Raymini : Moteur de Lancé de Rayons

Projet du Module INFSI 350 - 2012

Axel Schumacher
Bertrand Chazot
Samuel Mokrani

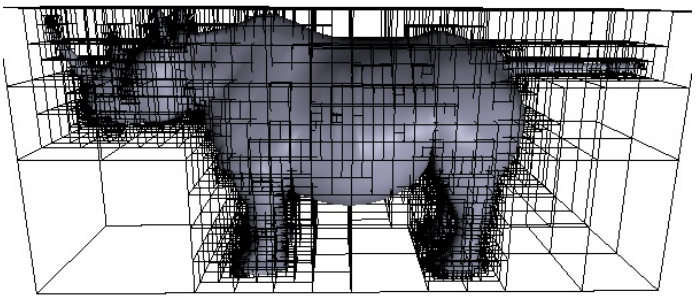
1 - Fonctionnalités du moteur

- Brdf
  - Lambert
  - Phong
- Antialiasing
  - uniforme
  - polygone
  - stochastique
- Ombrage
  - ombres dures et douces (multi-lumières)
  - ambient occlusion
- Eclairage global
  - path tracing
  - Point Based Global Illumination
- Effets
  - effet de focale
  - flou de mouvement
- Tout objet a une texture et une carte de normale qui peuvent être:
  - par le bruit
  - image
  - couleur unie / normale non modifiée initialement calculée
- Matériaux spéciaux
  - miroir
  - glossy surfaces
  - verre (réfraction)
- Transformation d'un mesh
  - rotation
  - homothétie

2 - Points techniques

- ***Kd-tree***

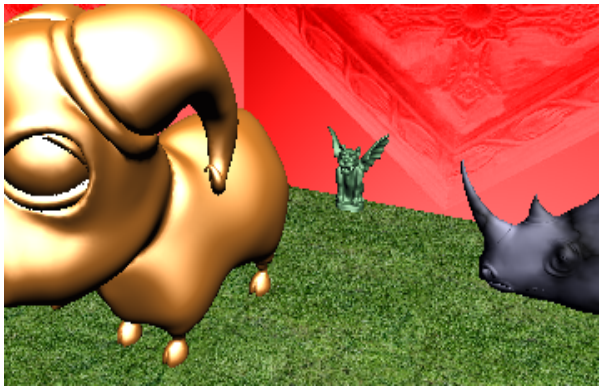
Un Kd-tree a été implémenté pour le stockage des vertex. Il nous permet d'évaluer l'intersection entre un rayon et la scène en complexité logarithmique. Nous utilisons une séparation des nœuds à la médianne.

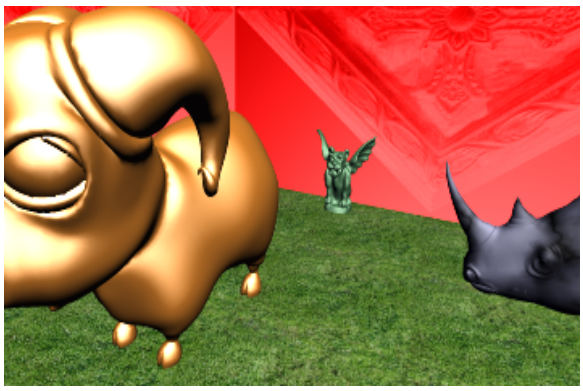


- ***AntiAliasing***

Pour réduire l'effet d'aliasing, on lance plusieurs rayons au sein d'un même pixel puis on fait la moyenne des couleurs rencontrées. Nous avons implémenté trois manières de générer les différents rayons:

  - au sein d'une grille (uniforme)
  - au sein d'un cercle (polygonal)
  - de manière aléatoire (stochastique)

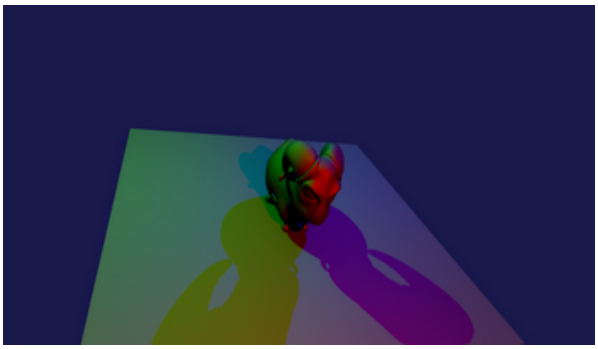




- ***Ombres dures et douces***

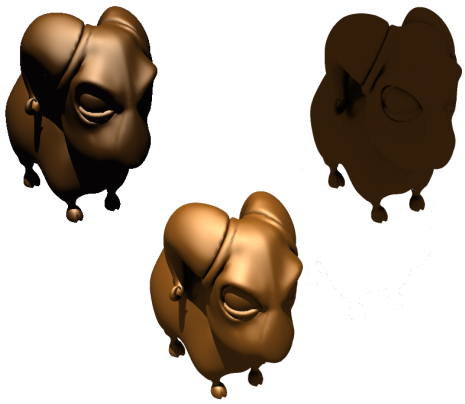
Pour calculer l'ombrage dur en un point, on tire un rayon vers les lumières présentent dans la scène. Si un rayon rencontre la scène avant de toucher la lumière, le point est dans l'ombre. Un coefficient de visibilité est renvoyé: 0 ou 1.

Pour les ombres douces, les lumières ne sont plus des points mais des disques (area light). On tire plusieurs rayons partant du point de la scène vers les lumière. Le coefficient de visibilité est dans ce cas entre 0 et 1.



- ***Ambient Occlusion***

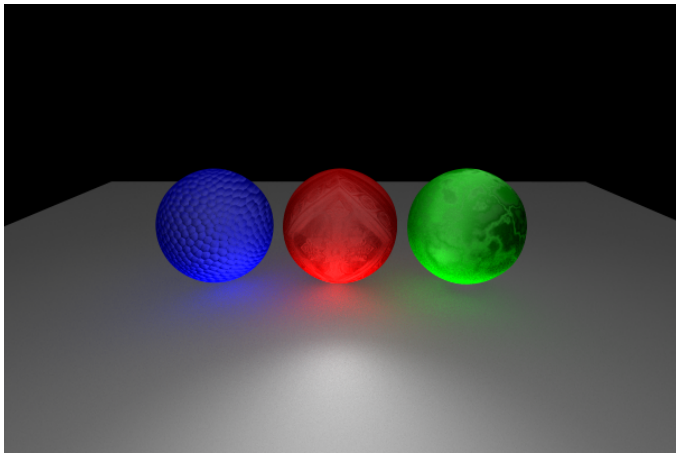
En chaque point, N rayons sont lancés selon un angle maximum autour de la normale. Un coefficient entre [0..1] est ensuite calculé en fonction du nombre de rayons occultés et le nombre de rayons envoyés.



de gauche à droite et de haut en bas:
ombre dure
ambient occlusion
ombre dure et ambient occlusion

- ***Path Tracing***

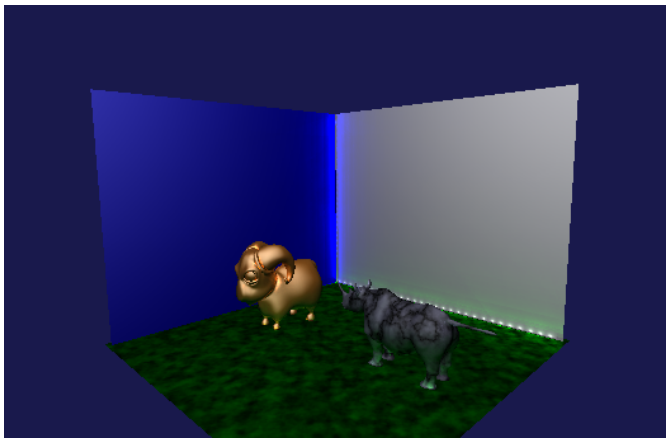
Le Path Tracing implementé est basé sur le document Monte Carlo Path Tracing [1]. C'est un ray-tracing récursif. L'idée est d'envoyer N rayons par pixels puis pour chaque intersection générer un nouveau rayon aléatoire dans une demi sphère autour de la normale au point d'intersection. L'algorithme s'arrête lorsque un rayon ne rencontre pas la scène où lorsque le nombre de rebonds dépasse un certain seuil paramétrable. Nous avons choisi d'ajouter les couleurs rencontrées en fonction de leur distance au point d'origine et du nombre de rebonds effectués.



- ***Point Based Global Illumination***

Des surfels sont calculés en fonction de la scène et des sources lumineuses. Ils sont ensuite mis à jour si un des deux paramètres précédents change. Ces surfels vont jouer le rôle de sources lumineuses secondaires. Ils sont organisés au sein d'un octree.

Puis à chaque intersection entre un rayon lancé depuis la caméra et la scène, on lance des rayons répartis selon un cube dont la résolution est paramétrable. Si un rayon intersecte un surfel suite au parcours de l'octree, on calcule un surfel moyen correspondant à la représentation des surfels se trouvant dans la même feuille que le surfel intercepté. Tous ces surfels moyens représentent des sources lumineuses et sont ainsi injectés dans la BRDF pour le calcul de la couleur.

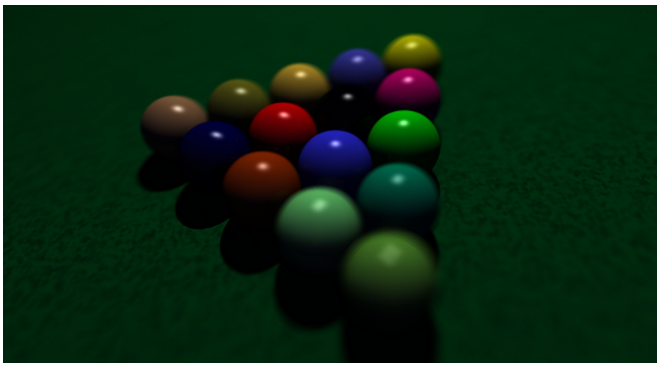


- ***Focus***

Pour avoir un effet de focus, il faut dans un premier temps définir le plan focal. Tout utilisateur de notre logiciel peut le choisir dynamiquement.

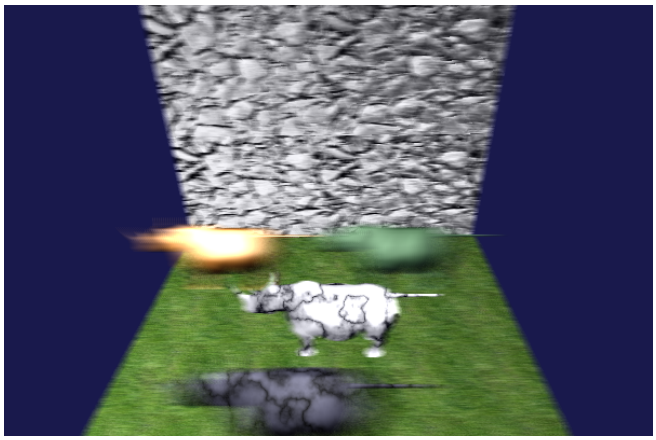
Une fois que l'on a le plan focal, pour chaque rayon lancé depuis la caméra, on va calculer le point théorique d'intersection entre le rayon et le plan focal. Il ne reste plus qu'à lancer ensuite N rayons depuis la caméra passant par le point précédent pour créer un effet de focus.

L'ouverture de la caméra peut également être choisie.



- ***Flou de mouvement***

Le flou de mouvement est crée en faisant une moyenne sur plusieurs images dans lesquelles certains éléments de la scène peuvent s'être déplacés.



- ***Texture mapping et bump mapping***

Chaque triangle connaît les coordonnées (u,v) de chacun de ses sommets.
Chaque matériau a une texture de couleur et une texture de normales, et s'en sert systématiquement pour renvoyer la couleur et la normale en un point.
Il existe 4 types de textures de couleur :

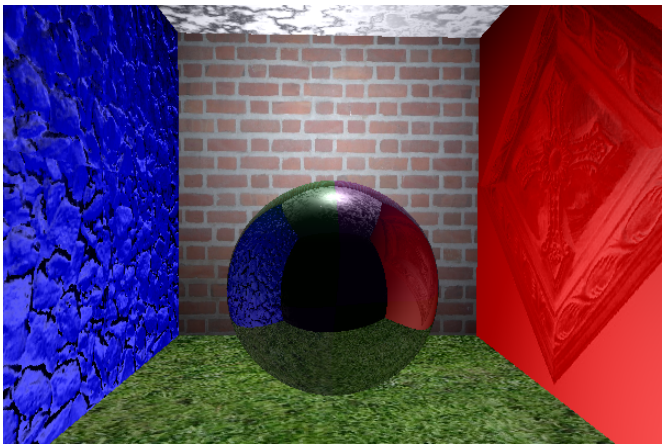
1. SingleColorTexture, qui renvoie toujours la même couleur
2. DebugColorTexture, qui utilise le mapping pour afficher un damier
3. NoiseColorTexture, qui applique une fonction de bruit à une couleur
4. ImageColorTexture, qui utilise le mapping pour afficher une image

Il existe 3 types de textures de normale :

1. MeshNormalTexture, qui renvoie la normale "normale" au point choisi

2. NoiseNormalTexture, qui utilise une fonction de bruit pour calculer la normale
3. ImageNormalTexture, qui utilise le mapping pour lire une normale dans une image

Pour le mapping, deux autres variables, uScale et vScale, spécifique à chaque objet, permettent de répéter ou bien de prendre une partie d'une texture mappée. Par exemple, mettre uScale et vScale à 2 affichera une ImageColorTexture 4 fois sur un mur.



Sur l'image précédente, il y a du texture mapping pour le sol et le mur du fond. Pour les deux murs sur le côté, une carte de normale a été appliquée. Pour le plafond, la normale est calculée à l'aide d'un bruit de Perlin.

- ***OpenMP***

  Chaque ligne de pixels est traitée dans un thread séparé grâce à OpenMP.

- ***Temps réel (CPU)***

  Un thread dédié au rendu permet de constamment redessiner la scène à faible résolution lorsqu'un modèle est modifié ou que la caméra a bougé, et de ramifier jusqu'à la qualité optimale dès que la caméra est fixe.

  Pour cela, il existe 3 types de qualité : OPTIMAL, qui tient compte de tous les paramètres, et qui est donc le plus lent, BASIC, qui désactive certains paramètres (anti aliasing, PBGI, ombres douces, flou de mouvement, focus et ambiant occlusion), et ONE_OVER_X, qui conjointement à une variable qualityDivider, ne va calculer qu'un pixel tous les qualityDivider², en qualité BASIC.

  L'utilisateur peut choisir la qualité la plus basse qui sera utilisée lors du temps réel, et le programme remontera rapidement jusqu'à OPTIMAL dès que la caméra arrête de bouger.

  Le path tracing se raffine en permanence lorsque la qualité OPTIMAL est atteinte.

- ***Contrôle avancé avec la GUI***
  - *"FPS mode" pour le focus en OpenGL*

En mode OpenGL, on peut changer le point de focal en visant le point souhaité avec le milieu de l'écran, pour ensuite le fixer et déplacer la caméra au bon endroit.

- *Scene "dynamique"*

Les objets, lumières, matériaux et textures sont modifiables "on the fly", que ce soit en rendu OpenGL ou RealTime.

Les lumières sont représentées on OpenGL par un cube lorsqu'elles sont sélectionnées, permettant de les placer avec précision. On peut aussi choisir leur couleur, leur rayon (pour les ombres douces) et leur intensité. On peut aussi les désactiver.

Chaque objet a un matériau, sélectionnable dans une liste, des coordonnées de position et de déplacement (si au moins un objet a des coordonnées de déplacement non nulles, l'option Motion Blur devient disponible). On peut aussi tout simplement le désactiver.

Pour chaque matériau, on peut modifier ses coefficients Diffuse, Specular et le ratio Glossy (1 pour un miroir parfait, 0 pour aucune réflection). On peut aussi choisir ses textures de couleur et de normales.

En bref, pratiquement TOUS les paramètres de la scène peuvent être modifiés à la volée, sans même arrêter le rendu temps réel.

- *Drag objects*

En mode RealTime, on peut demander à pouvoir déplacer les objets en cliquant dessus, et en les amenant à l'endroit voulu. Cela modifie l'attribut trans de l'objet, qui est son offset par rapport à l'origine, sans modifier le mesh.

- ***MVC design pattern***

Les vues sont constamment synchronisées aux modèles, et chaque manoeuvre de l'utilisateur a des répercussions sur tout le programme, de manière parfaitement encadrée et facile a compléter.

MVC signifie en français Modèle Vue Contrôleur, et est un pattern de programmation fréquemment utilisé dès lors qu'une GUI est nécessaire.

Chaque modèle est une classe qui connaît certains paramètres courant du programme. Ici, ce sont Scene, RayTracer, PBGI, RenderThread et WindowModel. Ce sont des Observables.

Chaque vue est une classe qui affiche certaines informations à l'utilisateur, et qui lui permet de modifier le modèle. Ici, ce sont GLViewer, Window et MiniGLViewer. Ce sont les Observers.

Chaque Observable connaît une liste d'Observers.

Le contrôleur est l'élément central de synchronisation : il connaît tout ce beau monde, et les vues ne peuvent modifier le modèle que par son intermédiaire, assurant la cohérence entre toutes les vues et les modèles.

En effet, à chaque action de l'utilisateur, la vue recevant l'évènement va appeler la fonction correspondante du contrôleur, qui va modifier les modèles en conséquence.

À chaque fois qu'un modèle est modifié, il va mettre un bit de caractérisation de sa modification à 1. Le contrôleur va ensuite demander à chaque modèle de prévenir tous ses observers avec la méthode notifyAll(). Chaque vue va appeler la fonction update(), et se mettre à jour en fonction du modèle modifié et de ses bits de modification à 1. Chaque modèle va ensuite remettre ses bits à 0 et le programme continue son cours jusqu'à la prochaine intervention de l'utilisateur.

Un exemple flagrant de la puissance du MVC est quand on déplace des objets à la souris en mode temps réel avec le mode "Mouse moves objects" activé, et le panel Objects affiché dans la barre d'outils. À chaque fois qu'un objet est bougé à la souris, on peut voir les coordonnées x, y et z de l'objet changer dans la barre d'outils, montrant la constance dans la cohérence du programme.

3 - Améliorations

Même si nous avons du temps réel implémenté sur CPU, nous aurions aimé implémenter notre moteur sur GPU. Une ébauche de ce travail a été développé en OpenCL. Nous pouvons transmettre une scène complète au GPU qui se charge de faire le lancer de rayons et remplir un buffer de pixels. Ce buffer de pixels est ensuite affiché à l'écran. Mais il nous manque un élément majeur pour arriver à avoir du temps réel: un Kd-tree sur GPU.

Concernant le Kd-tree, notre implantation de la métrique SAH n'apportant pas de gain nous avons gardé une séparation à la médianne. SAH étant adapté au raytracing, il faudrait retravailler cette approche.

4 - Bibliographie

[1]: http://etausif.com/Documents/MonteCarloReport.pdf. Rapport publié en Mars 2010 par Ahmed Tausif Aijazi et Carlo Navarra.

[2]: http://graphics.pixar.com/library/PointBasedColorBleeding/paper.pdf. Rapport sur le Point Based Approximate Color Bleeding de H. Christiensen.