, combinée à la DI, donne une vision assez réaliste des ombres portées. Pour chaque de pixel de l'image[4], l'algorithme calcule tout d'abord le point de la scène 3D auquel il correspond (quel mesh ? quelle primitive ? quelle position barycentrique sur cette primitive ?) puis il relance des rayons depuis ce point, dans le demi hémisphère orienté par la normale en ce point… et calcule la proportion de rayons qui sont occultés par la scène.

Plus cette proportion est forte, plus ce point est « enfermé » et sera sombre. L'image finale correspond à **une combinaison entre l'AO et la DI**. En théorie, il faudrait faire AOxDI mais, compte tenu des paramètres que j'avais et de l'éclairage présent, j'ai eu des résultats plus réalistes avec AO/2+DI :



Calcul de l'AO (clair = peu occulté)

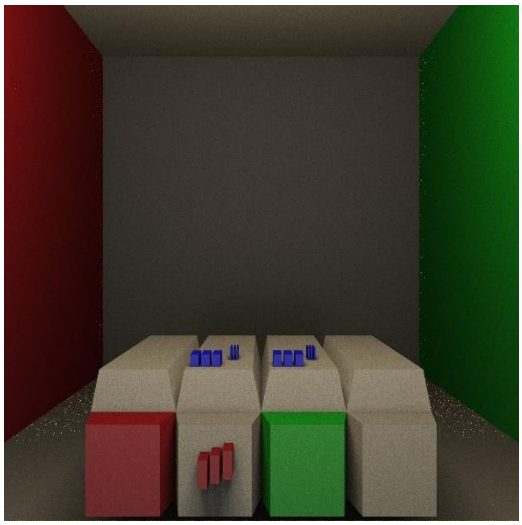


Image finale = DI + AO/2

4°/ Principe et résultats en SSAO

L'AO donne des résultats qui sont graphiquement très intéressants. Le problème, c'est que le temps de calcul peut être significatif :

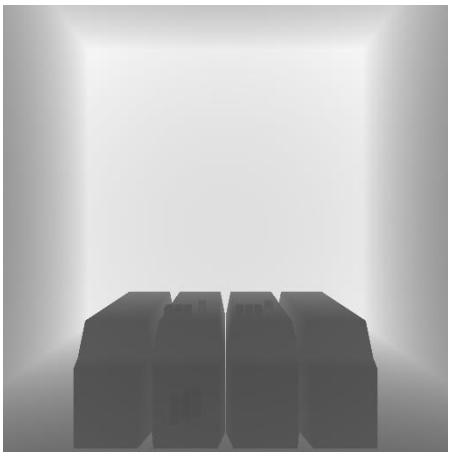
- Pour chaque pixel, il faut non seulement trouver le point d'intersection depuis la caméra
- Mais il faut ensuite et surtout relancer de nombreux rayons (pour limiter le bruit), et calculer à chaque fois une intersection pour ce nouveau rayon.

La SSAO propose l'approximation suivante de l'AO : plutôt que de travailler en 3D dans la scène elle-même, nous allons travailler dans le Z-Buffer qui est l'image 2D (« Screen Space ») des distances entre la caméra et

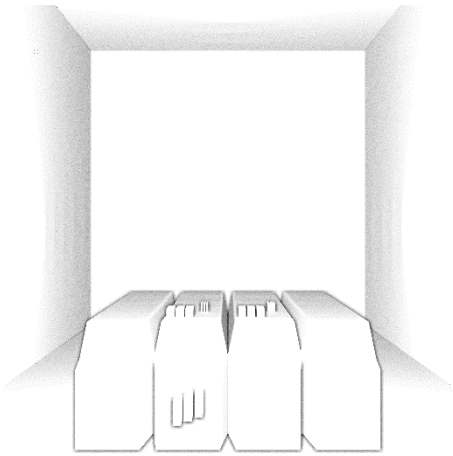
[4] On considère pour simplifier que tous les objets sont opaques : pas d'effet de transparence.

la scène pour chaque pixel. A noter : Z-Buffer est déjà calculé dans le processus de rastérisation pour la DI. Ainsi, on ne relance plus des rayons en 3D mais on compare les distances dans le Z-Buffer pour le pixel actuel et pour un échantillon de ses voisins[5].

Cette approximation, qui peut sembler assez brutale, donne des résultats graphiques très intéressants ! Comme pour l'AO, l'image finale est une combinaison entre SSAO et DI. Après quelques tests, j'ai choisi la formule (SSAO + 0.4) * DI, et obtenu les images suivantes :



Z-Buffer (sombre =proche)



SSDO (sombre = fortement occulté)

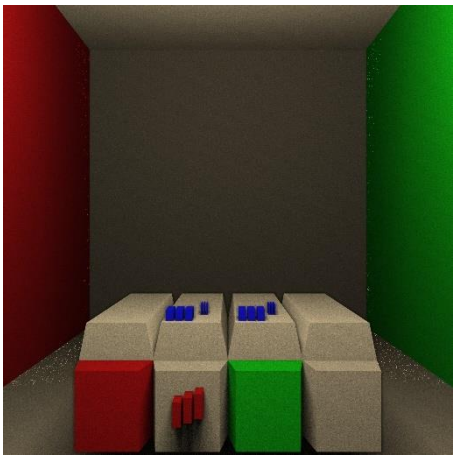


image finale : (SSAO + 0.4) * DI

II. Screen Space Directionnal Occlusion

1°/ Les idées de base dans l'article

L'article propose essentiellement une amélioration de la SSAO.

On va toujours se servir du Z-Buffer… en remarquant que son calcul permet d'avoir **plus d'information** que la seule distance à la caméra : dans le processus de rastérisation, on trouve aussi pour chaque pixel les coordonnées 3D qui correspondent dans la scène et la normale en ce point.

Comme dans la SSAO, pour chaque pixel PIX0, on trouve le point P0 de la scène 3D grâce au Z-Buffer (calculé une fois pour toutes). Mais plutôt que de regarder N pixels autour en 2D :

- On prend N points (P_1, …P_N) **en 3D** dans le demi hémisphère centré en P0, orienté par la normale à P0, dans un rayon R_{max} prédéfini.
- Pour chaque point P_i, on calcule le pixel PIX_i correspond dans l'image et on compare :
  - la distance d1 entre la caméra et le point 3D correspondant à PIX_i dans la scène (en s'aidant à nouveau du Z-Buffer)
  - et la distance d2 entre la caméra et P_i
- Si d1>d2, le point P_i est devant la scène et on considère que la lumière arrive depuis cette direction sur le pixel PIX0.
- Si d1<d2, on considère dans un premier temps qu'il n'y a pas de lumière depuis cette direction. Dans un second temps, une fois l'ensemble du calcul fait, on peut faire une seconde passe et transmettre de la lumière de PIX_i à PIX0 pour simuler un rebond lumineux.

L'article présente plusieurs extensions que je n'ai pas creusé.

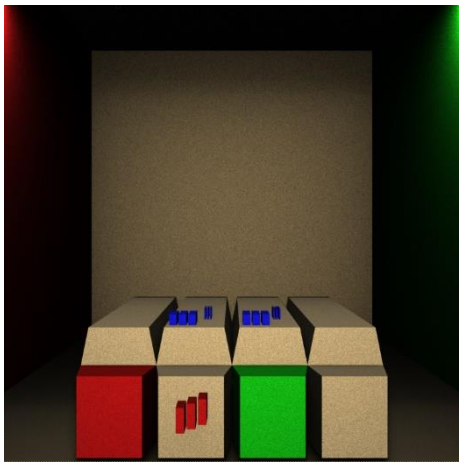
[5] Pour éviter qu'un mur plat comme cela du fond ne soit beaucoup plus lumineux dans la direction exacte de la caméra, j'ai mis un offset et considéré qu'il n'y avait pas occlusion par un pixel voisin si ZBuffer(voisin) > ZBuffer(point_testé) - 0.01.

2°/ Implémentation et résultats

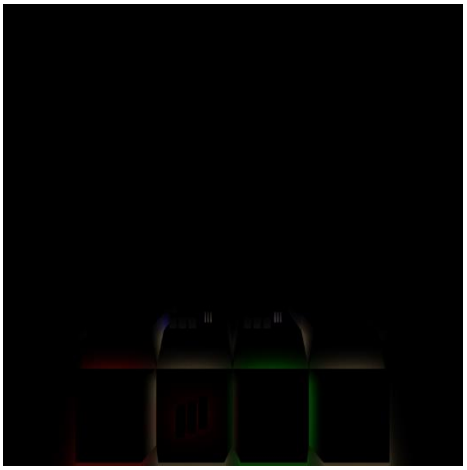
J'ai créé une classe Z-Buffer et une instance du constructeur pour calculer une fois pour toutes les informations nécessaires pour la suite (pour chaque pixel : positions 3D, normale, distance à la caméra…). La routine principale prend pour paramètres l'image de sortie, la scène, la BVH de la scène[6], le nombre de rayons à tirer et un booléen pour décider si on veut un rebond ou pas :

```
void compute_SSDO_ZBuff(Image& im, scene& scene, BVH& mon_BVH, int KMax, bool bounce)
```

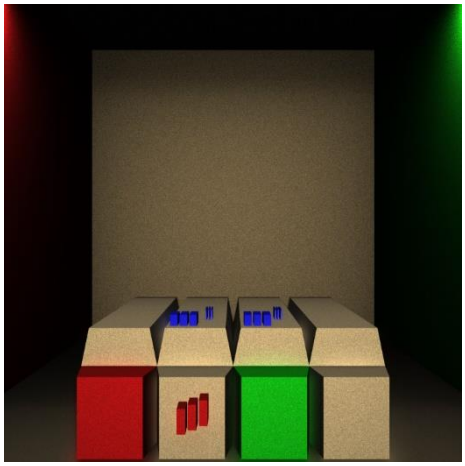
Je n'ai pas réussi à ce que les murs de droite et de gauche aient un éclairage comparable à celui des images précédentes… et n'ai pas particulièrement insisté car le sujet principal me semblait être le centre de la scène, qui lui, me semble assez réussi. Voici les images obtenues :



SSDO sans rebond

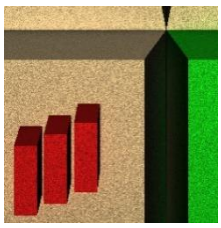


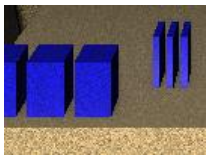
lumière supplémentaire du seul rebond



somme des deux

Voici de plus des détails correspondant à ceux présentés plus haut en global illumination. L'image ci-dessous à gauche fait bien apparaître un color bleeding rouge autour des méso-structures. Plus à droite, toujours sur la même image, le vert se peint sur le blanc et réciproquement. C'est nettement moins le cas pour l'image de droite où seuls les trois parallélépipède rectangle bleus fins à droite ont « blanchi » sur la face de devant (ils sont aussi un peu bleui du fait des structures en avant d'elles), mais aucun des six blocs bleus ne « bleede du bleu » sur le blanc... Je pense que c'est une conséquence de la géométrie particulière de cet endroit de la scène où tous les angles sont droits, presque perpendiculairement à la caméra : le cos1 x cos2 de la formule tend probablement très vite vers 0 dans ces conditions.





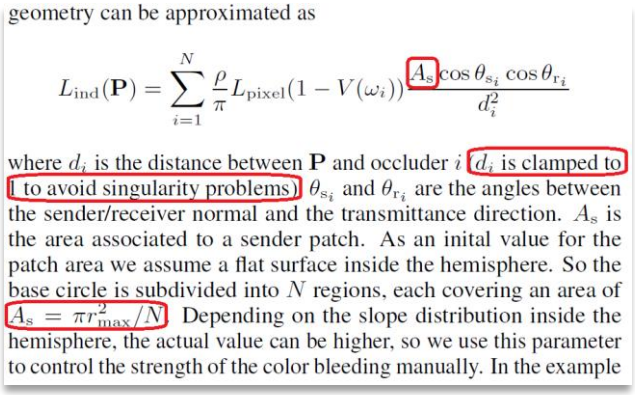
[6] J'ai implémenté une BVH comme structure d'accélération. En GI, les résultats sont impressionnants sur des scènes complexes (multiplication de la vitesse d'exécution par presque 100 pour afficher le visage utilisé en TD) mais la scène utilisée est très simple et ne bénéficie probablement pas énormément de cette amélioration. De plus, SSAO et SSDO travaillent dans l'espace de l'image et pas en 3D. Seul le calcul du Z-Buffer bénéficie alors de la BVH.

III. Discussion

1°/ Les difficultés de calibration

Les auteurs vont assez vite sur le sujet dans l'article mais la SSDO présente plusieurs paramètres à ajuster « à la main », tout changement affectant le rendu final :

- R_{max} : le rayon dans lequel les rayons sont tirés
- La distance minimale d_i qui est au moins de 1 (dans quelles unités ?)
- A_s ("*We used this parameter to control the strength of the color bleeding manually*").



geometry can be approximated as

$$L_{\text{ind}}(\mathbf{P}) = \sum_{i=1}^{N} \frac{\rho}{\pi} L_{\text{pixel}}(1 - V(\omega_i)) \frac{A_s \cos\theta_{s_i} \cos\theta_{r_i}}{d_i^2}$$

where d_i is the distance between $\mathbf{P}$ and occluder i (d_i is clamped to 1 to avoid singularity problems), θ_{s_i} and θ_{r_i} are the angles between the sender/receiver normal and the transmittance direction. A_s is the area associated to a sender patch. As an inital value for the patch area we assume a flat surface inside the hemisphere. So the base circle is subdivided into N regions, each covering an area of $A_s = \pi r_{\max}^2 / N$. Depending on the slope distribution inside the hemisphere, the actual value can be higher, so we use this parameter to control the strength of the color bleeding manually. In the example

De la même manière, en utilisant l'AO et la SSAO, je n'ai eu d'autre option que de calibrer manuellement pour aboutir à des formules empiriques qui donnaient des images visuellement satisfaisantes (DI + AO/2) et (SSAO+0.4)xDI… Et je ne parle même pas de la lumière !

J'imagine qu'en pratique, ce n'est pas un problème si compliqué ; il faut juste revoir la calibration de tous les paramètres importants en fonction du but poursuivi … et de la scène à rendre. Mais cela prend du temps et nécessite de bien maîtriser le code.

2°/ Quelques détails d'implémentation

Avant de comparer les temps d'exécution des différentes routines, il faut préciser quelques points pour bien comparer des choses comparables :

- La scène sur laquelle je travaille est très simple avec une petite centaine de primitives. Une scène plus complexe :
  - N'impacterait SSAO et SSDO que pour le calcul du Z-Buffer, ce qui est une faible part du temps total de calcul
  - Mais impacterait beaucoup plus DI, GI et AO qui travaillent dans l'espace 3D et non dans l'espace de l'image.
  - Je n'ai pas fait de tests faute de temps mais imaginons qu'au lieu de travailler sur quelques centaines de primitives (scène actuelle), on travaille sur 1000 fois plus. Comme $\log_2(1000)$ vaut environ $\log_2(1024) = 10$, on peut a minima compter sur une multiplication par 10 des temps d'exécution de DI, GI et AO.
- J'ai travaillé pour optimiser mon code autant que possible. Toutefois, les points suivants n'ont pas été optimisés et viennent perturber la comparaison des vitesses d'exécution
  - DI : pour calculer les effets d'ombrage liés aux Aera Light, je lance 256 rayons. On peut lancer beaucoup moins de rayons, et même un seul si on utilise des Point Light.
  - AO : je lance également 256 rayons, cette fois pour baisser le bruit. Il y a certainement des solutions pour débruiter avec moins de rayons !
  - SSAO : aucune optimisation entre DI et SSAO. Ex : le Z-Buffer est calculé deux fois.
  - SSAO et SSDO : je lance de nombreux rayons pour baisser le bruit (64 en SSAO et 1024 en SSDO). Je vais normaliser ces paramètres pour mieux comparer les techniques ; par ailleurs on peut garder une bonne qualité et peu de bruit en optimisant le lancer de rayon en cherchant d'emblée les bonnes directions. Je n'ai pas creusé cet aspect qui est manifestement un **axe central de l'amélioration de la performance**.
- Rien à voir : compte tenu de la géométrie de ma scène (avec des cubes ou des quasi cubes aux arrêtes saillantes), j'ai préféré prendre les normales à chaque face constante sur toute la face plutôt que d'utiliser les normales de Phong.

3°/ Les vitesses d'exécution

Voici mes résultats en millisecondes sur une petite image (150x150) et un petit PC portable :

	Nb Ray	Time (ms)	Nb Ray normalisé	Time normalisé
Direct Illumination	256	23 435	1	92
GI 16 samples	**16**	**6 217**	**16**	**6 217**
AO	256	6 039	16	378
AO + DI			**64 AO + 1 DI**	**470**
Calcul du Z-Buffer	SO	39	SO	39
SSAO sans Z-Buffer	64	65	16	16
SSAO (yc Z-Buffer)	64	104	16	55
SSAO (yc Zbuffer) + DI			**16 SSAO + 1DI**	**147**
SSDO + avec 1 Bounce	**1 024**	**5 820**	**16**	**91**

Les deux premières colonnes correspondent **aux résultats réellement obtenus**. Dans les deux colonnes suivantes, **j'ai essayé de normaliser** en considérant – ce serait à discuter !! – qu'on peut avoir une image correcte avec 1 seul rayon en DI (cas du point light), et 16 rayons en AO/ SSAO/SSDO (ce qui nécessite à l'évidence de bien les choisir). J'ai gardé ma scène simple et – comme discuté à la page précédente – une scène complexe pénaliserait principalement GI et AO.

Ces résultats ne prétendent pas à une cohérence parfaite. Ils sont plutôt illustratifs et indiquent clairement les points suivants :

- La technique la plus proche de la physique, la GI, est très couteuse en temps d'exécution puisqu'on est à 6 217 ms, très au-delà du reste
- L'AO donne des résultats graphiquement intéressants en allant 16 fois plus vite.
- SSAO va encore 3 à 4 fois plus vite que l'AO (3 fois chez moi – probablement 4 fois en optimisant mieux le code)
- Enfin, mon implémentation de la SSDO est plus rapide que la SSAO même avec un rebond. Ce n'est pas normal et cela traduit certainement le fait que j'ai dû oublier une optimisation dans la SSAO… et / ou que j'ai concentré mon énergie sur l'efficacité de ma routine SSAO.

Conclusion : pour aller plus loin

L'intérêt fondamental de la SSDO est de calculer des approximations très réalistes et un temps très court, ce qui permet de faire du rendu en temps réel de scènes complexes. J'aurai bien voulu poursuivre… mais comme mon stage a démarré en parallèle lundi 23 mars, que qu'une autre matière (INF562) a transformé le contrôle papier annulé un mini projet… il faut être raisonnable et arrêter là.

Si j'avais eu plus de temps, j'aurais souhaité travailler dans quatre directions pour mieux utiliser mon implémentation :

- **Tester différents effets de lumière**. La SSDO permet en effet de tenir compte de la couleur de la lumière émise pour différentier, par exemple, une ombre portée par une lumière rouge de celle portée par une lumière verte. Dans mon code actuel, toutes les lumières sont blanches (plus précisément, shade2 utilisée par toutes les routines de RayTracing sauf DI prend en compte l'intensité mais pas la couleur de la lumière émise, qu'il tient pour blanche).
- **Rendre des scènes complexes** avec des centaines de millier de primitives.
- **Réduire intelligemment le nombre de rayons** comme évoqué plus haut à plusieurs reprises pour améliorer l'efficacité computationnelle de la SSDO
- **Surtout, passer en RealTime Raytracing**, l'utilisateur ayant par exemple la possibilité de bouger lui-même à son gré dans la scène. J'aurais ainsi pu démontrer – au-delà des simples excellentes vitesses d'exécution - que le procédé fonctionne effectivement en temps réel !