Raytracing in One Weekend
Rapport

Alexis Dupuis and Loïc Hentz

February 2019

Table des matières

Github's link :

`https://github.com/alexis-dupuis/CONAV_Raycasting`

1 Introduction

Les tutoriels sont toujours un exercice complexe. S'ils sont courts, ils passeront probablement sous silence certains points et risquent de créer des blocages et une perte de temps conséquente ; s'ils sont trop longs ils risquent de s'éparpiller dans les détails ou de repartir trop aux bases, perdant notre attention avant qu'on n'ait pu observer des résultats intéressants et satisfaisants qui donnent envie de continuer. Le risque aussi est le côté "dicté" où le tutoriel va consister uniquement à recopier des pavés de code sans forcément réussir à nous donner envie de comprendre ce qu'ils contiennent.

Deux éléments me semblent un peu embêtant à première vue : le tutoriel manque d'une petite introduction pour présenter quelle va être son organisation (aurais-je un résultat intéressant au bout de 5 pages ? 25 pages ? 25h ?). Enfin, le code à comprendre et recopier est fourni en images. On est donc plutôt invité à le comprendre en le recopiant, ce qui peut être fastidieux. Il est possible de télécharger les sources mais la plupart des fonctions sont amenées à évoluer au fur et à mesure du tutoriel, et disposer de la version finale n'aide pas nécessairement comprendre la version première de cette fonction.

2 Raytracing in One Weekend : avancement et difficultés.

Chapitre 1 : une première image.

Ce chapitre, comme les autres d'ailleurs, propose un main en C++ à recopier. La seule "difficultée" jusqu'ici sera donc de compiler le projet. Il permet un rappel pas forcément utile sur le fait qu'une image est composée de pixels dont la couleur peut être codée en RGB. Le format d'image choisit (.ppm) est un peu embêtant car difficile à ouvrir. Nous avons utilisé Gimp sur Linux.

Une chose étrange est l'utilisation de "*cout*" là où on attend plutôt une écriture dans un fichier. Nous avons dû le remplacer par :

outfile.open(my/Path/myimage.ppm).

Cela étant dit ce chapitre simple permet d'obtenir un premier résultat. Une fois l'étape toujours ennuyeuse de préparer et compiler le projet une première fois, cela permet de mettre le pied à l'étrier assez rapidement.

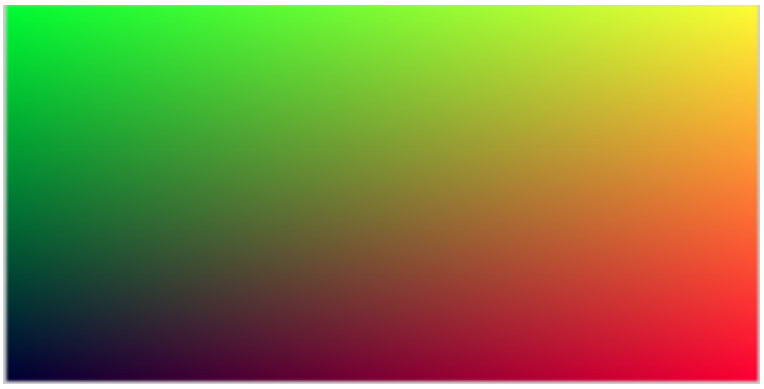

FIGURE 1 – Une première image.

Chapitre 2 : une classe vec3.

Je trouve l'idée de revenir sur la classe vec3 plutôt intéressante. Même si on perd un peu de temps à refaire quelque chose de basique, je trouve qu'il n'est pas inutile d'être forcé à bien connaitre cet élément qui va être manipulé amplement dans la suite. Le risque d'utiliser des bibliothèques toutes faites comme Eigen est d'être confronté à des structures plus complexes (car plus versatiles), et dont on ne va pas forcément avoir envie d'aller lire le code ou la documentation en détail.

Il y a en revanche énormément de code pas forcément intéressant à recopier. On se doute de ce que doit faire les différentes fonctions, pas besoin de les réécrire pour bien les comprendre. Heureusement ces fichiers-ci n'évolueront pas donc on peut utiliser les sources fournies pour éviter de les recopier à la main.

Chapitre 3 : classe ray.

Ce chapitre très court introduit la classe *ray* qui représente le rayon à lancer. Encore une fois pas grand chose à faire à part recopier la classe et plisser les yeux sur la fonction *main()* pour essayer de trouver quelles lignes ont été changées. On nous fait ici afficher un lerp bleu/blanc qui à mon avis aurait gagné à être créé immédiatement plutôt que d'afficher d'abord le dégradé en Chapitre 1 puis le remplacer par ce fond bleu/blanc juste après.

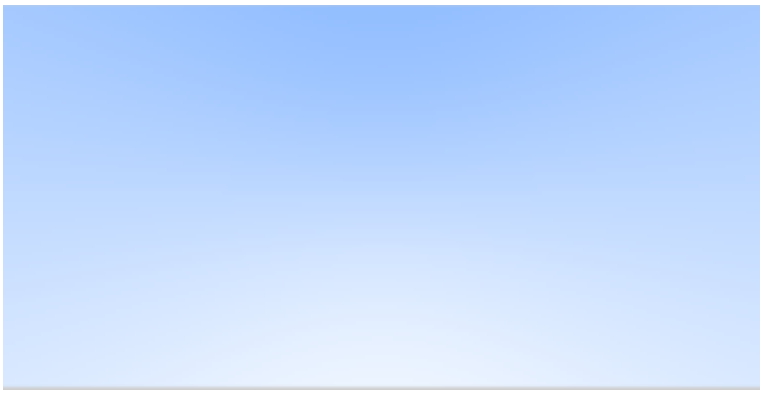

FIGURE 2 – Notre fond : un lerp bleu.

Chapitre 4 : ajout d'une sphère.

Ici, on détaille les formules simples permettant de savoir quel(s) point(s) d'un rayon touche potentiellement une sphère de rayon r posée dans la scène. Le chapitre est assez simple puisqu'il consiste simplement en l'ajout d'une fonction *hit_sphere()* prenant en argument les paramètres d'une sphère et un rayon à tester, et renvoit un booléen.

FIGURE 3 – Une première sphère.

Chapitre 5-1 : ajout des normales pour l'ombrage.

Dans un premier temps, on nous propose de modifier la fonction *hit_sphere()* pour qu'elle renvoit le point de contact rayon/sphère le plus proche de la caméra s'il existe. Ensuite, on décide de visualiser dans un premier temps les normales en les coloriant (chaque coordonnée est remappée entre [0,1] et utilisée comme paramètres RGB). Cette petite astuce permet une visualisation rapide du résultat, ce qui est plutôt une bonne idée pour confirmer que tout fonctionne bien.

FIGURE 4 – Une sphère avec des normales.

Chapitre 5-2 : plusieurs objets "hitable".

Afficher une sphère est intéressant, mais comment en afficher plusieurs ? Comme cette question va soulever des problèmes d'occlusion et d'ombrages entre autres, on nous propose de refaire les choses proprement en déclarant une classe abstraite "hitable" dont "sphere" hérite. La fonction *hit_sphere()* est donc abandonnée et remplacée par *sphere : :hit()*.

On écrit également une classe hitable_list dont la fonction *hit()* teste la collision avec le rayon pour chaque élément de la liste et renvoit le point de collision le plus proche de la caméra.

Malheureusement ces classes plutôt simples ne sont pas récupérables telles-qu'elles dans les sources, puisque *sphere* sera modifiée par la suite pour ajouter des matériaux. L'adaptation du *main* est également assez fastidieuse car il faut à nouveau chercher les modifications sur l'image pas très lisible... Enfin, *MAXFLOAT* n'est pas reconnu dans *float.h*. Une recherche Google propose de corriger ça en utilisant *FLT_MAX* à la place.

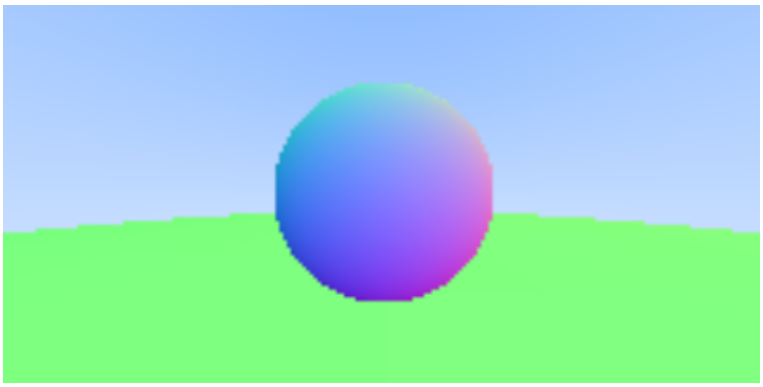

FIGURE 5 – Une seconde sphère (verte et de grande taille).

Chapitre 6 : antialiasing.

La discrétisation de l'image due aux pixels crée un effet "escalier" d'aliasing dont on veut se débarasser. Cela arrive lorsqu'un pixel est situé sur un bord, et devrait prendre 2 couleurs différentes. On va donc simuler cette précision "sub-pixel" en affichant dans ces pixels un mélange des différentes couleurs qu'il devrait contenir. Pour cela, on défni plusieurs "échantillons" (des points à différents endroits du pixel) où faire passer un rayon, et on moyennera les résultats des rayons sur ce pixel.

Ces échantillons sont tirés aléatoirement sur le pixel. On définit également ici une classe caméra qui permettra de changer l'angle de vue proprement en changeant ses vecteurs.

A nouveau, la classe caméra doit-être réécrite car sa version finale est relativement différente et incompatible avec le code à cette étape.

FIGURE 6 – L'effet de l'antialiasing : plus d'escaliers aux contours des formes !

Chapitre 7 : materiaux mats.

Afin de modéliser le comportement d'un "diffuse material", on veut faire rebondir les rayons dans des directions aléatoires à la surface de nos sphères (sans se préocuper de la couleur du materiau pour l'instant). On va donc tirer un vecteur aléatoirement dans une sphère unitaire tangente au point de contact de notre rayon, qui servira de direction de reflexion aléatoire.

Les explications et le code sont relativement clairs sur le début. Je n'ai en revanche pas bien compris l'explication sur le "gamma 2" qui consiste à eclaircir l'image en prenant les racines des composantes de notre couleur plutôt que la couleur elle-même. Je ne comprend pas non plus ce qui cause le problème du "shadow acne" tel que c'est expliqué ici...

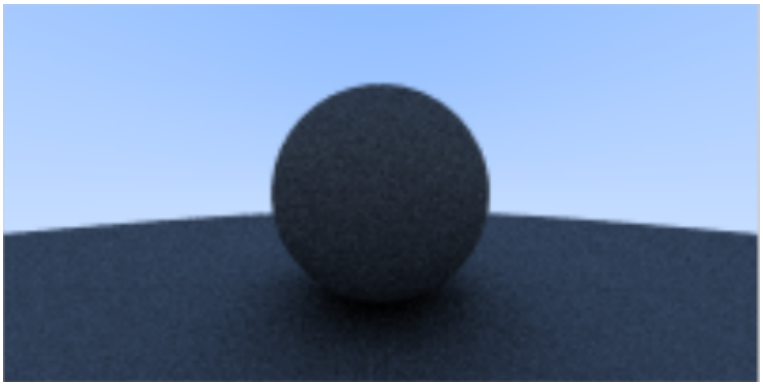

FIGURE 7 – Nos sphères avec des matériaux mats incolores : sans correction gamma2.

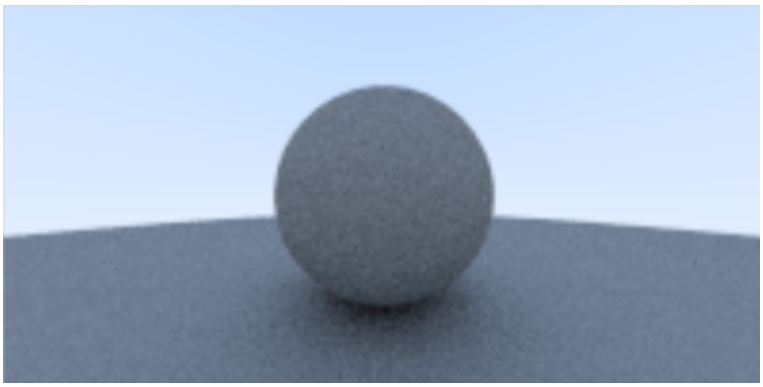

FIGURE 8 – Nos sphères avec des matériaux mats incolores : avec correction gamma2.

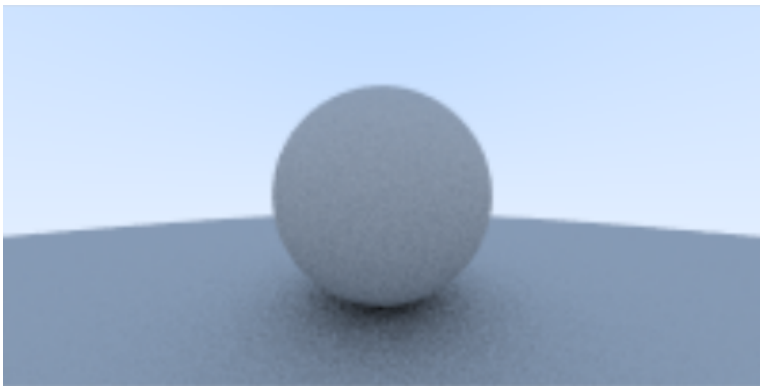

FIGURE 9 – Élimination du bug causant la “shadow acne”.

Chapitre 8 : couleur et matériaux métalliques.

Ce chapitre étend la classe des matériaux aux matériaux métalliques. On a donc désormais un fichier *material.h* contenant une classe abstraite : le matériau peut-être lambertien (mat) ou metallique pour l'instant. La nouveauté principale est la prise en compte des rayons réfléchis ou absorbés. On choisit de représenter les matériaux mats soit en absorbant une proportion p aléatoire des rayons, soit en n'en absorbant aucun mais en introduisant une atténuation sur ceux réfléchis. Pour chaque rayon, une fonction scatter renvoit un booléen *true* si le rayon a rebondi, *false* s'il est absorbé (auquel cas on renvoit du noir dans la fonction *color()*).

Pour les rayons réfléchis, l'atténuation s'exprime comme une multiplication vectorielle terme à terme sur la couleur renvoyée (ce qui représente la ”couleur naturelle” du matériaux, c'est à dire sa couleur sous une lumière blanche). Un rayon réfléchi est bien sûr passé récursivement à la fonction couleur pour qu'on puisse suivre le reste de son chemin. Pour éviter des réfléxions à l'infini, on défini une ”profondeur”, c'est-à-dire un nombre de rebonds maximaux autorisés au delà duquel on renverra simplement du noir en considérant ce rayon complétement atténué par tous ses rebonds.

Cette section est assez directe mais introduit tout de même plusieurs concepts : l'albedo (couleur naturelle), la réfléxion, la profondeur. J'ai trouvé la présentation un peu confuse, et j'ai plutôt déchiffré le code pour comprendre. Ici aussi, quelques petits changements futurs font perdre du temps si on tente de réutiliser *material.h* depuis les sources fournies.

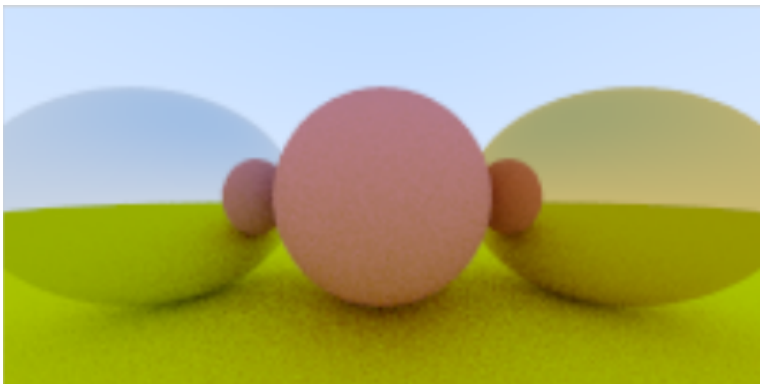

FIGURE 10 – Matériaux mats colorés et métalliques.

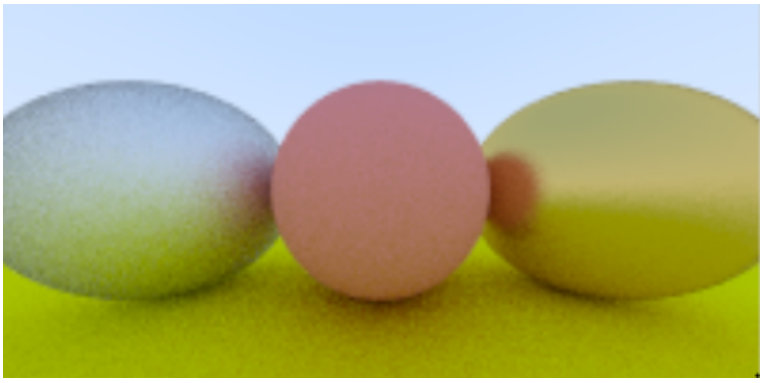

FIGURE 11 – Matériaux mats colorés et métalliques flous.

Chapitre 9 : matériaux transparents.

Ce chapitre traite des matériaux transparents via la loi de Snell-Descartes. Il reprend la class abstraite *material* précédente et y ajoute une nouvelle classe *dielectric* en introduisant la notion de refraction.

Je pense que ce chapitre mérite plus que tout autre une décomposition du code présenté car il n'est vraiment pas possible de comprendre correctement ce qui est fait ici sans reprendre tout le code présenté soi-même. Les explications donnent des rappels sur l'aspect physique des propriétés codées mais pas vraiment sur leur implémentation.

On nous propose de tracer des matériaux transparents et également une astuce permettant de simuler un matériau transparent creux (sphère creuse en l'occurence).

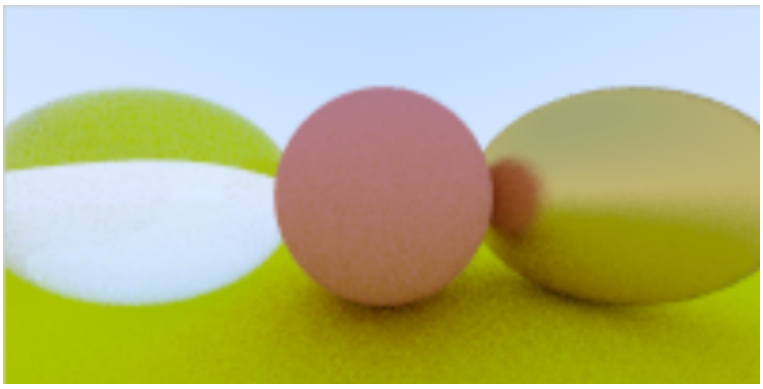

FIGURE 12 – Une sphère avec matériau transparent.

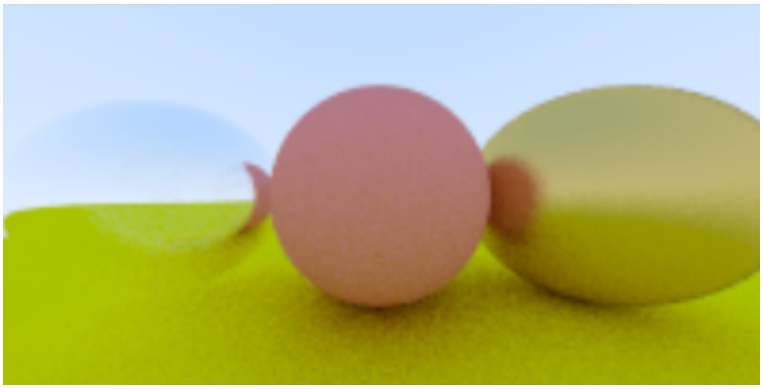

FIGURE 13 – Une sphère creuse avec matériau transparent.

Chapitre 10.1 : positionnement de la caméra : fov.

Ce chapitre propose d'apporter des modifications à la classe *camera* afin de pouvoir modifier la prise de vues. Le début est un peu léger en explications puisqu'il ne présente pas certaines variables du code tel que "aspect", un paramètre correspondant au ratio hauteur/largeur de la vue.

Dans un premier temps, on reste sur une caméra qui regarde droit devant. On détermine simplement les dimensions de l'écran en fonction du champ de vue vertical en degrés. On construit donc un constructeur surchargé de camera qui prend en paramètre l'aspect et le fov vertical. Dans notre cas l'aspect correspond au ratio de pixels qu'on désire pour notre image finale, donc nx/ny (où les n sont le nombre de pixels de l'image).

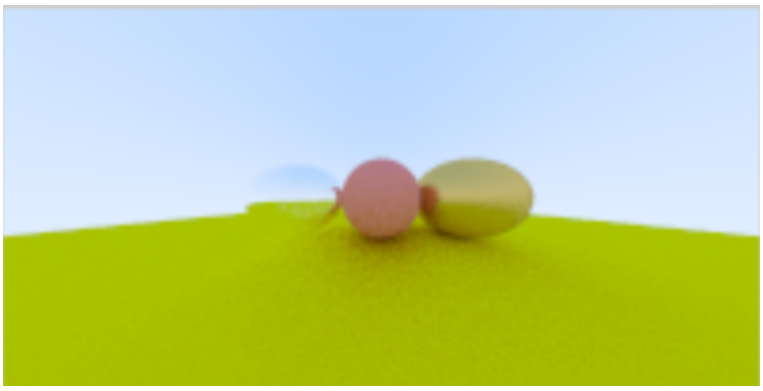

FIGURE 14 – Rendu avec un FOV vertical de 140 degrés.

FIGURE 15 – Rendu avec un FOV vertical de 45 degrés.

Chapitre 10.2 : positionnement de la caméra : direction du regard.

On poursuit ce chapitre en permettant de changer le placement de la caméra, ce qui revient à définir un nouveau repère caméra. On commence en définissant le vecteur du regard ($lookat - lookfrom$). On souhaite également pouvoir définir le roulis de la caméra autour de cet axe de regard, donc on passe un vecteur "up" en paramètre. La projection orthogonale de up dans le plan normal au vecteur regard nous donne le second vecteur du nouveau repère caméra. Le dernier vecteur est bien sûr obtenu par un produit vectoriel.

FIGURE 16 – Rendu avec un FOV vertical de 30 degrés et une caméra en hauteur regardant vers le bas.

Chapitre 11 : simulation de la profondeur de champ et effet de flou.

Ce chapitre nous permet de coder un effet de flou esthétique et la profondeur de champ (profondeur où on souhaiteras garder l'image nette). Il commence par un rappel du principe dans les cameras réelles, qui présentent la différence notable qu'une réduction de l'ouverture provoque une baisse de luminosité (contrairement à notre rendu informatique).

Concrètement, ce qu'on est en train de faire est de remplacer le *pinhole camera model* par une lentille simulée. On va donc faire partir nos rayons de la surface d'une lentille, ce qu'on simule par un tirage aléatoire sur un disque. Ce tirage donne un offset à prendre en compte dans la fonction *getray()*. L'ouverture est contrôlée en changeant le rayon de cette lentille virtuelle. Le point de départ du rayon est sur la lentille et non plus sur l'écran directement : il faut représenter le focus du rayon de la lentille au plan image (écran) avec un facteur *focus-dist* à appliquer sur les vecteurs définissant l'écran.

Je ne suis pas certain que ce chapitre soit indispensable pour comprendre le raycasting, mais il permet tout de même une réflexion sur les différents éléments d'une caméra réelle par rapport au pinhole model, ce qui est plutôt intéressant.

FIGURE 17 – Rendu avec un FOV vertical de 30 degrés et une caméra en hauteur regardant vers le bas.

Chapitre 12 : une belle image.

On utilise dans ce chapitre une fonction random-scene() qui permet de faire apparaître de nombreuses sphères dans notre monde. On augmente également la taille de l'image. On réduit le nombre de samples pour éviter un temps de calcul trop long, et on réarrange la caméra à notre aise.

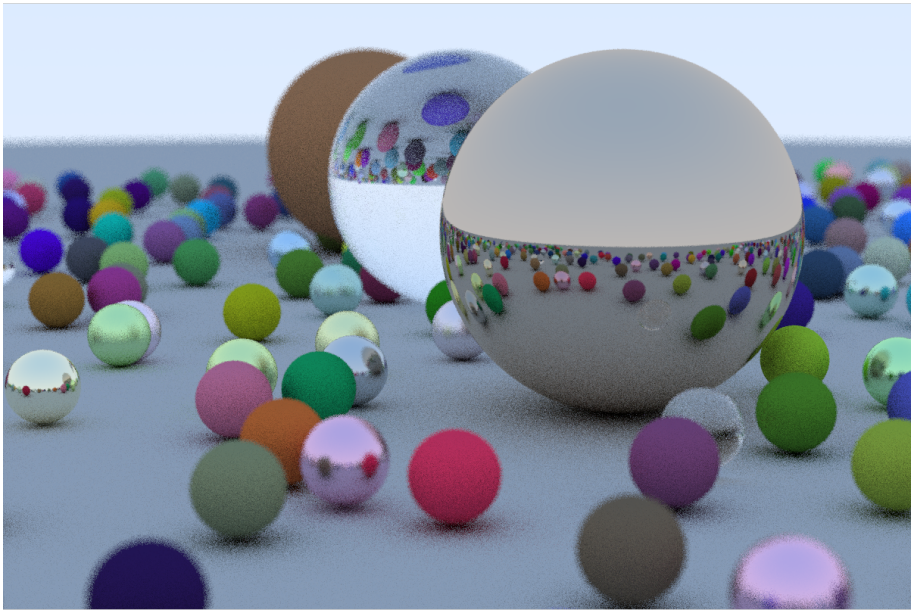

FIGURE 18 – Un exemple de rendu plus complexe.

3 Propositions d'améliorations.

Longueur

Le tutoriel a l'avantage d'avancer à un bon rythme avec des explications somme toutes claires. Il peut en revanche parfois paraître un peu long au début lorsqu'on fait quelques détours (affichage des normales en couleur par exemple, ou encore la fonction *hit_sphere()* remplacée par *sphere : :hit()*). Cela offre tout de même de bons moyens de confirmer le fonctionnement du code en assez peu d'efforts au fur et à mesure. Il est également plaisant d'avoir un résultat affiché à chaque étape et de voir l'évolution du projet.

Cela étant, je pense que le grand nombre de projets proposés en CONAV reduit le temps qu'il est possible de consacrer à un tutoriel comme celui-ci sur un thème aussi spécifique que le ray-tracing. Comme le début est relativement simple, je pense qu'il serait possible de fusionner quasiment les 4 ou 5 premiers chapitres afin de gagner du temps et d'aller plus rapidement dans le vif du sujet. La caméra aurait aussi pu etre définie depuis le début par exemple à mon avis.

Rythme

Le problème principal de ce tutoriel est le besoin de recopier le code. Il faut bien sûr en arriver là à un moment donné avec tout tutoriel, mais je pense que le plus important est que le lecteur ait bien compris les différentes parties du code plutôt que le fait qu'il les ait tapées lui-même.

Le tutoriel a l'avantage de séparer les differentes classes en fichiers, ce qui permet parfois d'éviter de recopier des classes comme hitable ou vec3 dont on comprend très facilement le code de toute façon. Malheureusement il n'est pas toujours possible de procéder ainsi avec certaines classes, ce qui fait perdre encore une fois du temps de manière un peu inutile.

Il serait possible d'écrire plus rapidement un *main.cpp* plus conséquent (comme proposé dans la section précédente), et de pousser plus loin certaines classes immédiatement. Pour cela, il faudrait en revanche décomposer le code et l'expliquer par petits bouts, plutôt que d'expliquer tout le concept et d'envoyer tout le code en un morceau à la fin comme cela est trop souvent fait. L'image 19 montre ma fonction *main()* commentée par chapitres, sachant que j'ai fait en sorte de ne pas commenter les parties qui étaient réutilisées plus tard. Le résultat montre que le code est très peu réutilisé, et on perd à mon avis plus de temps à faire beaucoup de petites étapes qu'on ne réutilisent pas plutôt que de prendre le temps d'expliquer immédiatement du code utile.

Bien sûr il serait également intéressant d'avoir le code copiable plutôt que sur des images, étant donné qu'on perd souvent du temps à recopier du code qu'on a déjà compris de toute façon. Proposer des sources réutilisables plus

régulièrement permettrait également d'éviter une partie du recopiage manuel des classes. Il serait également possible d'aérer et commenter le code présenté car il est souvent désagréable de devoir trouver et lire les lignes à changer dans le main. Alternativement, surligner les lignes à changer dans le main par exemple aiderait à la lecture.

```
    /* //Camera vectors //Chapters < 5
    vec3 lower_left_corner( -2.0, -1.0, -1.0 );
    vec3 horizontal( 4.0, 0.0, 0.0 );
    vec3 vertical( 0.0, 2.0, 0.0);
    vec3 origin( 0.0, 0.0, 0.0);*/

    /* //Chapter 5+ //Hitables in the world
    hitable *list[4];
    list[0] = new sphere(vec3(0,0,-1), 0.5, new lambertian(vec3(0.8, 0.3, 0.3)) );
    list[1] = new sphere(vec3(0,-100.5,-1), 100, new lambertian(vec3(0.8, 0.8, 0.0)) );
    list[2] = new sphere(vec3(1,0,-1), 0.5, new metal(vec3(0.8, 0.6, 0.2), 0.3) );
    //list[3] = new sphere(vec3(-1,0,-1), 0.5, new metal(vec3(0.8, 0.8, 0.8), 1.0) );
    list[3] = new sphere(vec3(-1,0,-1), -0.45, new dielectric(1.5) );//radius < 0 = hollow sphere

    hitable *world = new hitable_list(list,4); //4 spheres...
    */

    //Chapter 12: a whole new world...
    hitable *list[5];
    float R = cos(M_PI/4);
    list[0] = new sphere(vec3(0,0,-1), 0.5, new lambertian(vec3(0.1, 0.2, 0.5)));
    list[1] = new sphere(vec3(0,-100.5,-1), 100, new lambertian(vec3(0.8, 0.8, 0.0)));
    list[2] = new sphere(vec3(1,0,-1), 0.5, new metal(vec3(0.8, 0.6, 0.2), 0.0));
    list[3] = new sphere(vec3(-1,0,-1), 0.5, new dielectric(1.5));
    list[4] = new sphere(vec3(-1,0,-1), -0.45, new dielectric(1.5));
    hitable *world = new hitable_list(list,5);
    world = random_scene();

    //Chapter 6+ //Defining a camera
    //camera cam; //Chapter 6-9
    //camera cam(140, float(nx)/float(ny)); //Chapter10.1
    //camera cam(vec3(-2,2,1), vec3(0,0,-1), vec3(0,1,0), 30, float(nx)/float(ny)); //Chapter 10.2
    //camera cam(vec3(3,3,2), vec3(0,0,-1), vec3(0,1,0), 20, float(nx)/float(ny), 2.0, vec3(3,3,3).length()); //Chapter 11
    camera cam(vec3(13,2,3), vec3(0,0,0), vec3(0,1,0), 20, float(nx)/float(ny), 0.1, 10); //Chapter 11

    //rendering loop
    for( int j = ny-1; j >= 0; j-- ){
        for( int i = 0; i < nx; i++ ){

            /* //Chapter 1:
            float r = float(i) / float(nx);
            float g = float(j) / float(ny);
            float b = 0.2;

            int ir = int(255.99 * r);
            int ig = int(255.99 * g);
            int ib = int(255.99 * b);
            */

            /*//Chapter 2:
            vec3 col( float(i) / float(nx), float(j) / float(ny), 0.2);
            */

            /* //Chapter 3+4+5.1:
            vec3 col = color(r);*/

            /*
            float u = float(i) / float(nx);
            float v = float(j) / float(ny);
            ray r( origin, lower_left_corner + u*horizontal + v*vertical );*/

            /* //Chapter 5.2:
            vec3 p = r.point_at_parameter(2.0);
            vec3 col = color(r, world);
            */

            //Chapter 6:
            vec3 col(0,0,0);
            for(int s = 0; s < ns; s++){

                float u = float( i + drand48() ) / float(nx); //random sample placement
                float v = float( j + drand48() ) / float(ny);

                ray r = cam.get_ray(u,v);
                vec3 p = r.point_at_parameter(2.0);

                col += color( r, world, 0 );//no last arg for chapter <8

            }
            col /= float(ns);

            //Chapter 7: raising color to gamma 2
            col = vec3( sqrt(col[0]), sqrt(col[1]), sqrt(col[2]) );

            int ir = int(255.99 * col[0]);
            int ig = int(255.99 * col[1]);
            int ib = int(255.99 * col[2]);

            outfile << ir << " " << ig << " " << ib << "\n";
        }
    }

    outfile.close();// closes file;
    cout << "Image rendered successfully." << endl;
    return 0;
}
```

FIGURE 19 – Beaucoup de code perdu au fur et à mesure qu'on avance...

4 Opinion globale sur ce projet.

Pour être direct, je dirais que faire ce tutoriel semble apporter peu de choses comparé à la simple lecture du document, ce qui me semble être plutôt dommage. Ce tutoriel est clair, mais pas vraiment stimulant, étant donnée qu'il s'agit de recopiage de code assez brute. Il gagnerait à être raccourci, et à proposer des parties plus intéractives où on incite à essayer de coder soit-même la fonction expliquée par le chapitre (quitte à aller voir la solution plus tard si on ne trouve pas). Cela demanderait quelques modifications dans l'énoncé et probablement de devoir proposer des versions intermédiaires du code en téléchargement, plutôt que simplement la version finale.

Je pense également que d'une manière générale, le sujet est un peu long (environ 10h) au vue du nombre de projets à réaliser en CONAV et de la spécificité du sujet traité ici. Toutefois, je trouve tout-de-même l'idée d'approfondir le raycasting intéressante, et je suis assez content d'avoir fait ce tutoriel.