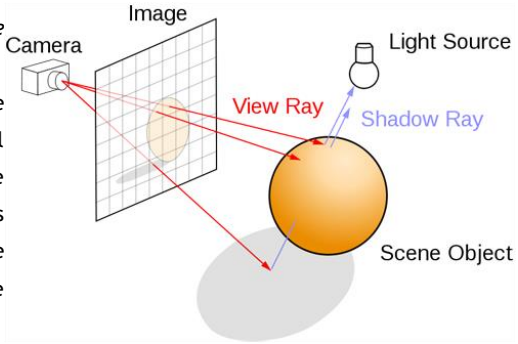
INFSI 350 - Informatique Graphique 3D et Réalité Virtuelle - Projet

Moteur de Rendu en Lancer de Rayon



L'objectif de ce projet est d'implémenter un moteur de rendu en *lancer de rayon*.

L'archive du projet contient un programme nommé *raymini* chargeant une scène 3D et l'affichant en OpenGL dans le fenêtre de gauche. L'essentiel du travail consistera à remplir l'image de droite à l'aide d'un algorithme de synthèse d'image par lancer de rayon. On pourra se référer à Blender pour tester les différents effets et comparer l'implémentation produite dans *raymini*. D'une manière générale, on considérera le problème avec une unique source de lumière avant d'en ajouter de nouvelles.

Préambule

Après avoir installé Qt, GLEW et libQGLViewer, observez le code de *raymini*, celui-ci contient déjà tout un environnement pour le lancer de rayon, sans toutefois implémenter l'algorithme complet. Le module *Scene* contient lumières et objets, les objets contenant géométrie (maillage) et apparence (matériaux). Notez que par simplicité, la scène ne contient pas de caméra : on se référera à la caméra gérée par libQGLViewer et disponible via la classe fille GLViewer.

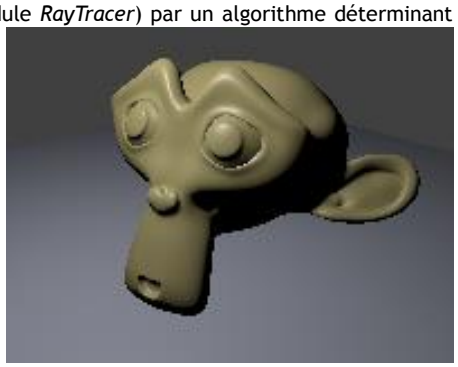
Dans sa version actuelle, *raymini* se contente de tracer des rayons depuis le point de vue, pour chaque pixel, et de s'arrêter à la boite englobante de la scène. Cette scène est initialisée par défaut avec un objet et un plan. On pourra composer d'autres scènes à l'aide de Blender par exemple. On autorise tout ajout de classes pour architecturer le code proprement (notamment une structuration de scène plus complexe, avec une matrice de transformation par objet, etc). Le point d'entrée du projet est la méthode *render* de la classe *RayTracer*. Dans le cadre de ce projet, il est demandé de créer au moins **2 scènes originales** par groupe, mettant en valeur le travail effectué. Pour se faire, on pourra chercher des modèles OFF sur internet et les assembler en une scène dans le programme ou bien écrire un chargeur de scènes OBJ. Les modèles présents dans les archives de TP peuvent également être utilisés.

Notez que l'on pourra modifier l'interface graphique pour ajouter des paramètres propres à chaque question. Doxygen peut également aider à rapidement comprendre la base de code. Une version instable pour GLUT est aussi fournie (fortement déconseillée).

Remise du projet : le projet est à effectuer par groupe de 3 à 4 élèves. Une présentation de 5 minutes par groupe sera faîte pendant le cours (voir l'emploi du temps, un template powerpoint est fourni dans l'archive). Les sources du projet, éventuellement un exécutable (si implémenté sous windows), un mini-rapport (2 à 3 pages, indiquant notamment les éléments techniques développés, les ajouts et d'une manière générale mettant en valeur le travail de façon illustrée) et 5 images de résultats seront à remettre par email (sujet de l'email : « INFSI350 <nom1> <nom2> <nom3> <nom4>), au plus tard la lundi suivant la présentation.

Partie 1 - Lancer de rayons

1.a « *Intersection rayon-Triangle* » Remplacer le code de la méthode *render* (module *RayTracer*) par un algorithme déterminant l'intersection **pi** du rayon **ri** avec la géométrie de la scène et affectant aux pixels une couleur relative à la position 3D de l'intersection (exemple : rgb(x,y) := xyz(**pi**)%255). La géométrie étant définie par un maillage triangulaire, on implémentera le test d'intersection rayon-triangle. On notera qu'une grande partie du code de *render* peut être conservée et que l'essentiel du travail ici consiste à ajouter une méthode de calcul d'intersection rayon-triangle à la classe *Ray* (voir le cours pour un pseudo-code).



1.b « *BRDF* » Ce test doit, en plus de déterminer s'il y a intersection entre le rayon ***ri*** et un triangle **tj**, fournir la position de l'intersection **pi** ainsi que ses coordonnées barycentriques dans **tj**. Celles-ci permettent d'interpoler le vecteur normal à la surface en ***pi*** à partir des vecteurs normaux stockés aux sommets de **tj**. On utilisera la BRDF de Phong pour calculer la réflectance en **pi** (propriétés stockées dans l'objet *Material* de la classe *Object*).

1.c « *KD-Tree* » Pour l'instant, il faut itérer sur l'ensemble des triangles pour chaque rayon (complexité linéaire). Ramener l'algorithme à une complexité logarithmique en employant un *Kd-Tree*.

Partie 2 – Effets avancés

L'image obtenue dans la partie 1 est en tout point semblable à un rendu par rasterization avec éclairage par pixel (classiquement implémentée via un fragment shader en OpenGL/GLSL). On souhaite maintenant profiter des avantages du lancer de rayon pour ajouter plusieurs effets et se déplacer de la qualité d'image type « jeux vidéo » à la qualité type « animation/effets spéciaux ».

2.a « *Antialiasing* » On souhaite améliorer la qualité de l'image en éliminant l'effet de crénelage (*aliasing*). Pour cela, lancer plusieurs rayons par pixel, en choisissant une distribution de rayons à l'intérieur du pixel et en faisant la moyenne des réponses couleurs obtenues pour le remplir. On pourra commencer par une distribution uniforme (2x2 ou 3x3 rayons par pixels, régulièrement distribués) avant d'expérimenter des distributions non alignées sur les axes (optionnel : 5 rayons sur un pentagone dans le pixel) ou stochastique à k échantillons.


2.b « Ombres » Rajouter les ombres à votre rendu en générant un rayon d'ombre à chaque intersection rayon de vue/géométrie, en direction d'une ou de plusieurs sources lumineuses. L'ombrage ne sera calculé que si le rayon surface/source n'intersecte aucune géométrie. On pensera également à ne considérer que les intersections epsilon-distantes de l'origine du rayon afin d'éviter le problème d'auto-occultation (précision numérique).

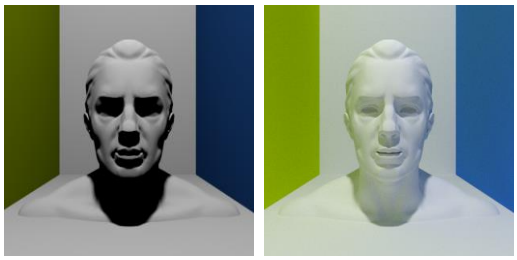

2.c « *Area Lighting* » Définissez une source de lumière comme « étendue » : en lieu et place de la source ponctuelle, définissez une source ayant la surface d'un disque de rayon variable (paramètre dans l'interface graphique). Les sources étendues (*area lights*) provoquent des zones de pénombre (ombre « douces »). Pour les évaluer, la notion de visibilité point-lumière v n'est plus binaire (ombres dures de la question 2.a) mais scalaire sur l'intervalle [0,1] avec **v**=0 équivalent à la totalité de la source occultée, et **v**=1 à la totalité de la source visible au point. Pour évaluer cette valeur d'ombrage, on émettra k rayons depuis le point de surface en direction de points échantillonnés au hasard sur le disque de la source étendue. La proportion de rayons atteignant la source sans intersection définit v. Si v>0, on multipliera l'ombrage calculé en *pi* par v pour remplir le pixel en question. Tester également avec l'évaluation de l'équation du rendu pour chaque échantillon (physiquement plus juste).


2.d Jusqu'à présent, l'éclairage par défaut était considéré comme nul. Idéalement, l'éclairage direct calculé aux questions précédentes devrait être complété par l'éclairage indirect pour une estimation physiquement (plus) juste de la radiance en chaque point. Mais ce processus est long. A la place, on se propose de l'approximer à l'aide de l'occultation ambiante (cf cours et tp sur l'*ambient occlusion*). Implémenter l'*ambient occlusion* dans le moteur en émettant k rayons, à partir de chaque intersection primaire, distribués sur l'hémisphère aligné sur la normale et en calculant la proportion d'intersections trouvées avec ces rayons dans une sphère de centre **p** et de rayon **r** (avec **r**=5% de la taille de la scène par exemple).


2.e Le lancer de rayon permet d'intégrer très facilement des effets de type « miroir ». Ajouter ce type d'effet avec les paramètres de contrôle nécessaire. Voir blender pour un exemple des effets obtenus. Comment obtenir des surfaces glossy ?


Partie 3 – Rendu basé-physique

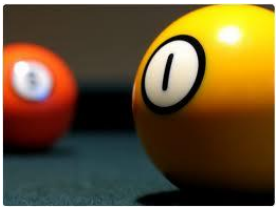
Implémenter un système d'éclairage global basé sur la *path tracing*. Pour rappel, le principe du path tracing est une généralisation du *ray tracing* : une fois une intersection **x** trouvée pour un rayon primaire, on ne s'arrête pas, et on tire un rayon selon une distribution donnée à partir de x, en cherchant de nouveau une intersection et ainsi de suite jusqu'à une profondeur maximale donnée. On remonte ainsi un « chemin de lumière », en modulant le transport d'énergie d'un sommet au précédent du chemin à l'aide de la BRDF au point. Dans un premier temps, on se donnera une distribution uniforme pour choisir aléatoirement sur l'hémisphère la direction selon laquelle tracer la suite du chemin à chaque intersection. Dans un second temps, on prendra en compte une distribution fonction de l'angle formé avec la normale au point d'origine de chaque segment du chemin (distribution en cosinus). On n'oubliera pas à chaque étape de sommer la contribution de l'éclairage directe (depuis les sources primaires) modulé par la BRDF également. Voir les références suivantes pour plus de détails, notamment la première :



- page wikipedia : http://en.wikipedia.org/wiki/Path_tracing
- Cours SIGGRAPH : http://geometry.caltech.edu/~keenan/mcrt-sg03c.pdf
- Une excellente web app en WebGL sur le path tracing : http://madebyevan.com/webgl-path-tracing/

Bonus « effets optiques »

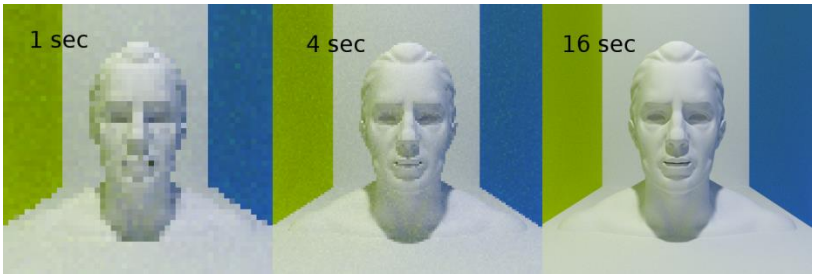
a. « ***Flou de mouvement*** » Définir un objet mobile dans la scène et implémenter un effet de flou de mouvement. Pour cela, on distribuera plusieurs rayons par pixel (comme pour l'antialiasing) mais qui inspecteront la géométrie de la scène sur une fenêtre de temps avant le temps courant (exemple : de -5 frames à la frame courante). Un rayon vivra entièrement à un pas de temps donné, et ainsi la couleur d'un pixel correspondra à la moyenne des rayons pour plusieurs frames, reproduisant ainsi le flou d'un temps d'exposition trop long pour un mouvement rapide. On modélisera les paramètres du capteur de la caméra (temps d'exposition, ouverture).



b. « ***Profondeur de champ*** » Implémenter un effet de focus (image nette dans le plan focal, floue ailleurs). Pour cela, on modélisera les paramètres de la lentille dans la caméra.

Bonus « Rendu Interactif »

Implémenter une version *progressive* du path tracing, afin de pouvoir naviguer dans la scène en ayant le rendu qui se raffine lorsqu'on arrête de bouger la caméra. Vous pouvez vous inspirer de la web-app mentionnée plus haut, ainsi que de la version 2.62 de blender (mode de rendu « cycle »).



Plusieurs approches sont possibles ici, notamment le calcul multi-cœur avec OpenMP ou le calcul **GPU** (shaders, OpenCL, CUDA). Il est recommandé de tester en mono-thread CPU et à faible résolution (256x256) dans un premier temps.