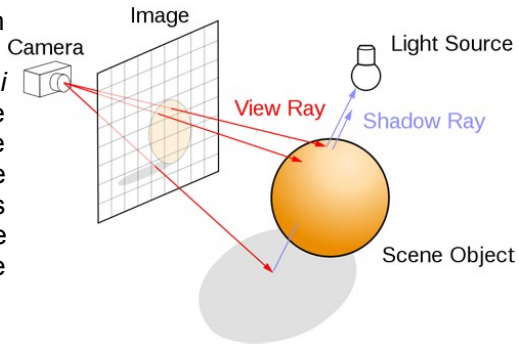
INFSI 350 – Informatique Graphique 3D et Réalité Virtuelle – Projet

Moteur de Rendu en Lancé de Rayon

L'objectif de ce projet est d'implémenter un moteur de rendu en *lancé 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 lancé 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
Observez le code de *raymini,* celui-ci contient déjà tout un environnement pour le lancé 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. Le point d'entrée du projet est la méthode *render* de la classe *RayTracer*.
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 fournit (déconseillée).

Partie 1

1.a Remplacer le code de la méthode *render* (module *RayTracer*) par un algorithme déterminant l'intersection $\{p_i\}$ du rayon $\{r_i\}$ 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(p$_i$)%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*.

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

1.c 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 une structure hiérarchique de partitionnement (Kd-Tree, Octree, BSP-Tree). Le Kd-Tree est conseillé.

Partie 2

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 lancé de rayon pour ajouter plusieurs effets.

2.a 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 plusieures 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.b 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 p_i par v pour remplir le pixel en question. Tester également avec l'évaluation de l'équation du rendu pour chaque échantillon.

2.c 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.d On souhaite maintenant 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 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 (5 rayons sur un pentagone dans le pixel) ou stochastique. Ces expérimentations sont optionnelles.

Partie 3
Implémenter un système d'éclairage global.
Alternative 1 : Path Tracing (http://en.wikipedia.org/wiki/Path_tracing)
Alternative 2 : éclairage global basé point (Point-Based Global Illumination). On s'appuiera sur les travaux de Christensen :

- Notes de cours : http://cgg.mff.cuni.cz/~jaroslav/gicourse2010/giai2010-03-per_christensen-notes.pdf
- Slides : http://cgg.mff.cuni.cz/~jaroslav/gicourse2010/giai2010-03-per_christensen-slides.pdf

Attention, les calculs sont très long, penser à expérimenter à toute petite résolution.

Bonus

a. Implémenter un « debugger de raytracer », permettant de cliquer sur n'importe quel pixel de l'image de synthèse (à droite) et de visualiser le chemin des rayons relatifs à ce pixel dans la vue OpenGL à gauche.

b. 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.

c. Implémenter un effet de focus (image net dans le plan focal, floue ailleurs).