RenderBoy

why you'll love it!

Fast kD-Tree Rendering

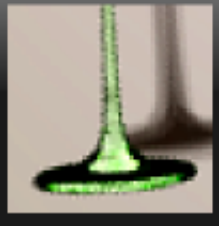
Your picture will be rendered using a kD-Tree, using a fully parallelized algorithm, that allows speedups of more than 60x, compared to what you were seeing before!

Soft and hard shadows

For added realism, Renderboy allows you to add life to your scenes with hard shadows, as well as defining area lights for physically realistic soft shadows.

Realistic refraction and reflection

Did you know you can even model glass with RenderBoy? Play with light and convey the feelings you want to, be it with water, glass or transparent things!



Anti-Aliasing

You want your renderings to be sharp and just what you intend them to be? Our x4 anti-aliasing does just that! Each edge will be crystal clear, without any artifacts.

Global Illumination

Even if you don't see the light, we do! RenderBoy takes your scene further with Point-Based Global Illumination. No more static, uniform colors!

Yann Jacquot & François-Xavier Thomas
http://bitbucket.org/fxthomas/renderboy

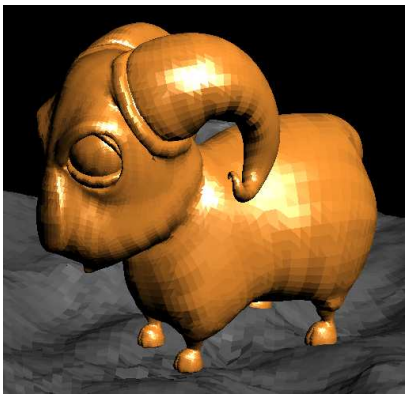
INFSI350 – Projet

by Yann Jacquot & François-Xavier Thomas

L'objet de ce projet a été d'implémenter un moteur de rendu utilisant le lancer de rayons, et disposant d'autres fonctionnalités.

Premières versions

Pour démarrer, nous nous sommes servis du code fourni en début de projet. La première version utilisable du programme consistait en un lancer de rayon classique.



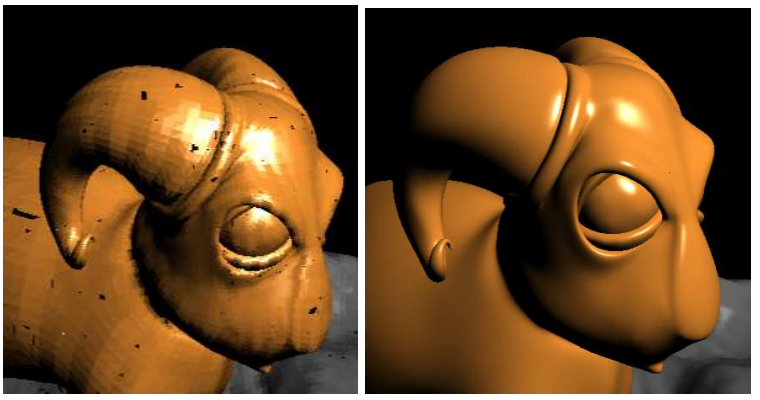
Le problème est que de cette manière le rendu est extrêmement lent. En effet, il nous a fallu près d'une heure pour obtenir l'image ci-dessus.

kD-Tree

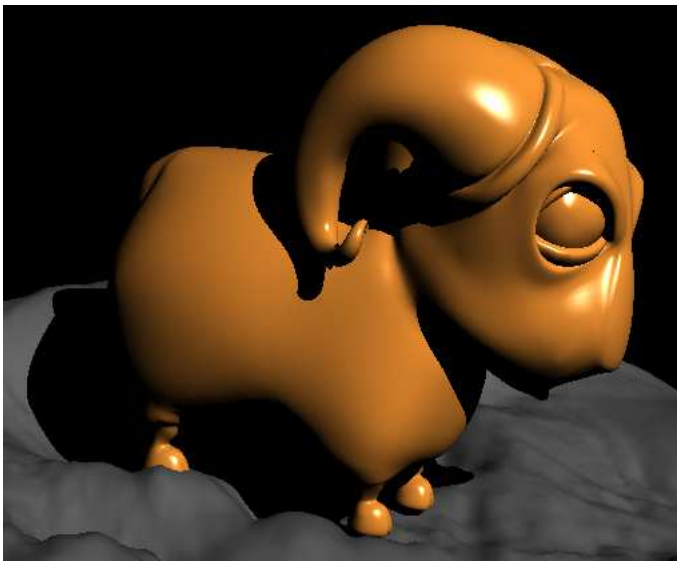
Pour remédier à cela, une solution consiste à se servir d'une structure d'arbre afin d'accélérer le

rendu. Nous avons choisi la structure de kD-tree (comme tous les autres groupes), bon compromis entre vitesse et facilité d'utilisation.

Lors de la construction du kD-tree nous nous sommes rendus compte du gain de temps considérable apporté par cette structure de données (la même scène mettait moins d'une minute à apparaître). Cependant, le résultat obtenu comportait de nombreux trous de dimension variable. Nous avons alors décidé de définir un coefficient de flou, agrandissant les boîtes englobantes d'un faible pourcentage afin d'éviter ces trous. Cela se fait aux dépens de la rapidité d'exécution.



Nous nous sommes alors attaqués au rendu des ombres, en commençant par les ombres dures. La méthode utilisée est simple. Lors du lancer de rayon, avant de calculer la BRDF, nous vérifions si le chemin entre le point d'intersection et la source de lumière n'est pas obstrué par un obstacle. Si c'est le cas, c'est que le point considéré est dans l'ombre, la couleur associée sera donc le noir. On profite là encore de la structure de kD-tree pour avoir un rendu suffisamment rapide.

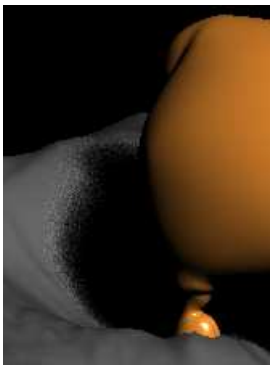


Ombres dures et douces

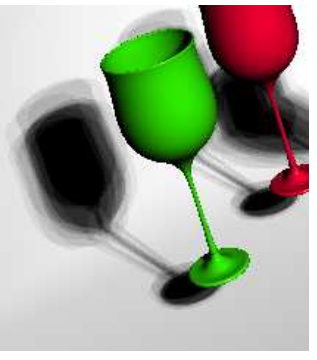
Nous avons ensuite cherché à modéliser les ombres douces, obtenues lorsqu'il a y a des sources de

lumière étendues. Nous échantillonnons un nombre de points (défini à l'avance) sur le disque faisant office de source de lumière. A chacun de ces points on associe un rayon reliant le point d'intersection et le point échantillonné sur la source de lumière. Nous définissons alors un coefficient de visibilité, correspondant à la proportion de ces rayons n'étant pas obstrués par des triangles du maillage. La couleur finale correspond à la multiplication du résultat de la BRDF par ce coefficient de visibilité.

Si l'on prend des points sur la source de lumière différents pour chaque point d'intersection considéré, on obtient des ombres très bruitées (le coefficient de visibilité étant partiellement aléatoire). Nous avons décidé de garder le même échantillon de points afin de limiter ce bruit. Cela a pour effet un résultat plus agréable à l'oeil, même si selon l'évolution de certains paramètres (nombre de points choisis sur la source, taille de la source de lumière, taille de l'objet), on observe plus ou moins facilement le fait que l'ombre obtenue correspond à une superposition de plusieurs ombres claires légèrement décalées.



Points totalement aléatoires



Pré-calcul (16 points)

Pré-calcul (64 points)

Anti-aliasing

Nous nous sommes alors occupés de la réduction du phénomène de crénelage (aliasing). Nous avons juste multiplié le nombre de rayons lancés depuis la caméra, puis effectué une moyenne des contributions pour obtenir la couleur d'un pixel.


Sans anti-aliasing *Avec anti-aliasing* $\times 4$

Réfraction et réflection

Il est très simple, une fois le moteur de lancer de rayons défini, construire des rayons réfractés et réfléchis.

Cependant, comment doser précisément la le pourcentage d'irradiance affectée à la réflection, à la réfraction et à l'éclairage direct? Nous avons introduit deux coefficients : r_{refl} et r_{refr}, respectivement indiquand le pourcentage de lumière réfléchie et réfractée. Pour la lumière directe, il va de soi que plus le matériau laisse passer la lumière réfractée (ou non), plus il est transparent et moins il a de couleur propre. L'irradiance totale est donc obtenue par la formule :

$$i_{\text{total}} = i_{\text{refl}} \cdot r_{\text{refl}} + i_{\text{refr}} \cdot r_{\text{refr}} + i_{\text{direct}} \cdot (1 - r_{\text{refr}})$$

Point-Based Global Illumination

Même si nous n'avons pas implémenté dans se totalité l'algorithme de *Point-Based Global Illumination*, nous avons néanmoins un algorithme basique qui, même si cela n'est pas très visible, augmente un peu le réalisme de la scène.

Pour combiner cette illumination globale avec l'illumination déjà calculée (et provenant des réflections, réfractions et éclairage direct), nous avons pris le parti de calculer la demi-somme des deux contributions, la contribution de chaque surfel dans l'éclairage global étant pondérée par la surface de celui-ci. (Ici, on note i_s l'irradiance produite par un surfel, et S_s la surface de celui-ci)

$$i = i_{\text{total}} + \frac{\sum_s i_s S_s}{\sum_s S_s}$$

Conclusion et améliorations à apporter

Une des principales choses à améliorer, dans ce moteur de rendu, serait la vitesse. En effet, les triangles sont stockés de manière non-optimale dans le kD-Tree. Nous n'avions pas beaucoup de temps, donc nous avons dû user d'astuces un peu sales, qui n'ont pas amélioré les choses. La vitesse obtenue est globalement satisfaisante pour les scènes simples (de 30s à 1min), mais, dès qu'on commence à utiliser des modèles un peu plus évolués, le rendu prend largement plus de temps (plusieurs heures parfois avec ombres douces, global illumination, réfraction, réflection et anti-aliasing).

Une autre piste pour l'amélioration serait d'implémenter la totalité de l'algorithme pour la *Point-Based Global Illumination*, et de surtout stocker *tous* les surfels possibles (et non pas un échantillon de taille réduit) dans un *octree*, comme indiqué par Christiansen.