LECOEUR Clément Mai 2020

Introduction à la Synthèse d'Images Réalistes

Projet : Création d'un moteur de lancer de rayons

I – Techniques utilisées

Dans ce projet, j'ai réalisé les techniques suivantes :

- Calculs d'intersection entre un rayon et une sphère, un triangle, une boite et un plan.
- Gestion des maillages d'un triangle : lorsqu'un objet est créé, on récupère une liste de triangles et on parcourt donc cette liste pour afficher l'objet 3D.
- Utilisation d'un BVH.
- Calcul des réflexions et des réfractions.
- Une BRDF à microfacette de type CookTorrance
- Une source lumineuse ponctuelle pour le calcul des ombres dures.
- Une source lumineuse sphérique pour le calcul des ombres douces.
- De l'anti-aliassage en utilisant la technique du supersampling.
- Une surface implicite sous la forme de métaballs, fait à partir d'objets sphères.
- Une texture procédurale pour faire le damier sur le plan.

II – Détails des techniques

A - Bounding Volume Hierarchy

Pour le BVH, le cours de M. Frederic CLAUX m'a été suffisant pour les explications. Cependant j'ai rencontré deux problèmes :

- Une mauvaise gestion lorsque un des deux enfants renvoyés *null*, alors je n'affichais rien, même si l'autre enfant n'était pas *null*. Je suis resté longtemps bloqué sur ce problème, mais M. Guillaume GILET a fini par me debloqué.
- Lorsque je testais les différents SAH, je n'effaçais pas ces tableaux. Alors lorsque le mesh devenait trop gros, je me suis heurté à une erreur de type " Stack Overflow ". J'ai cependant corrigé ce problème assez rapidement en effectuant une meilleure gestion mémoire et en ne gardant pas les tableaux inutiles.

Pour finir, le BVH qui est utilisé dans ce projet est vraiment basique et pourrait être grandement amélioré.

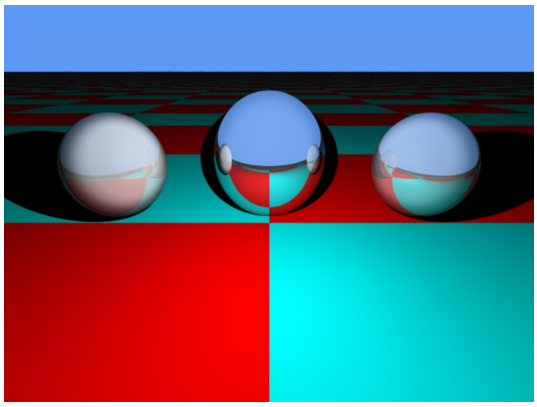
Tout d'abord, il ne prend en compte que l'axe *x* pour les calculs de SAH. J'ai fait ce choix car la majeur partie de ma scène s'étale sur cet axe et non les autres, mais pour de meilleures performances sur des scènes qui s'étaleraient sur un autre axe, il pourrait être judicieux de tester les SAH sur d'autres axes.

Ensuite, le BVH ne contient pas les primitives de bases telles que les sphères et le plan. J'ai fait ce choix car je n'ai que très peu de ces objets, et leurs intersections sont très rapides à calculer, alors que lorsque j'ajoute un objet 3D, le maillage contient de nombreux triangles et c'est eux qui ralentissent le plus le programme. Mais encore une fois, pour de meilleures performances, il pourrait être judicieux que le BVH prenne en compte tous les objets de la scène.

B – Reflexion et réfractions

C'est cette partie qui m'a parrue la plus difficile, et principalement les réfractions.

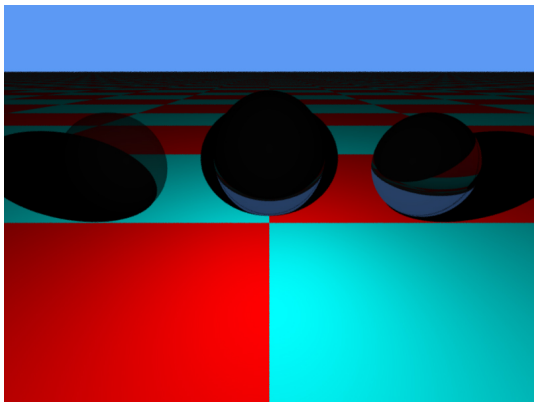
Pour les reflexions, le programme calcule la couleur de reflexion à partir d'une fonction récursive qui relance des rayons dans la direction de reflexion du point touché, obtenue à partir de la fonction *glm::reflect*, jusqu'à ce que le nombre maximum de récursions défini dans defines.hpp soit atteint. J'ai choisi 10 pour les images présentes dans ce rapport.



Réflexions avec des indices différents

Pour les refractions, le programme la couleur de refraction à partir d'une fonction récursive qui relance des rayons dans la direction de refraction du point touché, obtenue à partir de la fonction *glm::refract*, jusqu'à ce que le nombre maximum de récursions défini dans defines.hpp soit atteint. Il faut cependant faire attention à plusieurs choses ici :

- On a besoin de savoir si on est dans ou hors de l'objet afin d'utiliser correctement les indices de refractions de chaque milieu. Dans ce projet, je considère soit que nous somme dans l'objet, soit nous sommes dans l'air extérieur. Nous ne pouvons pas être dans le cas où nous passons d'un objet à un autre sans passer par l'air.
- Il ne faut pas calculer la normale à l'envers, si on est dans l'objet, la normale va vers l'objet, sinon elle va vers l'exterieur.
- *Glm::refract* renvoie un vecteur nul lorsque l'on est dans le sens de reflexion. Dans ce cas, soit on peut utiliser le résultat de *glm::refract* ce qui va afficher des résultats qui serait impossible à obtenir dans la vie réelle, soit on choisi d'afficher le resultat obtenue avec *glm::reflect*.

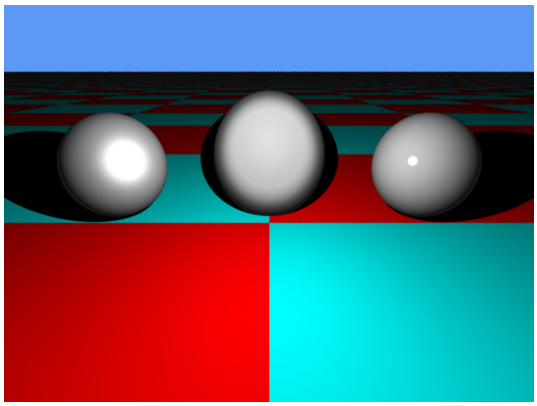


Réfractions avec différents indices

C – BRDF type Cook-Torrance

Pour cette BRDF, j'ai pu voir sur internet qu'il existait de nombreuses fonctions de distribution, ainsi que de fonctions calculant le terme de masquage / ombrage. J'ai choisi d'utiliser la distribution de Beckmann comme nous l'avions vu en cours, mais j'ai utilisé la géométrie de Cook-Torrance plutôt que celle de Shlick-Beckmann car cette dernière m'affichait de mauvais résultats lorsque les angles étaient trop rasants.

Tout ces calculs sont présents dans le fichier du dossier BRDF, CookTorrance.cpp.



Différents indices de rugosité

D – Sources lumineuses

Pour les sources lumineuses, j'ai implémenté la source lumineuse ponctuelle définie dans pointLight.hpp, et la source lumineuse sphérique définie dans sphereLight.hpp. Ces deux objets héritent de Light.hpp, et doivent donc être définis avec une position et une couleur. La sphereLight nécessite également un rayon.

Pour la pointLight, lorsque l'on accède à sa position, alors on renvoie directement sa position.

Pour la sphereLight, lorsque l'on fait la même chose, alors on obtient un point généré aléatoirement de manière uniforme sur la surface de la sphère. Pour cela, on génère un theta et un phi sur la sphère dans les coordonnées polaires, puis on transforme ces dernières en coordonnées cartésiennes.

Lors du calcul des ombres, avec une pointLight, on obtient alors des ombres dures (soit il y a un objet entre le point d'intersection et la lumière, soit il n'y en a pas), pour la sphereLight, on obtient des ombres douces dès que l'on génère plus d'un rayon d'ombre, puisque l'on fera alors une moyenne avec le résultat de chacun de ces rayons.

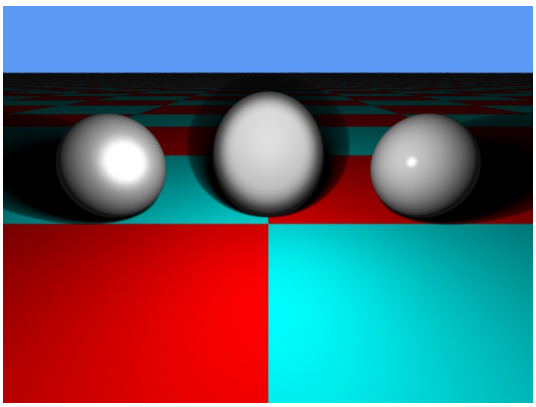


Image avec ombres douces

E – Anti-aliassage

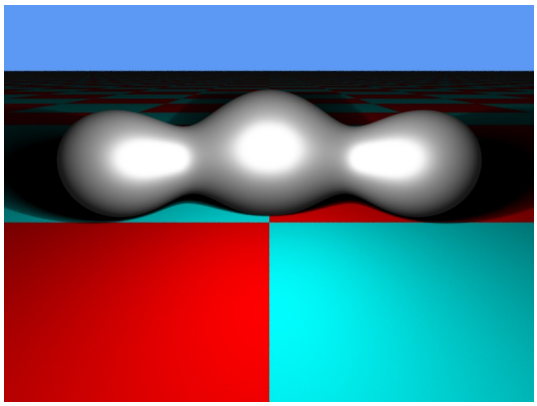
Pour l'anti-aliassage, j'ai utilisé la technique du supersampling. Pour celà, en fonction du nombre de rayons défini dans defines.hpp pour l'anti-aliassage, on divise chaque pixel en quatre, puis on envoie un rayon aléatoirement dans chacun de ces sous-pixels. Enfin, on calcule la couleur moyenne en fonction de chacun de ces rayons.

F – Surface implicite

Pour les surfaces implicites, j'ai utilisé la méthode du Sphere Tracing afin de représenter des metaballs.

Pour cela j'ai également utilisé une fonction de distance signée pour connaitre le champ potentiel d'une sphère de la metaball.

On avance donc le long du rayon selon un pas, ce pas est égal à la distance entre le point actuel sur le rayon et la metaball. Lorsque ce pas est assez proche, alors on a une intersection avec la metaball. Sinon si le pas est arrivé jusqu'à une distance maximum définie dans defines.hpp, alors on peut s'arrêter il n'y a pas d'intersection.



Surface implicite représentant des metaballs

G – Textures procédurales

Pour les textures procédurales, j'ai implémenté le damier, qui est une des textures procédurales les plus simples, on utilise une fonction périodique, comme par exemple sinus dans ce projet, et on affiche différentes couleurs selon si le résultat dépasse un seuil ou non.

J'ai également pu voir sur internet que l'on pouvait faire des textures plus complexes en utilisant un bruit de perlin, comme par exemple du marbre, mais je ne les ai pas implémentées.

III – Annexes

Afin de réaliser ce projet, j'ai utilisé les sources suivantes :

- http://www.realtimerendering.com/intersections.html pour des explications sur le calculs des différentes intersections utilisées.
- https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-ray-tracing/adding-reflection-and-refraction pour des explications sur le calculs des réfractions.
- http://www.codinglabs.net/article_physically_based_rendering_cook_torrance.aspx pour des explications sur le calcul d'une BRDF type Cook-Torrance.
- http://corysimon.github.io/articles/uniformdistn-on-sphere/ pour des explications sur le calcul d'un sampling uniforme d'une sphère.