École Centrale Lyon

S9
Informatique Graphique
Rapport

Rapport

Elèves :
Thomas Pennamen

Enseignants :
Nicolas Bonneel

24 mars 2021

Table des matières

1 Introduction

Dans ce projet, nous allons traiter la méthode de synthèse d'image par Ray Tracing. Tout le projet s'articule autour des différents effets visuels pour créer des scènes physiquement réalistes en terme d'éclairage, de forme et de perspective. Pour ce faire, nous utiliserons le langage C++.

2 Objets et Fonctionnalités de base

2.1 Caméra

Pour faire une scène cohérente, nous allons d'abord créer une caméra. À partir de cette dernière, nous allons lancer des "rayons" qui vont percuter les éléments de la scène pour créer l'image. Dans la première étape, chaque pixel de l'image sera uniquement la synthèse d'un rayon qui percute un élément. Chacun des pixels est composé de trois valeurs (rouge, bleu, vert). Dans le code ci dessous, W est la largeur de l'image en nombre de pixel et H est la hauteur.

```
for (int i = 0; i < H; i++)
{
    for (int j = 0; j < W; j++)
    {
        Vector u(j - W / 2, i - H / 2, -W / (2. * tan(fov / 2)));
        Ray r(C, u);
        Vector color(0., 0., 0.);

        color = scene.getColor(r, 0);

        image[((H - i - 1) * W + j) * 3 + 0] =
        std::min(255., std::pow(color[0], 0.45));
        image[((H - i - 1) * W + j) * 3 + 1] =
        std::min(255., std::pow(color[1], 0.45));
        image[((H - i - 1) * W + j) * 3 + 2] =
        std::min(255., std::pow(color[2], 0.45));
    }
}
```

FIGURE 1 – Calcul de la valeur des pixels depuis la caméra

Vous trouverez en annexe la description de Vector, Ray qui seront utiles tout au long du projet.

La fonction getColor de la classe Scene va être l'enjeu crucial du projet. Elle permet, à partir d'une caméra et d'un angle donné -considéré comme un rayon-, d'avoir la couleur du pixel qui découle de l'intersection du premier objet de la scène rencontré par le rayon.

2.2 Sphère

L'objet de base qui nous servira de test pour l'entierté du projet est la sphère. Elle sera défini par un centre et un rayon au départ. Son centre est un Vector de 3 coordonnées. Nous lui ajouterons un albedo sous la forme d'un Vector des trois couleurs rgb.

À partir de cette sphère, nous pouvons faire les murs de la scène. Nous prendrons des sphères avec un rayon suffisamment élevées pour qu'elles soient considérées du point de vue de la caméra comme des plans (1000 de rayon).

2.3 Lumière

Pour avoir des formes et des ombres, il est nécessaire d'ajouter une lumière dans la scène. Dans un premier temps, nous allons considérer que c'est une source ponctuelle de lumière.

Pour déterminer l'intensité de la lumière sur une sphère, il faut avoir le point d'intersection du rayon avec la sphère, on le nommera P. Ensuite, il faut calculer le produit scalaire entre la normale de la sphère et le vecteur PL avec L la position de la lumière le tout divisé par la distance (norme du vecteur PL) au carré.

$$Couleur = albedo_{sphere}.I_{lumiere}.(P\vec{L}/d.\vec{N})/d^3.\pi^2$$

Avec cela, nous avons tous les éléments pour pouvoir créer notre première scène. Pour la suite, la lumière sera une sphère.

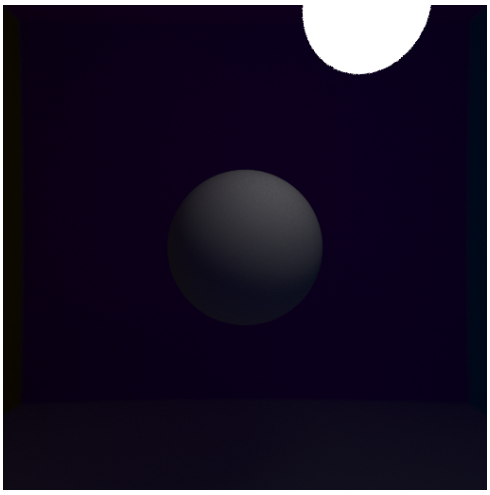

FIGURE 2 – Premier rendu avec une sphère et une lumière

3 Fonctionnalités optiques ajoutées

3.1 Propriété du miroir

Une propriété intéressante pour les objets est la propriété miroir. L'intérêt derrière est de pouvoir avoir la notion de rebond pour un rayon. En effet, pour pouvoir rendre l'effet miroir, lorsque le rayon percute la sphère réfléchissante, il rebondit sur la surface. La couleur ainsi fournie au pixel de la sphère est celle obtenue au bout du rebond.

3.2 Propriété de transparence

Une autre propriété est le fait d'avoir une transparence. Les rayons percutants la sphère ne vont pas rebondir cette fois mais être réfracté. En suivant la loi de Descartes.

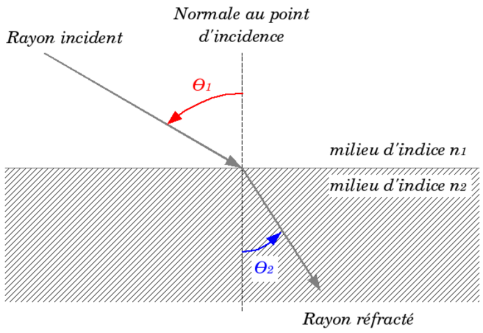

FIGURE 3 – Réfraction de Descartes

En suivant ce schéma, le rayon sortira en suivant la formule :

$$n_1.sin(\theta_1) = n_2.sin(\theta_2$$

3.3 Éclairage indirect

À partir de maintenant, nous allons synthétiser la lumière d'une autre façon, au lieu d'avoir un rayon par pixel, nous allons en avoir plusieurs. Chacun de ses rayons vont se réfléchir pour aller chercher les contributions des autres éléments de la scène sur l'élément premièrement percuté. Ainsi nous ferons la moyenne des contributions de chaque rayon pour avoir donc l'image finale. Cela ajoute beaucoup plus de temps de calcul, il est donc nécessaire de paralléliser l'algorithme pour rester dans des temps de calcul acceptables.

Cela se base physiquement sur les BRDFs composées en partie d'une intégrale sur toute la plage d'angles possibles de rebond d'un rayon sur une sphère. Pour calculer cette intégrale, nous utiliserons la méthode Monte-Carlo. Elle consiste à approcher la valeur d'une intégrale en faisant la moyenne de la fonction évaluée sur un grand nombre de points aléatoires de l'intervalle couvert par l'intégrale.

Avec ces trois premières fonctionnalités avancées, on peut obtenir des scènes comme la figure ci-dessous :

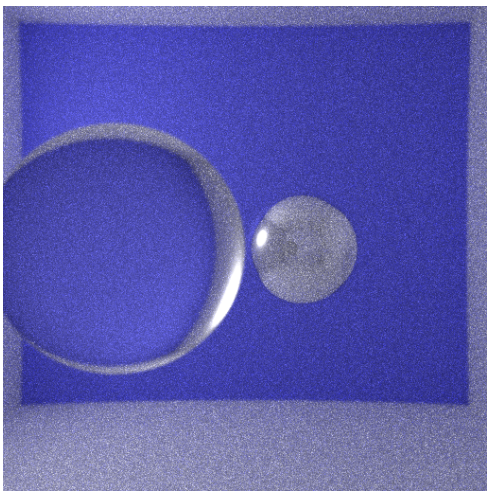

FIGURE 4 – Éclairage indirect avec boule transparente et miroir (1000 rayons)

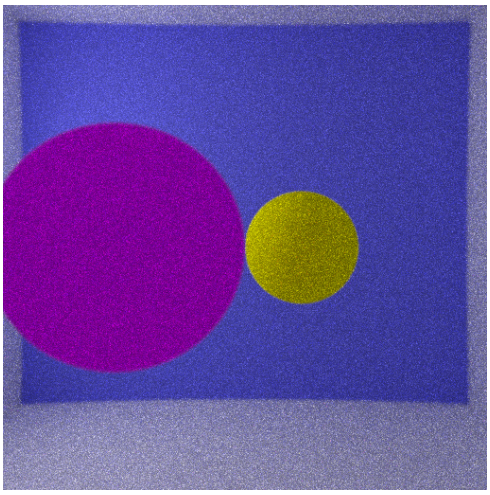

FIGURE 5 – Éclairage indirect avec boules classiques (1000 rayons)

3.4 Antialiasing

Le fait que chaque pixel soit synthétisé par contact ou non avec un objet fait cet effet "crénelé" sur les contours des objets. Pour résoudre ce problème, il suffit d'ajouter une composante aléatoire pour les coordonnées x et y du vecteur de la caméra. Cela fera la synthèse donc de rayon percutant l'objet et d'autres non pour avoir cet aspect plus nuancé sur les bords.

3.5 Profondeur de champ

En photographie classique, un aspect n'est pas présent ici, à savoir la notion de focale. Un objet est net uniquement s'il est à une distance de la caméra proche de la distance de focale. S'il est trop prêt ou trop loin, il sera flou.

4 Maillage

Maintenant que les effets optiques sont implémentés, nous pouvons ajouter des objets plus complexes dans notre scène. Nous prendrons un chien du type berger australien pour l'exemple.

4.1 La recherche efficace d'intersection

L'ajout d'objet du type maillage donne un inconvénient majeur en terme de temps d'exécution. En effet, dans une méthode brute de recherche ou non d'intersection, nous devons parcourir l'entièreté des vertices que possèdent le chien et tester si le rayon les touche ou non. La méthode est donc d'utiliser des boites englobantes ; autrement dit, chaque vertex du maillage sera dans une boite indiquant les coordonnées max et min de l'ensemble des vertices de cette boite. Cette boite sera elle-même stockée dans une autre boite plus englobante avec d'autres et ainsi de suite. Le but étant de parcourir les boites de plus en plus précisément pour éviter un parcours brutes de la liste des vertices.

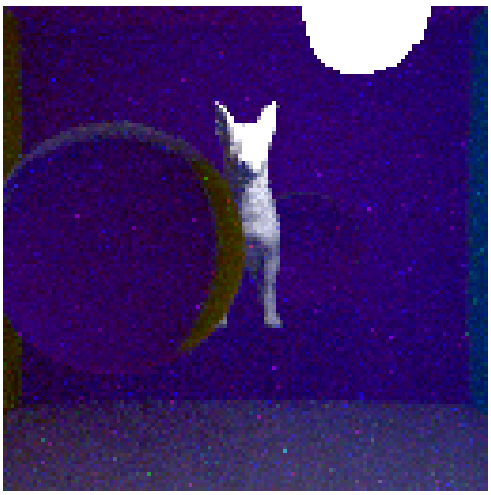

FIGURE 6 – Intégration simple du maillage (20 rayons)

4.2 Texture

Les objets de type maillage peuvent être complétés par des textures pour ajouter une couleur à chaque vertex. Ce type de fichier contenant les textures sont stockées sous un type image. Pour coupler les vertices avec leur texture associée, il suffit d'associer la position dans l'image avec celle du vertex.

FIGURE 7 – Intégration avec texture du maillage (20 rayons)

5 Conclusion & retours sur le cours

Ce projet nous a permis d'une part d'apprendre les bases du langage C++ et d'une autre les méthodes d'informatique graphique. L'intérêt premier pour moi était de découvrir comment la géométrie et les effets visuels étaient créés informatiquement.

Cependant, la barrière du langage complexifie grandement la tâche. Je pense qu'il aurait été préférable de faire une introduction ou nous partager des exercices de C++ pour que l'on puisse s'y accommoder. Une autre solution aurait aussi été, au lieu d'écrire le code directement sous nous yeux (ce qui nous influence dans tous les cas), d'écrire du pseudo-code pour nous forcer à l'adapter par nous-mêmes.

6 Annexes

```
class Vector
{
public:
    explicit Vector(double x = 0, double y = 0, double z = 0)
    {
        coords[0] = x;
        coords[1] = y;
        coords[2] = z;
    }
    double operator[](int i) const { return coords[i]; };
    double &operator[](int i) { return coords[i]; };
    double sqrtNorm() const
    {
        return coords[0] * coords[0] + coords[1] * coords[1] + coords[2] * coords[2];
    }
    Vector get_normalized()
    {
        double n = sqrt(sqrtNorm());
        return Vector(coords[0] / n, coords[1] / n, coords[2] / n);
    }

private:
    double coords[3];
};
Vector operator+(const Vector &a, const Vector &b){
    return Vector(a[0] + b[0], a[1] + b[1], a[2] + b[2]);}
Vector operator-(const Vector &a, const Vector &b){
    return Vector(a[0] - b[0], a[1] - b[1], a[2] - b[2]);}
Vector operator*(double a, const Vector &b){
    return Vector(a * b[0], a * b[1], a * b[2]);}
Vector operator*(const Vector &b, double a){
    return Vector(a * b[0], a * b[1], a * b[2]);}
Vector operator*(const Vector &a, const Vector &b){
    return Vector(a[0] * b[0], a[1] * b[1], a[2] * b[2]);}
Vector operator/(const Vector &b, double a){
    return Vector(b[0] / a, b[1] / a, b[2] / a);}
Vector cross(const Vector &a, const Vector &b){
    return Vector(a[1] * b[2] - b[1] * a[2], a[2] * b[0] - b[2] * a[0], a[0] * b[1] -
double dot(const Vector &a, const Vector &b){
    return a[0] * b[0] + a[1] * b[1] + a[2] * b[2];};
double sqr(double x){    return x * x;}
```

FIGURE 8 – Classe Vector et ses opérateurs

```
class Ray
{
public:
    Vector C;
    Vector U;
    Ray(const Vector &C, Vector &U) : C(C), U(U.get_normalized())
    {
    }
};
```

FIGURE 9 – Classe Ray