BEGG Rain-Alexandra (TDA1.1)

TP IGTAI – Lancer de Rayons

1.1 Calcul d'Intersection

- Intersection rayon-sphère OK
- Intersection rayon-plan OK
- Test du lancer de rayons OK :

1.2 Couleur et *shading*

- Fonction shade OK

1.3 Ombres

- Ombres OK

2.1 Distribution de Beckmann et terme Fresnel

- RDM_Beckmann et RDM_Fresnel OK

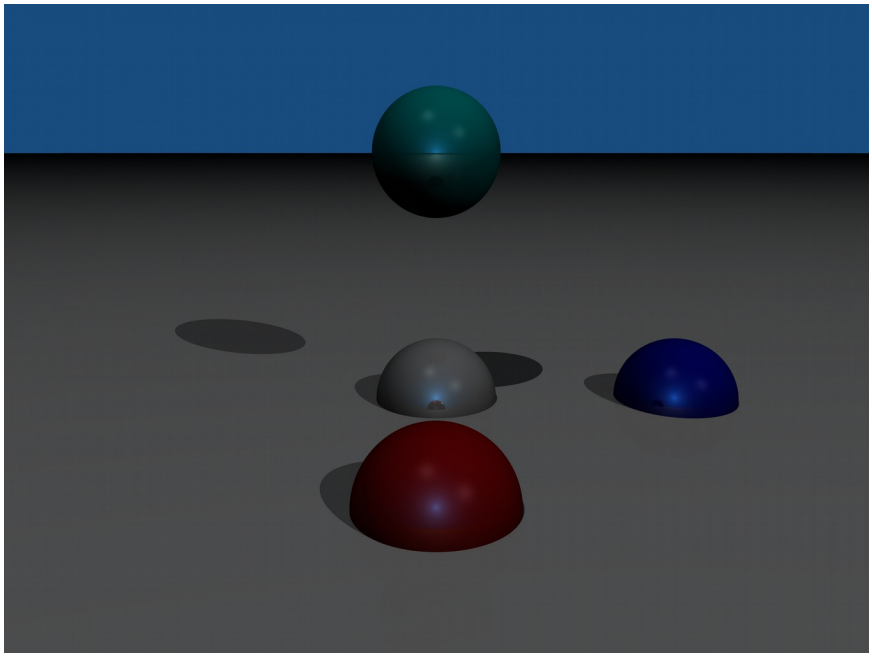
2.2 Implémentation de la BSDF

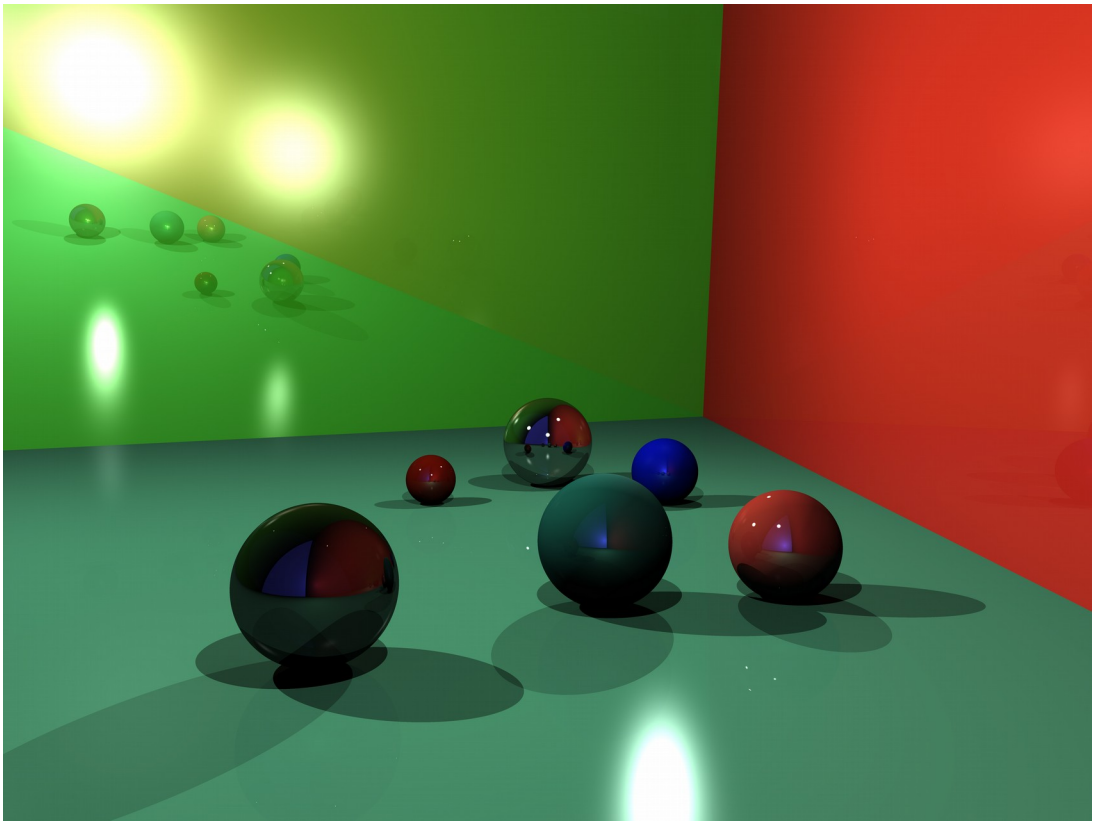
- RDM_G1 OK
- RDM_Smith OK
- RDM_bsdf OK
- shade OK

2.3 Réflexions

- Nouveau trace_ray OK

J'ai rencontré un problème au niveau de cette partie qui m'amenait aux résultats suivants :





En effet, les réflexions semblait être « trop fortes ». De conséquence les matériaux qui à la base ne devait pas refléter le faisaient quand même.

Pour régler ça, j'ai moduler la couleur du reflet par la couleur spéculaire du lumière, ainsi au lieu il fallait modifier le calcul de la couleur finale (avec `F = RDM_Fresnel(LdotH, 1, intersection.mat→IOR` et `cr = trace_ray(scene, &newRay, tree)`) :

~~`color += (F * cr);`~~

```
color += (F * cr * intersection.mat→specularColor);
```

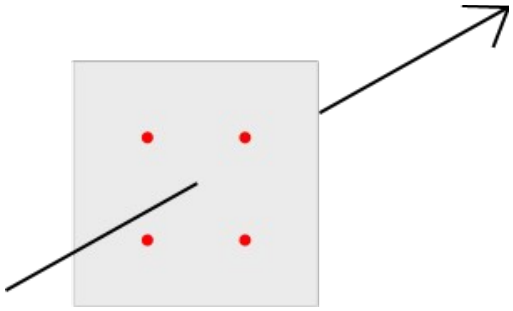
3.1. Antilaliasing

- Anti aliasing OK.

J'ai compris que dans la fonction renderImage, on parcourt chaque pixel à un indice i, j de l'image pour faire passer un rayon au travers. J'estimais que chaque pixel avais une dimension de 1, donc au lieu de faire passer un seul rayon, j'en faisais passer cinq, en ajoutant ou en enlevant 0.25 du i et j à l'initialisation d'un direction de rayon. Par exemple,

```
vec3 ray_dir3
= scene->cam.center + ray_delta_x + ray_delta_y + float(i - 0.25)*dx +
float(j + 0.25)*dy;
```

Voici l'idée que j'envisageais, avec le rayon en noir étant le rayon de base, et les points rouges étant là où passent les quatre nouveaux rayons :



La couleur finale est donc la moyenne des couleurs obtenues par ces rayons.

3.2 Kd-tree

Je n'ai pas pu faire le K-d tree mais j'ai tenté l'implementation du bbox du root du tree.

J'initialise un point min très grand et un point max très grand pour le bbox et je parcours chaque objet de la scene. Si c'est une sphère (dans mon implémentation je ne prend que les sphères en compte), je calcule ses coordonnées min en prenant les coordonnées de son centre et puis en enlevant la valeur du rayon de ces coordonnées là (je fais pareil pour le max mais au lieu d'enlever le rayon je l'ajoute). A chaque itération je modifie le min et le max du bbox.

3.3 Autre objets

Triangle OK

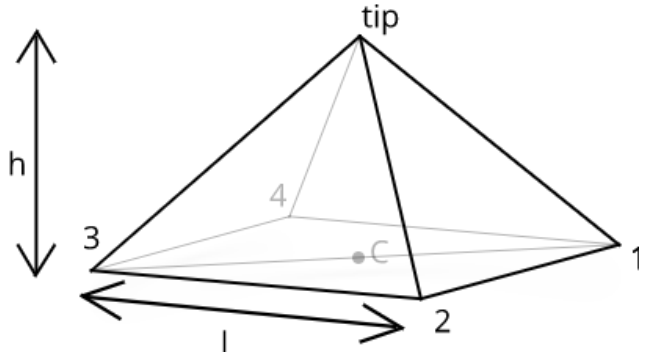
Ajout de la structure *triangle* dans la structure Geometry. Un triangle est paramétré par ses trois points a, b, c.

Implémenté en utilisant l'algorithme Möller-Trumbore. J'avais des soucis par rapport au calcul de la normale d'intersection, ce qui faisait que j'avais un triangle noir même si il était face à une lumière et bloqué par rien. Il fallait que je teste si la normale du triangle était de la même direction que le rayon (si `dot(ray→dir , triNorm) ≥ 0`). Dans ce cas la normale de l'intersection sera opposée à celle du triangle.

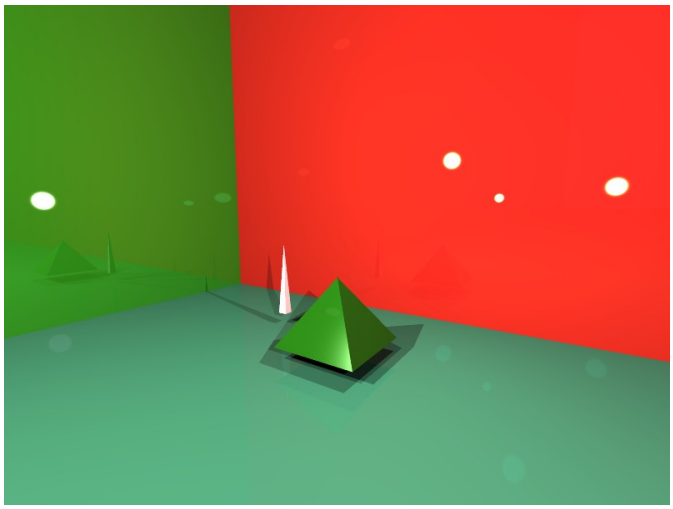
Pyramide OK (Scène 6)

Ajout de la structure *pyramid* dans la structure Geometry. J'ai choisi de la paramétrer par une hauteur h, une longueur l (de sa base carré), un centre C (la où il sera placé dans la scène), et un tableau de 6 triangles, 4 pour ses et 2 pour former le carré de la base.

Son initialisation fait appel à la fonction de `initTriangle` pour chacun des triangles qui lui formera. Donc, je passe en paramètre les points qu'il faut (qui sont calculés avec la longueur et la hauteur) pour chacun. Par exemple, un des triangles de la base sera paramétré par le point 1, 2 et 3 et une des face sera paramétré par les points tip, 1 et 4.



Problème : avec mon spécification, on ne pas orienté le triangle sur un axe, donc sa base sera toujours orthogonal par rapport aux axes x et z :

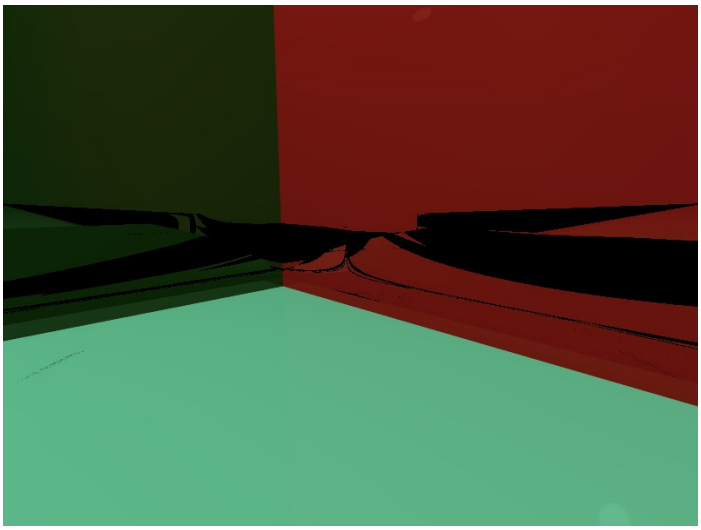


Cylindre KO Cone KO

En faisant des recherches j'ai remarqué qu'il existe beaucoup de représentations différentes pour le cylindre et le cone, mais j'ai choisi de les paramétrer par un centre, une hauteur, et un rayon.

Échec → Il y avais sûrement d'autres erreurs dans mon implémentation mais je suis certaine que la normale d'intersection était une des causes d'images pareils :

CYLINDRE



CÔNE

Maillage OK (Scène 7)

Ajout de la structure *mesh* dans la structure Geometry. J'ai choisi de le paramétrer par le nombre de vertices, le nombre de triangles, un tableau de points (pour les vertices) et un tableau de triangles.

À partir d'un fichier .obj, je récupère les sommets (aux lignes 'v', des fois je suis obligée de les mettre à l'échelle, ça dépend du fichier) et les faces (c'est à dire j'initialise et je stocke des triangles dans ma structure *mesh* à partir des indices des sommets indiqués aux lignes 'f'). Donc mon `intersectMesh` parcours les triangles de l'objet et fait appel à `intersectTriangle` pour chacun.

J'utilise des maillages « low poly », mais la génération des images est assez lent (en effet, il y a 1508 triangles dans le maillage du cerf). Il prend entre 30 et 50 minutes avec anti aliasing et 15 mn sans. C'est là où le kd-tree aurait été très utile.