École Centrale de Lyon

Rapport du projet de RayTracing MOS 2.1

Étudiant :
Amr Miguil

Responsable :
M. Nicolas BONNEEL

31 mars 2019

Table des matières

1 Contexte de travail

Après une longue hésitation sur le langage de programmation à utiliser (C++ ou Java), j'ai finalement opté pour C++ en partie parce que Java ne gère pas les pointeurs. J'ai développé mon travail sur MacOS en utilisant **XCode** pour écrire du code, et **LLVM** de **Clang** pour compiler mes travaux. Au niveau des performances, j'ai un MacBook Pro avec un processeur i7, 16 GB de RAM, 2 cartes graphiques, 6 coeurs et 12 *threads*.

2 Les objets de la scène utilisés

J'ai utilisé et défini les classes *Sphere, Triangle* et *Geometry* qui héritent de l'interface *SceneObject*. Tous les attributs décrits dans *SceneObject* ne sont pas forcément tous utilisés par ces classes qui l'implémentent (exemple : A, B et C pour les sommets des triangles qui ne sont pas utilisés pour des sphères).

La méthode la plus intéressante dans chaque classe est la méthode abstraite *intersection* qui sera redéfinie pour chaque type.

2.1 Définition d'une sphère ou d'un triangle

J'ai implémenté 3 constructeurs différents au niveau de la classe abstraite *SceneObject*. Ainsi on doit préciser les points A, B et C dans le cas d'un *Triangle*, le centre et le rayon pour une *Sphere*. Les autres paramètres (*TypeMateriel*, *albedo* et *n* l'indice de réfraction) restent les mêmes pour les deux objets.

2.2 Chargement d'une géométrie

J'ai pris le code fourni dans le document *https ://pastebin.com/sm3z7ELy*, la version qui charge le maillage en inversant d'office les composantes y et z pour la fille *Beautiful Girl*. Je charge de plus les textures en utilisant la fonction *add_texture* fournie sur ce même site. La version vue en classe faisant intervenir *stbi_uc* ne fonctionne par car il y'a un problème au niveau des axes donc je ne l'ai pas utilisée.

On redimensionne le cas échéant notre géométrie en redimensionnant chaque élément de la liste des *vertices*. On la décale aussi l'objet si on le souhaite d'un certain vecteur *offset*.

2.2.1 Boite englobante

Le concept de boite englobante permet de gagner en temps d'exécution de manière considérable. On la construit en prenant la valeur maximale des *vertices* de chaque triangle que compose notre géométrie. Le but étant de voir si un rayon n'intersecte pas cette boite car il ne touchera pas notre géométrie.

2.2.2 Hiérarchie des boites englobantes

Le but de cet objet est de regrouper une multitude d'objets de type *BoiteEnglobante*. On y trouve deux attributs de type *HierarchieBoitesEnglobantes* : un droit et un gauche, un haut et un bas, un avant et un arrière selon la construction à l'issue de la méthode

construireHierarchie. Le mode de fonctionnement est décrit dans le code avec des commentaires, mais pour résumer je vais dire qu'on prend pour chaque sous structure de notre structure initiale (de BVH) la diagonale, puis on regarde la direction (x, y ou z) selon laquelle la projection de la diagonale est la plus grande . On divisera la BVH selon cette dimension. la figure suivante illustre mes propos :

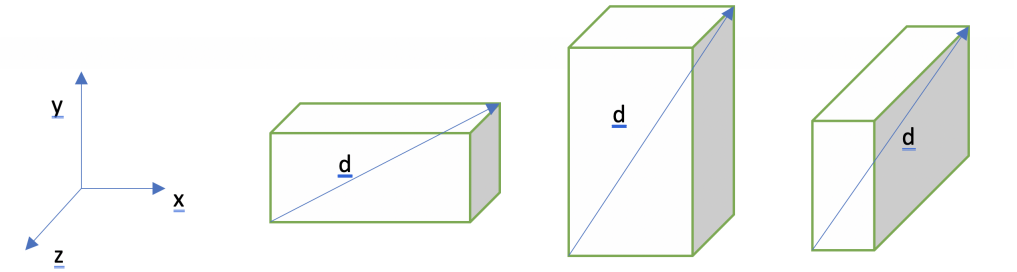


FIGURE 1 – 3 cas de hiérarchies boites englobantes. De gauche à droite, on divisera selon x, y ou z

Dans la pratique, la construction de la hiérarchie des boites englobantes se fait via un algorithme de *Quicksort* (voir le code pour les détails).

3 La méthode getColor

Cette méthode définit la couleur que va prendre un pixel de notre scène à partir des données de la scène. Elle renvoie un *Vector* des proportions des composantes *RGB* de ce pixel.

3.1 Définition des paramètres

On appelle la fonction qui détermine la couleur d'un pixel par :
- **rayon** : le rayon qu'on envoie, qu'on réfléchit, ou bien qu'on réfracte.
- **rebonNum** : Le critère d'arrêt de la fonction, utilisé par exemple pour fixer le nombre maximum de réfraction, de calcul de rayon indirect ...

3.2 Explication du fonctionnement

Pour chaque pixel de la scène, on envoie un rayon qui part de la caméra et atterrit sur la position de ce pixel. En fonction de la nature de l'objet intersecté, on va :
- **Objet de type *Emissif* :** Renvoyer la couleur de l'objet, qui sera aussi la couleur d'émission.
- **Objet de type Reflectif :** Relancer getColor en envoyant un rayon vers la direction donnée par *reflechir* de la classe Rayon
- **Objet de type *Refractif* :** Relancer getColor en envoyant un rayon vers la direction donnée par *refracter* de Rayon.
- **Objet de type *Fresnel* :** Relancer getColor en envoyant un rayon vers la direction donnée par *fresneliser* de Rayon.

— **Objet de type *Diffus* :** Continuer le traitement de getColor.

4 Principe général d'une intersection

On commence par appeler la routine d'intersection de l'objet *scene* de type *Scene*. Sur cette méthode on fait une boucle sur tous les éléments qui s'y trouvent et en fonction de la distance d'intersection on va renvoyer l'objet intersecté.

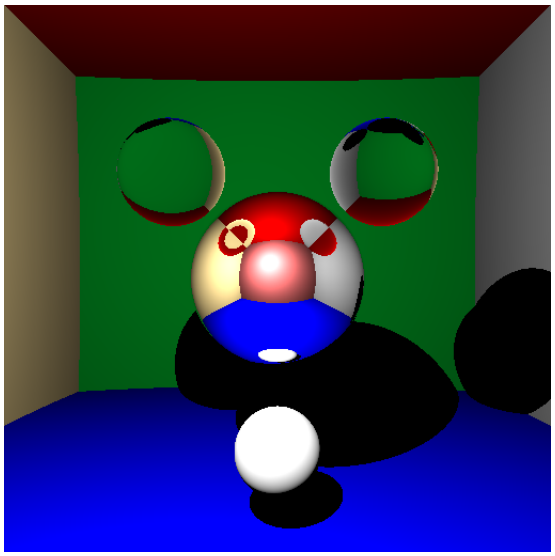
Dans mon code, les paramètres de la fonction qui renvoie un *booléen* s'il y'a intersection sont :

— **rayon** de type *Rayon*.
— **n** : un vecteur qui définit la normale à l'objet au point intersecté.
— **t** : la distance depuis l'origine du rayon et l'objet intersecté au point d'intersection.
— **alphaBetaGamma** : un vecteur qui définit les coordonnées barycentriques dans le cas d'une intersection avec un *Triangle*.
— **textures** : un vecteur qui prend les composantes RGB d'une texture dans le cas d'une *Geometry*.

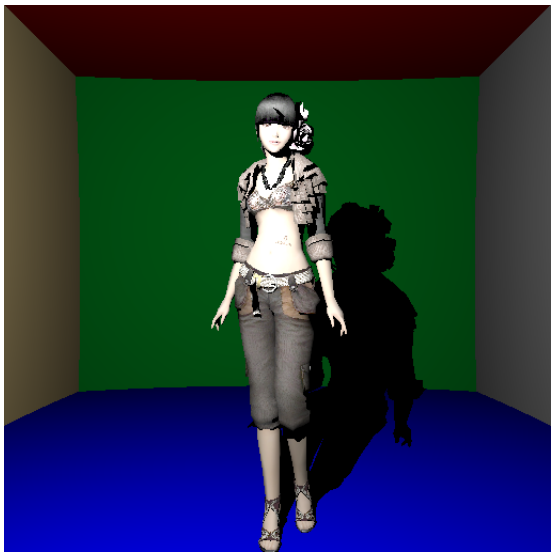
Les vecteurs alphaBetaGamma et textures interviennent pour la géométrie. En effet, lors d'une projection 3D vers un plan en 2D, ce qui est le cas pour la géométrie, on prend depuis le vecteur *uvs* les composantes x, y et z du triangle qui a été touché. Ensuite, le code va colorier le point défini à partir des coordonnées barycentriques α, β et γ que la routine d'intersection avec un Triangle fournit et que l'on multiplie par les longueurs et largeurs de chaque texture. Les composantes RGB de la couleur sont ainsi obtenues de la même manière que lors de l'export de l'image. On renvoie dans le cas de cette intersection un vecteur *Vector(R,G,B)* au lieu de *Vector(-1,-1,-1)*

5 Eclairage direct

Dans ce type d'éclairage, on va envoyer un seul rayon vers la scène et on prend la couleur du pixel intersecté, éventuellement après un certain nombre de réflexions ou de réfractions.



Rendu avec au centre une sphère diffuse, une réflective et 2 réfractives en éclairage direct (0.21 seconde)



Rendu avec une géométrie en éclairage direct (0.44 secondes)

6 Eclairage indirect

6.1 Rappel de la méthode de Monte-Carlo

Cette méthode permet de faire la moyenne de tous les éclairages issus des différents rayons qui seront lancés sur la scène. Cette méthode stipule que :

$$\int f(i,o) \times L(x,i) \times cos(\theta_i) \times di = \frac{1}{n} \times \sum_{j=1}^{n} f(i_j,o) \times L(x,i_j) \times \frac{cos(\theta_i)}{p(i_j)}$$

Autrement dit que chaque rayon va être renvoyé vers une direction aléatoire autour de la normale de l'objet intersecté et ira chercher la contribution de la lumière issue de la deuxième, 3^{me}, ... $rebonNum^{me}$ intersection. D'un point de vue pratique, j'ai choisi la distribution aléatoire donnée par la fonction *randomCos* vue en cours. Le principe est d'envoyer un rayon totalement aléatoirement dans une direction qui est la somme de deux vecteurs tangents à la surface et d'un vecteur normal :

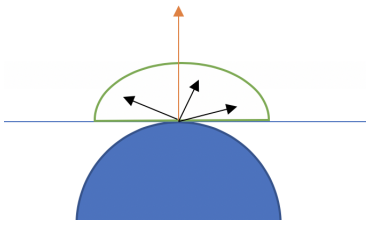


FIGURE 2 – On envoie autour du vecteur orange des vecteurs noirs pointant vers la scène

6.2 Application à une sphère "Fresnel" - Anti-aliasing

J'ai volontairement choisi de ne pas implémenter l'anti-aliasing pour un éclairage direct et de l'implémenter pour un éclairage indirect seulement. Les sphères obtenues à la section éclairage direct ont des irrégularités en leurs bords, c'est encore plus visible avec les objets triangles et géométrie. L'image suivante qui comporte le dernier type de sphère (Fresnel) avec un indice de réfraction de 1.4 a été générée avec l'option *anti-aliasing* telle que définie dans le cours en utilisant la méthode de **Box-Muller**. Notons que la nature réfractive ou réflective d'un pixel d'un objet Fresnel va dépendre du coefficient de réflexion et donc de l'indice de réfraction (voir le code)

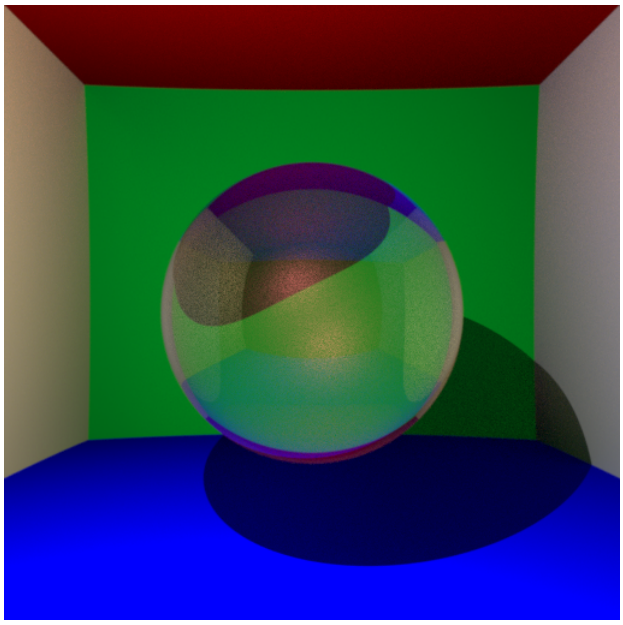


FIGURE 3 – Sphère de *TypeMateriel* Fresnel, généré avec 500 rayons indirects, 137 secondes

7 Source de lumière sphérique et ombres douces

Pour cet éclairage, il est préférable d'utiliser des rayons indirects, sinon faire tendre le rayon de la source de lumière vers 0. L'algorithme de détermination de la couleur d'un pixel est :

- Lancer un rayon vers la scène
- S'il intersecte la sphère lumière, donner la couleur d'émission de la lumière
- Sinon adopter le comportement défini pour chaque type d'objet s'il ne s'agit pas d'un objet Diffus.
- Pour un objet Diffus, j'ai implémenté la méthode vue en séance qui consiste à faire un changement de variable lors de l'intégration de Monte-Carlo afin de raisonner non pas en angles mais en surfaces entre la sphère lumière et l'objet touché :

$$direct = \frac{I}{4\pi^2 \times R^2} \times \frac{albedo}{\pi} \frac{< w_i, N >< xx^{'}, N > V(x, x^{'})}{d(x, x^{'})^2 \frac{<Ox, Ox^{'}>}{\pi}}$$

Avec $x^{'}$ un point de la sphère lumineuse que le rayon intersecte en second lieu. V est

un terme de visibilité, qu'on multiplie par le produit scalaire entre $xx^{'}$ et N et qu'on divise par la distance au carré entre x et $x^{'}$ pour avoir le jacobien du changement de variable. En toute rigueur, on adoptera toujours $std :: max(0, dot(w_i, N))$ pour dire que la source de lumière et la normale de l'objet au point d'intersection sont en regard l'une de l'autre.

J'ai implémenté cela sur une scène où j'ai mis tous les types d'objets que j'ai définis, les triangles étant implicitement présents dans la géométrie :

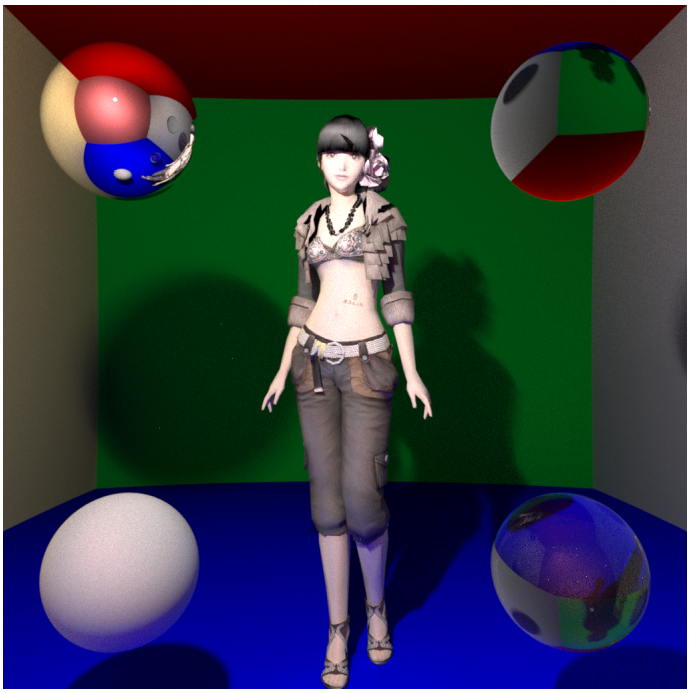


FIGURE 4 – Rendu avec 200 rayons ppx, sphère lumière avec rayon = 1, résolution 1024×1024 (912 secondes)

Notons que les ombres sont moins agressives et que la qualité de l'image croît avec la résolution et le bruit diminue avec le nombre de rayons par pixel.

8 Rotation autour d'une géométrie

Pour simuler une rotation, nous allons utiliser une matrice de rotation. La méthode *getRotation* permet de tourner selon l'axe **axis**, un Vector **vecteur** selon un angle **angleRotation**. Ainsi, pour faire tourner la caméra seule autour de la scène, on va faire tourner sa position ainsi que la direction du *Rayon* qui ira vers la scène. On pourra faire tourner la source de lumière, la position d'un objet ou bien pourquoi pas un objet autour de lui-même mais je n'ai considéré qu'une rotation autour de la position de l'axe *axis* = 0. Un des gifs joints au travail (*7603s100rpp1lum.gif*) a été généré avec 100 rayons par pixel, une source lumineuse sphérique de rayon 1, est constitué de 50 images générées avec une rotation uniforme de pas $\frac{\pi}{50}$ et a été obtenu après 7600 secondes.
Pour créer un gif à partir des images obtenues, j'ai utilisé la librairie **Magick++** en précisant le nombre d'images à joindre, la durée pour chaque image et le chemin vers le dossier d'entrée. (voir Annexe)

9 Impressions et conclusion

J'ai trouvé ce projet très intéressant au sens où j'ai pu apprendre un nouveau langage de programmation. J'ai trouvé plusieurs difficultés lors de l'implémentation du code. J'ai dû refaire 3 fois le code depuis le début par exemple, ou encore désactiver *pragma omp parallel* pour déboguer pixel par pixel. La partie qui m'a le plus fatigué est l'implémentation d'une géométrie car je n'arrivais pas à visualiser les concepts qui facilitaient les calculs d'intersection. J'ai aussi travaillé sur *Visual Studio*, *Visual Studio Code* et *Eclipse* au début mais le débogage ainsi que la mise en oeuvre de openMP y étaient très difficiles. J'ai utilisé XCode à la fin et cet éditeur m'a beaucoup facilité la tâche car j'ai déjà fait du Swift dessus et l'IHM est très *user-friendly*.
Au final malgré tout le temps nécessaire pour appréhender le sujet, le résultat obtenu était à la hauteur et j'ai pu comprendre plusieurs concepts en Ray tracing, à la fois pour les aspects pratiques que pour les méthodes utilisées.

10 Annexe

Pour lancer le code C++, je n'ai pas pu tout mettre en forme pour lancer le raytracer depuis un terminal. Il faut changer les valeurs de :

- **pathOut** à changer pour le path où exporter la scène. (Ou vers le dossier pour une animation). Exemple : "/Users/amrmiguil/Desktop/imageeOut.png"
- **pathGirl** à changer pour le path à se trouve le fichier .obj de la fille. Exemple : "/Users/amrmiguil/Desktop/beautifulGirl.obj"
- **pathToTexturesFolder** le chemin vers le dossier où se trouvent les textures. Exemple "/Users/amrmiguil/Desktop/girlOK/"
- **useBVH** pour utiliser ou pas une BVH pour la géométrie
- **antiAliasing** pour activer ou pas l'anti-aliasing
- **indirectColor** pour activer les couleurs indirectes
- **sourceDeLumiereSpherique** pour une source de lumière sphérique. Vous pouvez changer le rayon tout en bas dans le main, *sphereLum*.
- **animation** pour exporter une animation. Vous pouvez changer le pas et l'axe de rotation dans la fonction *generateRotation*.

Lien du GitHub privé : https ://github.com/aMiguil/RayTracer/invitations

Code Magick++ pour générer un gif :

```
1
2  #include <iostream>
3  #include <Magick++.h>
4  #include <thread>
5
6  int counter = 50;
7  void exportGif(void){
8      Magick::InitializeMagick("");
9
10     std::list<Magick::Image> frames;
11
12     for(int i = 0; i < counter ; i++){
13         std::string location = "/Users/amrmiguil/Desktop/ToBeGiffed/image"
               + std::to_string(i) + ".jpg";
14         Magick::Image img = Magick::Image(location.c_str());
15         img.animationDelay(20);
16         frames.push_back(img);
17     }
18     Magick::writeImages(frames.begin(), frames.end(), "/Users/amrmiguil/
           Desktop/ToBeGiffed/giffedImages.gif");
19     return;
20 }
21
22 int main(int argc, const char * argv[]) {
23     std::thread tr(exportGif);
24     tr.join();
25     return 0;
26 }
```