compte rendu lancer de rayon phase 1 et 2

Emilien Rey

novembre 2024

1 Phase 1

La première partie consiste à mettre en place le calcul des intersections entre les rayons et les différents éléments de la scène. Dans notre cas, il faut traiter la sphère et le carré. On pourrait également calculer les intersections avec les triangles, mais ici cela ne sert pas.

Pour les intersections avec la sphère, on utilise cette formule :

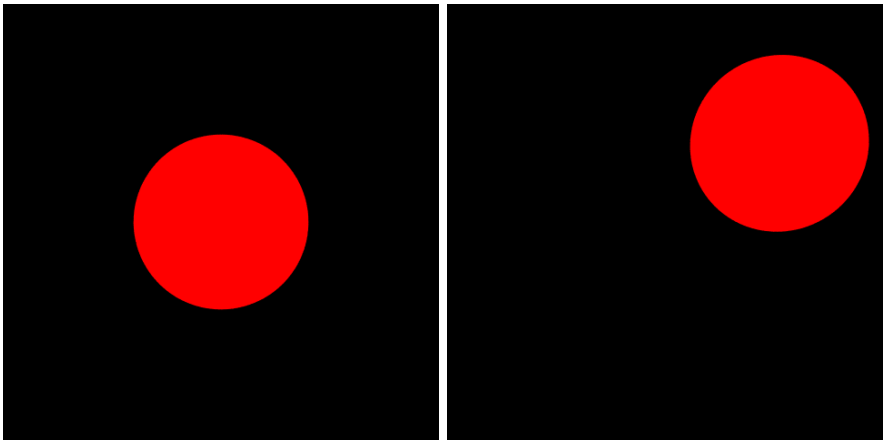
$$t^2\,\mathbf{d}\cdot\mathbf{d} + 2t\,\mathbf{d}\cdot(\mathbf{o}-\mathbf{c}) + ||\mathbf{o}-\mathbf{c}||^2 - r^2 = 0$$

't' est l'inconnue dans cette équation. On obtient une équation du second degré dont le résultat détermine les deux intersections, si elles existent. De plus, lorsqu'il y a deux intersections, donc deux valeurs de 't', on prend la plus petite des deux (et non négative), car cela correspond à l'intersection visible par la caméra.

```
RaySphereIntersection intersect(const Ray &ray) const {
    RaySphereIntersection intersection;

    Vec3 origin =ray.origin();
    Vec3 direction =ray.direction();


    float a=dot(direction,direction);

    Vec3 oc=origin-m_center;
    float b=2.0*dot(direction,oc);

    float c=dot(oc,oc)-m_radius*m_radius;

    float discriminant = b * b - 4 * a * c;

    if (discriminant < 0) {
        intersection.intersectionExists=false;
        return intersection;
    } else {
        float t1 = (-b - std::sqrt(discriminant)) / (2.0f * a);
        float t2 = (-b + std::sqrt(discriminant)) / (2.0f * a);
        Vec3 point_intersection1= origin+t1*direction;
        Vec3 point_intersection2= origin+t2*direction;
```

Figure 1: code de l'intersection avec la sphère

Ce qui donne ceci:



Ensuite, pour le carré, il faut d'abord modifier le code des translations et rotations utilisé pour mettre en place la scène, car celui-ci ne modifie pas la valeur des normales, des vecteurs de direction, ni du point de départ. Pour ce faire, j'ai redéfini les fonctions pour réaliser ces transformations et, notamment, modifié la fonction suivante pour ajuster les valeurs :

```
void apply_rotation(Mat3 rotation_matrix) {
    m_bottom_left = rotation_matrix * m_bottom_left;
    m_right_vector = rotation_matrix * m_right_vector;
    m_up_vector = rotation_matrix * m_up_vector;
    m_normal = rotation_matrix * m_normal;
    m_normal.normalize();
}
```

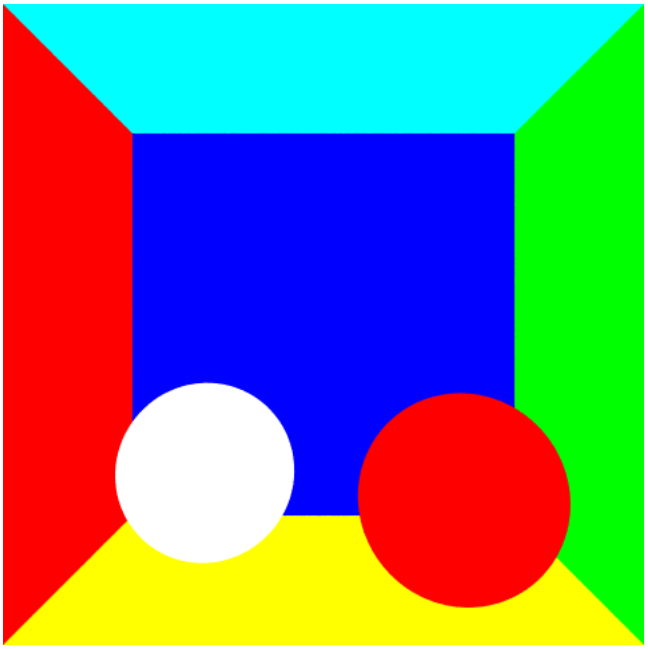
Une fois cela fait, j'ai écrit le code pour calculer l'intersection avec le carré. Pour cela, je teste d'abord si le rayon a une intersection avec le plan défini par les vecteurs qui définissent le carré, en utilisant la formule suivante :

$$t = \frac{D - \boldsymbol{o}.\boldsymbol{n}}{\boldsymbol{d}.\boldsymbol{n}}$$

Une fois 't' calculé, on obtient un point que l'on teste pour vérifier s'il se trouve dans le carré. Si c'est le cas, on le retourne.

Afin de calculer les intersections dans une scène contenant des sphères et des carrés, il faut, dans la fonction computeIntersection, retourner l'intersection la plus proche de la caméra.

Avec tout cela, on peut obtenir la boîte de Cornell suivante :



Il n'y a pas de lumière ni d'ombre, on se contente ici de récupérer la couleur du point d'intersection.

2 Phase 2

Dans cette seconde partie, nous allons nous intéresser au calcul de la lumière et de l'ombrage afin d'obtenir un meilleur rendu. Pour commencer, nous allons calculer l'éclairage en chaque point d'intersection en utilisant le modèle de Phong, défini comme suit :

$$I = I_a + I_d + I_s$$

Le modèle de Phong correspond à la somme des réflexions ambiante, diffuse et spéculaire. Le calcul de la réflexion ambiante correspond au produit de l'intensité de la lumière ambiante réfléchie et de l'intensité de la lumière ambiante. Dans le code, on la calcule ainsi :

```
Vec3 ambient= Vec3(material.ambient_material[0] * lightColor[0],
                   material.ambient_material[1] * lightColor[1],
                   material.ambient_material[2] * lightColor[2]
                  );
```

Ensuite, la réflexion diffuse se calcule en faisant le produit de l'intensité de la lumière diffuse, du coefficient de réflexion diffuse du matériau et du cosinus de l'angle entre la source de lumière et la normale. On peut remplacer le cosinus de l'angle par le produit scalaire entre le vecteur de lumière et la normale.

$$I_d = I_{sd} * K_d * \cos\theta$$

```
float diffuseFactor = std::max(dot(normal, lightDir), 0.0f);

double diffusex = material.diffuse_material[0] * lightColor[0] * diffuseFactor;
double diffusey = material.diffuse_material[1] * lightColor[1] * diffuseFactor;
double diffusez = material.diffuse_material[2] * lightColor[2] * diffuseFactor;

Vec3 diffuse = Vec3(diffusex,diffusey,diffusez);
```

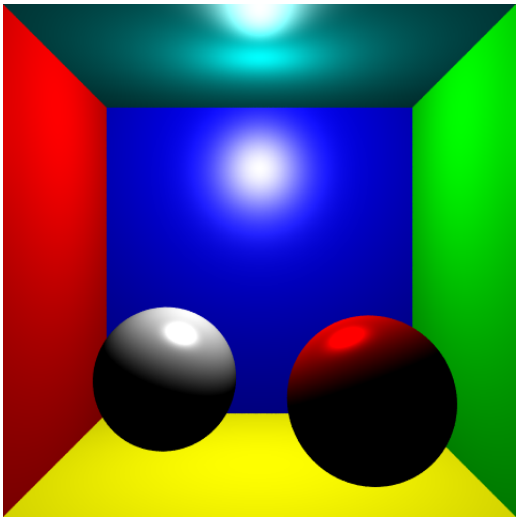
Enfin, la réflexion spéculaire se calcule de la manière suivante :

$$I_s = I_{ss} * K_s * \left(\vec{R}.\vec{V}\right)^n$$

Avec Iss l'intensité de la lumière spéculaire de la source, Ks le coefficient de réflexion spéculaire du matériau, R la direction de réflexion de la lumière sur un miroir, V la direction de la vue et n le coefficient de brillance de l'objet. Le code pour le calcul est le suivant :

```
Vec3 viewDir = -1*rayStart.direction();
Vec3 reflectDir = 2 * dot(normal, lightDir) * normal - lightDir;
float specFactor = std::pow(std::max(dot(viewDir, reflectDir), 0.0f), material.shininess);
Vec3 specular(
    material.specular_material[0] * lightColor[0] * specFactor,
    material.specular_material[1] * lightColor[1] * specFactor,
    material.specular_material[2] * lightColor[2] * specFactor
);
```

Avec cela fait, on obtient le rendu suivant :



Maintenant, nous allons ajouter des ombres au rendu. La première tentative pour ajouter des ombres consiste à calculer les intersections entre un rayon partant du point d'intersection et allant vers la lumière. Si ce rayon trouve une intersection avant d'arriver à la lumière, alors ce point est dans l'ombre. Dans notre cas, l'ombre est causée par la présence des sphères. Donc, pour simplifier, on vérifie juste si le rayon intersecte une sphère, et si c'est le cas, alors le point est dans l'ombre. De plus, afin d'éviter des problèmes, on considère qu'il y a une intersection seulement si 't' est supérieur à 0,01, ce qui permet d'éviter que les points des sphères soient considérés comme faisant partie de l'ombre. De plus, on teste si le nouveau 't' est supérieur à la distance entre le point d'intersection et la lumière. Si c'est le cas, cela signifie que le point d'intersection de base se situe derrière la lumière et qu'il n'y a pas d'objet devant lui, donc on ignore le 't' trouvé. Si l'on ne fait pas cela, on verra apparaître des projections d'ombres des sphères au niveau du toit.

```
Ray rayOmbre=Ray(pointIntersection , lightDir);

float maxDistanceToLight = (lightPos - pointIntersection).length();

RaySceneIntersection rayOmbreIntersection = computeIntersection(rayOmbre);

if( rayOmbreIntersection.intersectionExists && rayOmbreIntersection.raySphereIntersection.t > 0.01
&& rayOmbreIntersection.raySphereIntersection.t < maxDistanceToLight){
    return Vec3(0,0,0);
}
```

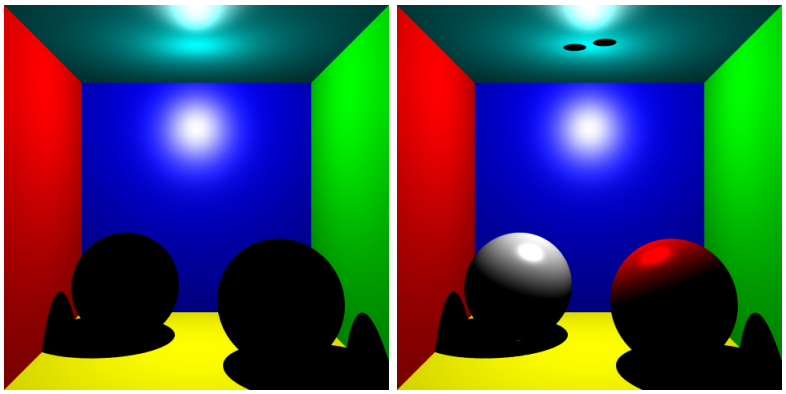
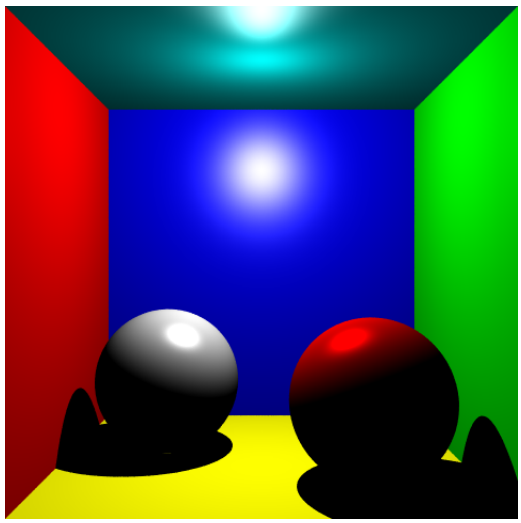


Figure 2: À gauche, le rendu sans test de 't', à droite, le rendu sans comparaison avec la distance à la lumière.

Une fois cela fait, on obtient le rendu suivant :

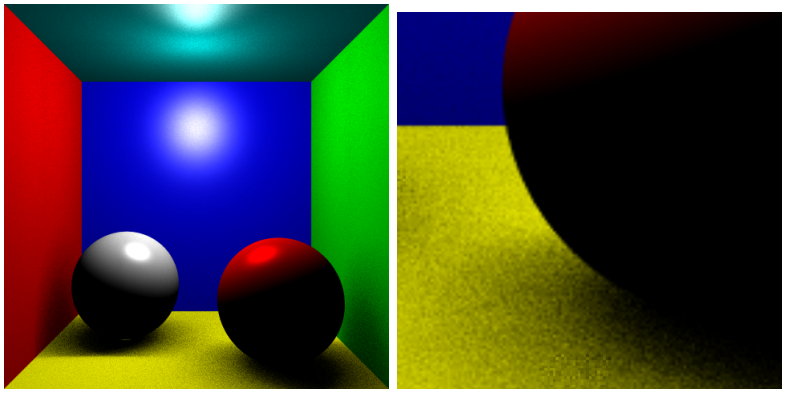


On va maintenant s'occuper du rendu avec des ombres douces. Pour calculer celles-ci, on doit considérer un carré autour de la lumière et échantillonner des

points au hasard à l'intérieur. On teste alors les intersections comme pour l'ombre non douce, mais cette fois-ci avec tous les points échantillonnés. On compte le nombre de lumières atteignant le point d'intersection, ce qui nous donne un coefficient qu'on utilise pour multiplier la couleur du point. Si aucune lumière n'a touché le point, le coefficient vaut 0, ce qui donne une couleur noire, tandis que si tous les points ont touché, on obtient un coefficient égal à 1, ce qui ne modifie pas la couleur de base du point. Le code suivant permet de calculer le coefficient de l'ombre :

```
int echan = 50;
int touche = 0;

for (int i = 0; i < echan; i++) {
    Vec3 lightEchanPos = randomPointDansCarre(lightPos, lights[0].radius);
    Vec3 lightEchanDir = (lightEchanPos - pointIntersection);
    lightEchanDir.normalize();
    float maxDistanceToLight = (lightEchanPos - pointIntersection).length();

    Ray shadowRay(pointIntersection + normal * 0.00001f, lightEchanDir);
    RaySceneIntersection ombreIntersection = computeIntersection(shadowRay);

    if (ombreIntersection.intersectionExists && ombreIntersection.raySphereIntersection.t > 0.01
        && ombreIntersection.raySphereIntersection.t < maxDistanceToLight) {
        touche++;

    }
}

float facteurOmbre = 1.0f - (float)touche / echan;
```

Suivant le nombre d'échantillons que l'on prend, le rendu ne sera pas identique. Avec peu d'échantillons, on obtient un rendu peu convaincant, comme ici où seulement 2 échantillons ont été pris :



Si l'on zoom dans l'image, et plus précisément dans l'ombre des sphères, on voit que la couleur de l'ombre n'est pas cohérente et alterne entre des points dans l'ombre et d'autres n'en faisant pas partie. Pour éviter ce problème, il faut choisir un nombre d'échantillons suffisamment grand. Le rendu suivant a été réalisé avec 20 échantillons.