Compte rendu Projet Lanceur de Rayon

Fin Phase 3

Ludovic LONLAS

8 janvier 2023

Workflow

Depuis le dernier compte rendu, mon workflow n'a pas beaucoup changé, cependant durant les vacances d'hiver je n'étais pas chez moi mais chez mes parents, ne pouvant pas amener ma machine, j'ai fais du télétravail.
J'ai utilisé l'application de bureau à distance Parsec pour pouvoir continuer à travailler durant mes vacances, malheureusement la connections n'étais pas très bonne et la latence fut très élevé.

Optimisation, optimisation et optimisation

Réorganisation

Le squelette même si très utile n'est pas très organisé, de plus le compilateur ignorait les Header donc j'ai réorganiser l'intégralité du projet.

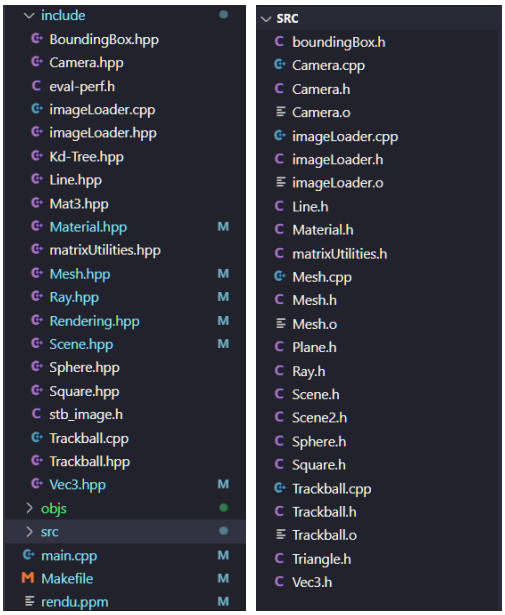

FIGURE 2.1 – À gauche ma structure, à droite celle du squelette

Kd-Tree

La partie la plus importante de cette phase du Projet est le Kd-Tree, je n'ai cependant pas eu trop de difficulté à l'implémenter.
Le Kd-Tree sert à accélérer le calcul des intersection, il est principalement utile pour les maillages.

La structure de mon Kd-Tree

Mon Kd-Tree hérite de ma classe de boîte englobante que j'ai réalisé précédemment, cette boîte englobante calcule les intersections en vérifiant si une intersection existe avec chacune de ces faces représenté par des Squares. Avant de choisir des squares, j'utilisais 2 triangles mais les performances étaient évidement bien pire qu'avec des carrées.

Le Kd-tree contient aussi deux enfant ainsi que la liste de tout les triangles se situant dedans. Avant de tester l'intersection avec les faces d'un maillage, on teste l'intersection avec son Kd-Tree et de tout ces enfants jusqu'à atteindre une feuille, si on n'atteint pas une feuille cela signifie qu'il n'y a pas d'intersection. On renvoie alors la liste des triangles de la plus petite Boîte ou une liste vide si il n'y a pas d'intersection.

```
1
2  private:
3      KdTree* child1;
4      KdTree* child2;
5      Mesh* ReferenceMesh;
6      std::vector<unsigned int> trianglesInBox;
7      std::vector<Vec3> centerOfTriangles;
8      void sortTriangleByDir( unsigned char xyz);
9
```

La génération prend beaucoup de temps mais est très utile, quand je génère l'arbre je lui donne un nombre de répétition, cela me permet à l'aide de Log2 d'obtenir des feuilles d'une taille précise. J'ai trouvé que le Kd-Tree est plus rapide avec une taille 64 triangles dans chaque feuille. Pour séparer en deux l'arbre, je trie sur l'un des angles tout les triangles, je coupe en deux la liste et je donne chaque moitié à à chacun de ces enfants et je recalcule leurs Boîte englobante avec les triangles qu'ils possèdent.

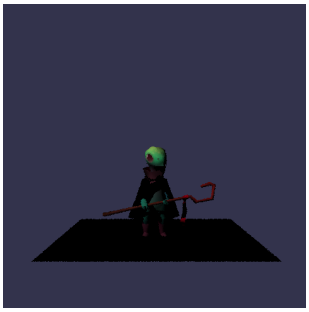

FIGURE 2.2 – Rendu d'un modêle de 888 Triangles en 4.8s

Multi-Threading

Pour améliorer les performances et faire fondre mon processeur J'ai implémenté en même temps que j'ai réorganisé mon code du multi-threading. Je créer autant de thread que possible et je leur associe la même fonction, cependant il faut que je les resynchronise à l'arrêt et que je leur donne toutes les données nécessaire en argument. les threads perdent aussi le contexte openGL donc je dois aussi lui donner les informations de la caméra.

Tant que les threads ne se sont pas tous rejoint, j'affiche les parties de l'image déjà calculer, cela me permet de savoir à l'avance le temps que va prendre un rendu et aussi de savoir si il y a un plantage quand il se produit précisément.

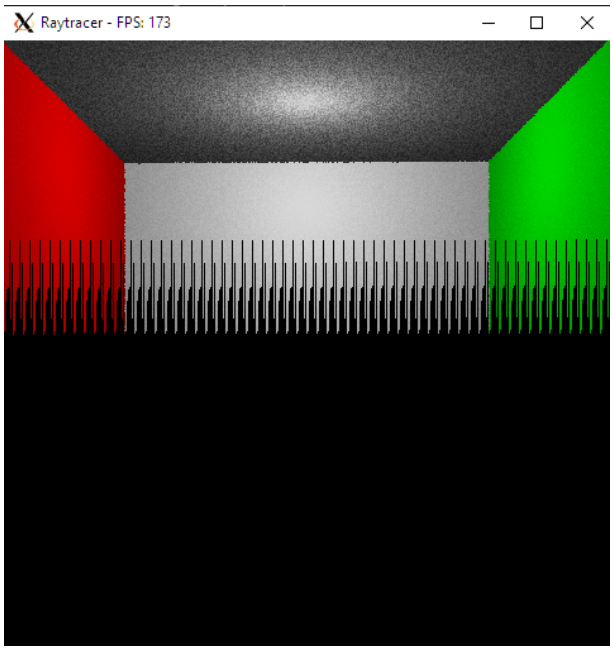

FIGURE 2.3 – Rendu en cours

Modèles et textures

Les textures

Dans la partie précédente j'ai déjà calculé les UVs pour la sphère et le carrée, il me rester donc à implémenter les UVs pour les maillages.

Dans un premier temps j'ai utilisé stb_image.h pour charger les textures. j'ai ensuite créer une classe pour plus facilement l'implémenter dans mon et pour plus facilement accéder au pixels.
J'ai 2 fonctions pour obtenir un pixel, une prenant des entier et l'autre des nombres flottant, celle qui prend des nombres flottant les ramène dans [0,1[et ensuite les ramène dans les dimensions du Tableau. La fonction fait ensuite appel à la fonction d'obtention de pixels prenant des entiers et renvoie un Vec3 de la couleur.

```
1  const Vec3 Texture::getPixel(const unsigned int x,const unsigned int y) const{
2      return Vec3((float)imageData[3*(w*y+x)]/255.0,(float)imageData[3*(w*y+x)+1]/255.0,
3                  (float)imageData[3*(w*y+x)+2]/255.0   );
4  }
5
6  const Vec3 Texture::getPixel( float x, float y) const{
7
8      return getPixel((unsigned int)floor(x*w),
9                      (unsigned int)floor((1-y)*h));//on inverse y car c'est Opengl
10 }
```

FIGURE 3.1 – 2 fonctions d'obtention des pixels

FIGURE 3.2 – Texture plaquée sur un carrée et sur une sphère

On peut bien voir que le mur du fond est texturé, on peut aussi voir une ébauche d'une normal map sur la sphère réfléchissante, cependant je l'avais implémenté sans faire de conversion entre l'espace tangent et l'espace monde donc j'ai abandonnée cette partie du code.

Les modèles

Après avoir réussi à charger des textures sur des formes simples, je me suis attaqué à chargé des textures sur des maillages, cependant les .off ne peuvent pas charger d'UVs à ma connaissance, donc j'ai utilisé assimp pour charger des modèles 3D dans la scène ainsi que leurs propriétés.
J'ai utilisé principalement des modèles gratuit en .gltf trouvé sur sketchfab, la source des modèles peut être trouvé dans le fichier licence.txt de chaque dossier. Je n'ai cependant pas implémenté tous les attribut disponibles en chargeant les modèles.

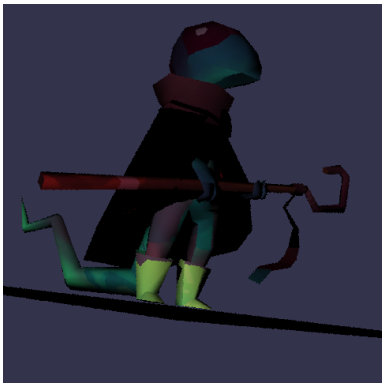

FIGURE 3.3 – Modèle de Reptile avec les coordonnées y inversées

FIGURE 3.4 – Modèle de Reptile avec les coordonnées correctes

À noté que par la suite une partie de ce modèle devient noir car ce modèle utilise une technique de rendu particulière, on peut résoudre le problème en augmentant la distance minimal de calcul d'intersection.

Lumière

Diffuse

J'ai voulu améliorer le rendu des matériaux diffus donc j'ai utilisé des Sources lumineuses orthotropes pour calculé la couleur à rajouter, je fais rebondir chaque rayon plusieurs fois et au dernier je renvoie juste le calcul de phong. sinon j'ajoute le calcul de phong et la lumière "diffuse" multiplié par Kd.

Refraction

J'ai implémenté la réfraction dans mon lanceur de rayon en m'inspirant du code de scratchapixel : https ://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-shading/reflection-refraction-fresnel
cependant j'ai fais en sorte que ce soit les rayons qui stockent l'indice de refraction actuel.

Caustiques

En plus d'implémenter la réfraction de la lumière directe, j'ai voulu implémenté le calcul de réfraction indirect lors du calcul de l'ombre, celui-ci ne créer pas directement de caustique mais avec le calcul des diffuses cela donne un résultat convenable.

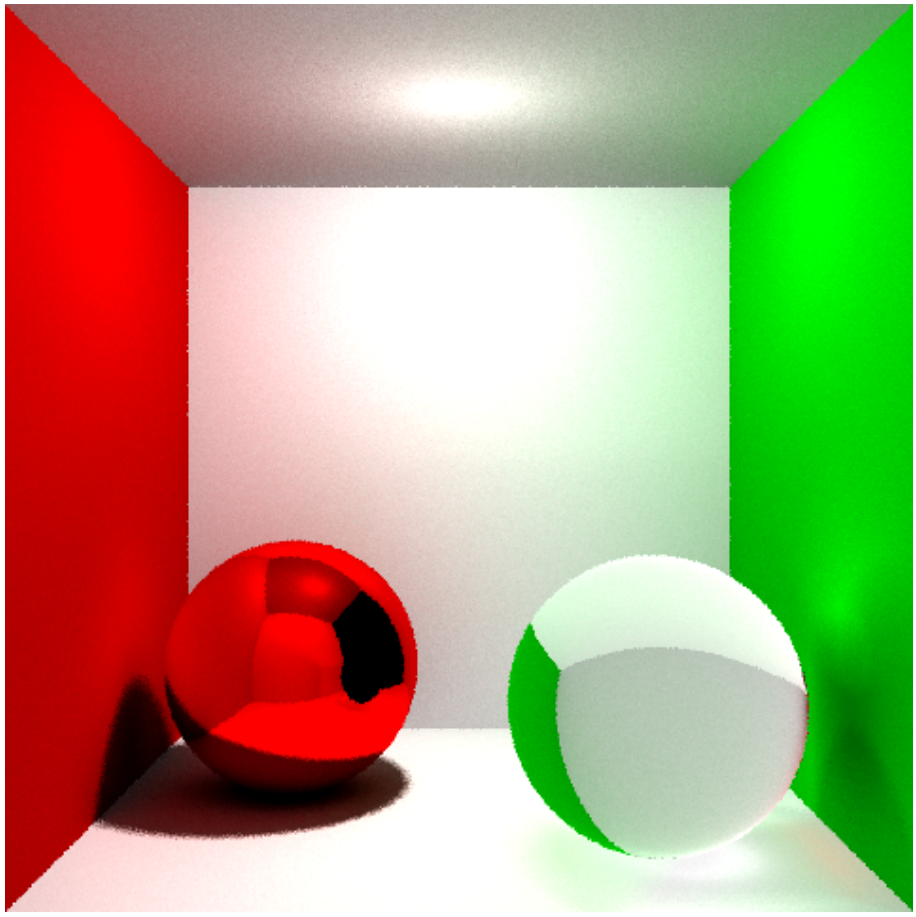

FIGURE 4.1 – Boîte de Cornell avec réfraction, couleur diffuses et Caustiques

Rétrospective

J'ai fais de mon mieux pour implémenter le plus de fonctionnalité possible tout en m'assurant que qu'elle soit les plus efficaces possible, pour beaucoup de fonctions dont le calcul de collisions avec un carré et une boite englobante, les fonctions que j'ai trouvé sur internet sont quasiment équivalente à mes fonctions pour des gros rendus.

J'aurais bien aimé implémenter les cartes de normales, cependant je n'ai pas trouvé de bonnes explications sur le passage de l'espace tangent vers l'espace monde.

Si vous voulez tester mon code, vous pouvez changer les valeurs dans les scène et dans le fichier Rendering.hpp, cependant il faut recompiler tout le code car c'est un header.

```
#include "Material.hpp"
#include "Ray.hpp"
#include "Scene.hpp"
#define LAMBDADIS 0.0005f
#define AMBIENTREF 0.4f
#define DIFFUSEREF 0.8f
#define SPECREF 0.05f
#define LIGHTRES 1
#define NBSAMPLES 1
#define MAXBOUNCE 1
#define MAX_DIFFUSE_BOUNCE 0
#define DIFFUSE_RAYS 128
```

Pour charger votre modèle rajouté juste au instruction de lancement la position du fichier.

Bonus

En bonus les 2 Bugs les plus drôles que j'ai rencontré.

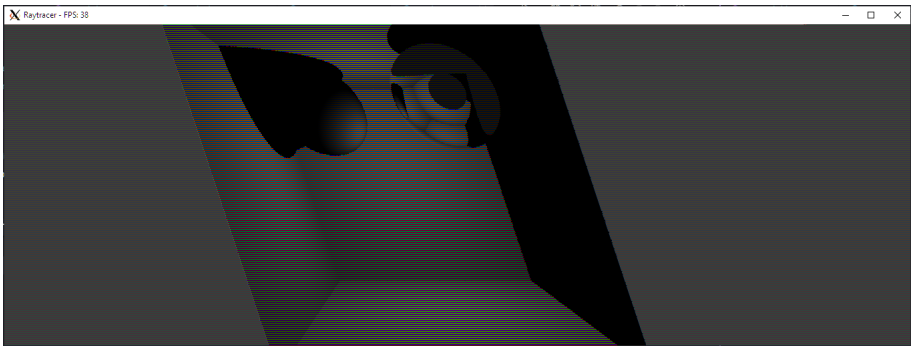

FIGURE 6.1 – Paranormal Activities

FIGURE 6.2 – Suzanne de cheshire