Nicolas Luciani
Mer. 9 Novembre 2022
Master 1 IMAGINE Université de Montpellier

Projet RayTracing
Compte-rendu

---

1 Introduction

Le projet de raytracer se décompose en plusieurs phases. Nous détaillerons ici des portions de code ainsi que les résultats obtenus. Seules les portions de code pertinentes pour l'obtention d'un résultat seront explicitées.

2 Phase 1

2.1 Fonction de lancer de rayon

Le lancer de rayon se décompose en plusieurs méthode dans le fichier Scene.h. La fonction rayTrace démarre le lancer de rayon et fais appel à la méthode rayTraceRecursive. Pour l'instant, le nombre de rebonds est à 1 donc il n'y a pas encore réellement d'appels récursifs.

```
Vec3 rayTrace( Ray const & rayStart )
{
        Vec3 color;
        color = rayTraceRecursive(rayStart, 1);
        return color;
}

Vec3 rayTraceRecursive( Ray ray , int NRemainingBounces)
{
    RaySceneIntersection raySceneIntersection = computeIntersection(ray, 4.9f);
    Vec3 color;
    if(raySceneIntersection.intersectionExists)
    {
        if(raySceneIntersection.typeOfIntersectedObject == ObjectType_Sphere)
        {
          color = spheres[raySceneIntersection.objectIndex].material.diffuse_material;
        }
        else if(raySceneIntersection.typeOfIntersectedObject == ObjectType_Square)
        {
            color = squares[raySceneIntersection.objectIndex].material.diffuse_material;
        }
    }
    return color;
}
```

La fonction computeIntersection permet de calculer l'intersection du rayon avec l'objet le plus proche de l'origine du rayon. La valeur flottante passée en paramètre permet de ne traiter les intersections à une distance supérieure à cette valeur entre l'origine du rayon et le point d'intersection.

```
RaySceneIntersection computeIntersection(Ray const & ray, float minDist)
{
    RaySceneIntersection result;
    float t = FLT_MAX;
    for(size_t i = 0; i<spheres.size(); i++)
    {
        RaySphereIntersection intersection = spheres[i].intersect(ray);
        if(intersection.intersectionExists && intersection.t < t && intersection.t > minDist)
        {
            t = intersection.t;
            result.intersectionExists = true;
            result.typeOfIntersectedObject = ObjectType_Sphere;
            result.objectIndex = i;
            result.t = t;
            result.raySphereIntersection = intersection;
        }
    }
    for(size_t i = 0; i<squares.size(); i++)
    {
        RaySquareIntersection intersection = squares[i].intersect(ray);
        if(intersection.intersectionExists && intersection.t < t && intersection.t > minDist)
        {
            t = intersection.t;
            result.intersectionExists = true;
            result.typeOfIntersectedObject = ObjectType_Square;
            result.objectIndex = i;
            result.t = t;
            result.raySquareIntersection = intersection;

        }
    }
    return result;
}
```

Les calculs d'intersections assurés par les fonctions intersect sont détaillés ci-après.

2.2 Intersection Rayon-Sphère

Le calcul du point d'intersection entre un rayon et une sphère est assuré par la fonction intersect dans le fichier Sphere.h. La fonction résout l'équation d'inconnue t suivante :

$$t^2 d.d + 2td.(o - c) + ||o - c||^2 - r^2 = 0 \tag{1}$$

telle que :

- d est le vecteur directeur du rayon
- o l'origine du rayon
- c le centre de la sphère
- r le rayon de la sphère

Après la résolution de l'équation, la fonction calcule les coordonnées du point d'intersection et sa normale.

```
RaySphereIntersection intersect(const Ray &ray) const {
    RaySphereIntersection rayinter;
    Vec3 CO = ray.origin() - m_center;
    double a = Vec3::dot(ray.direction(), ray.direction());
    double b = 2 * Vec3::dot(ray.direction(), CO);
    double c = pow(CO.length(), 2) - pow(m_radius, 2);
    double disc = pow(b,2) - 4*a*c;
    if(disc < 0 || (2*a) == 0)
    {
      rayinter.intersectionExists = false;
      return rayinter;
    }
    double t1 = (b*(-1) - sqrt(disc))/(2*a);
    double t2 = (b*(-1) + sqrt(disc))/(2*a);
    if(t1>0 && (t1<t2 || t2<0))
    {
        rayinter.intersectionExists = true;
      rayinter.t = t1;
        Vec3 point = ray.direction() * t1 + ray.origin();
        rayinter.intersection = point;
        Vec3 normal = point - m_center;
        normal.normalize();
        rayinter.normal = normal;

    }
    else{
        rayinter.intersectionExists = true;
        rayinter.t = t2;
        Vec3 point = ray.direction() * t2 + ray.origin();
        rayinter.intersection = point;
        Vec3 normal = point - m_center;
        normal.normalize();
        rayinter.normal = normal;
    }

    return rayinter;
}
```

On obtient donc les résultats suivants. On montrera la scène 3D et le rendu du rayTracing.

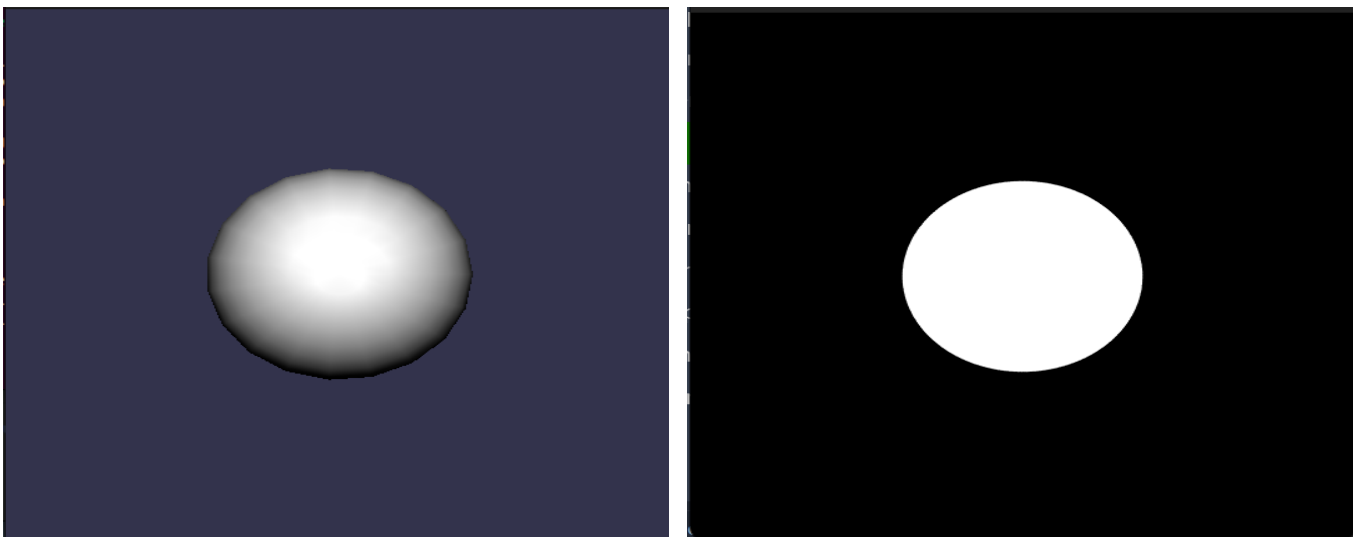

Figure 1: Sphère seule

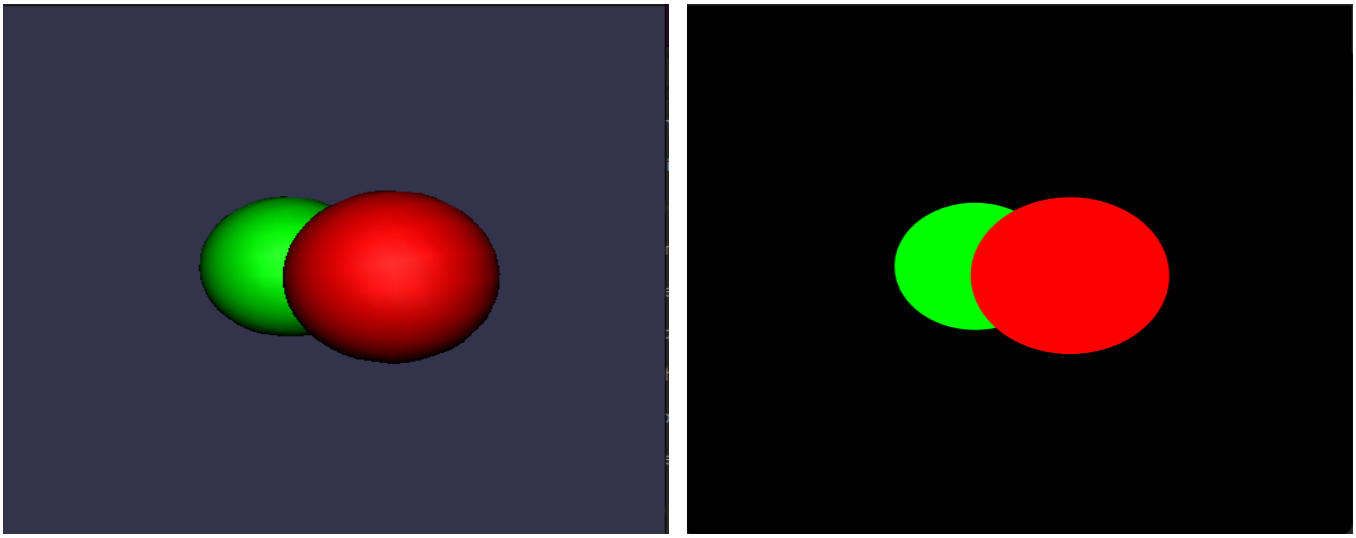

Figure 2: Deux sphères

2.3 Intersection Rayon-Carré

La fonction intersect du fichier Square.h calcule le point d'intersection entre un rayon et un carré. Dans un premier temps la fonction résout l'équation suivante (équation permettant de trouver l'intersection entre le rayon et le plan portant le carré):

$$D = a.n \tag{2}$$

$$t = \frac{D - o.n}{d.n} \tag{3}$$

telle que :

- a un point du plan
- o est l'origine du rayon
- n est la normale au plan portant le carré
- d est le vecteur directeur du rayon

Ensuite, il faut vérifier si le point d'intersection se trouve dans le carré. Pour cela, on définit un vecteur entre le point d'intersection et un sommet du carré. Si le produit scalaire entre ce vecteur et chacune des deux arêtes formant se sommets est compris entre 0 et 1, alors le point se situe dans le carré.

```
1  RaySquareIntersection intersect(const Ray &ray) const {
2      RaySquareIntersection intersection;
3      float D = Vec3::dot(bottomLeft(), normal());
4      float t = ( D - Vec3::dot(ray.origin(), normal()))/Vec3::dot(ray.direction(), normal
       ());
5      Vec3 point = ray.direction() * t + ray.origin();
6      Vec3 vecToBottom = point - bottomLeft();
7      Vec3 X_axis = (bottomRight() - bottomLeft());
8      Vec3 Y_axis = (upLeft() - bottomLeft());
9      float u = Vec3::dot(vecToBottom, X_axis);
10     float v = Vec3::dot(vecToBottom, Y_axis);
11     if(    u > 0 && u < Vec3::dot(X_axis, X_axis)
```

```
        && v > 0 && v < Vec3::dot(Y_axis, Y_axis))
    {
        intersection.intersectionExists = true;
        intersection.t = t;
        intersection.intersection = point;
        intersection.normal = normal();
        return intersection;
    }
    else
    {
        intersection.intersectionExists = false;
        return intersection;
    }
}
```

On a donc le résultat suivant.

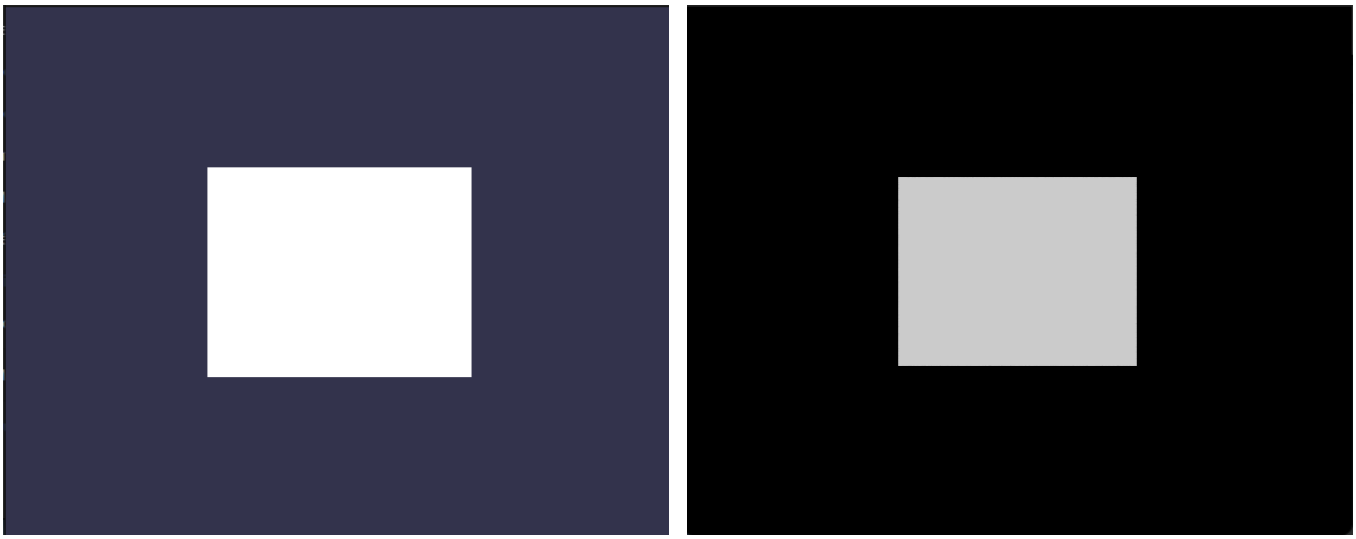

Figure 3: Carré

2.4 Boîte de Cornell

Une scène représentant une boîte de cornell permet de tester la totalité de nos fonctions. Nous utiliserons donc cette scène pour la suite du projet.

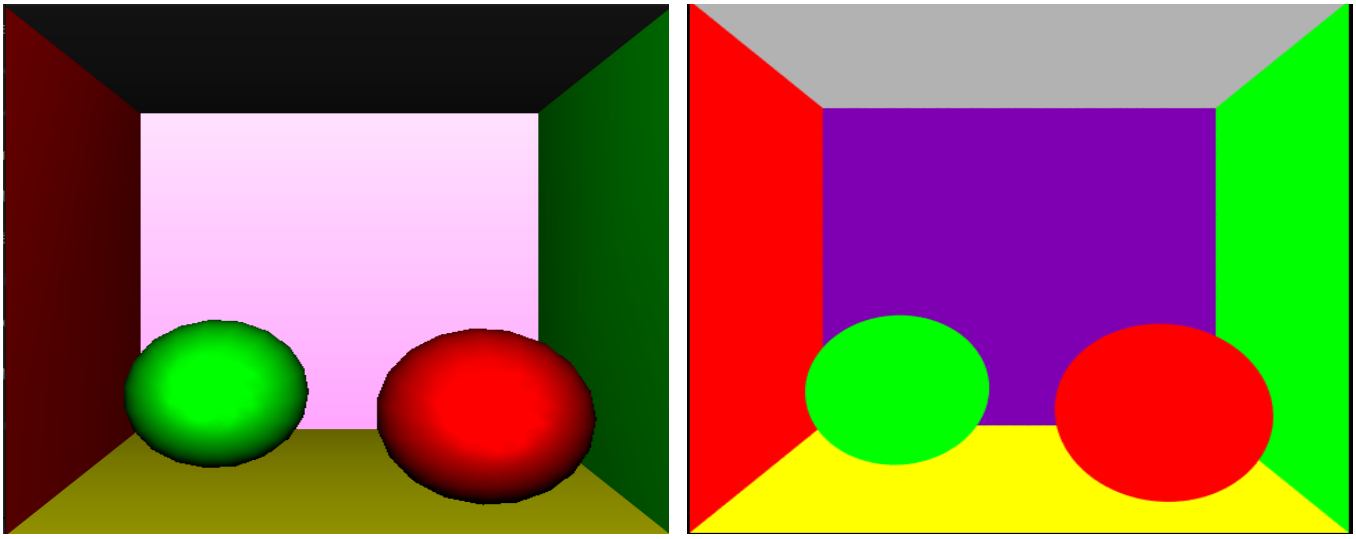

Figure 4: Boîte de Cornell

3 Phase 2

3.1 Illumination de Phong

Le calcul de l'illumination de Phong est réalisé dans deux fonctions dont la différence réside dans le type d'objet intersecté et donc l'accession aux paramètres nécessaires pour le calcul. On reprends les formules du cours mais nous l'intensité ambiante a été écartée car nulle ici.

```
Vec3 phong_sphere (Ray ray, Light light, RaySphereIntersection intersection, int index)
{
  Vec3 L = light.pos - intersection.intersection;
    L.normalize();
    Vec3 normal = intersection.normal;
    Vec3 reflexion = 2 * (Vec3::dot(normal, L)) * normal - L;
    reflexion.normalize();
    Vec3 rayDir = (-1) * ray.direction();
    rayDir.normalize();
    Vec3 specular = Vec3::compProduct(
            light.material,
            spheres[index].material.specular_material) *
            pow(Vec3::dot(reflexion, rayDir), spheres[index].material.shininess);
    Vec3 diffuse=Vec3::compProduct(
            light.material,
            spheres[index].material.diffuse_material)*
           Vec3::dot(L,normal);

    return specular + diffuse;

}

Vec3 phong_square(Ray ray, Light light, RaySquareIntersection intersection, int index)
{
  Vec3 L = light.pos - intersection.intersection;
    L.normalize();
    Vec3 normal = intersection.normal;
    Vec3 reflexion = 2 * (Vec3::dot(normal, L)) * normal - L;
    reflexion.normalize();
    Vec3 rayDir = (-1) * ray.direction();
```

```
    rayDir.normalize();
    Vec3 specular = Vec3::compProduct(light.material,
                      squares[index].material.specular_material) *
                      pow(Vec3::dot(reflexion, rayDir),
                          squares[index].material.shininess);
    Vec3 diffuse =  Vec3::compProduct(light.material,
                      squares[index].material.diffuse_material)  *
                      Vec3::dot(L,normal);

    return diffuse + specular;
}
```

La fonction rayTraceRecursive a donc été modifiée pour utiliser le résultat de ces calculs.

```
Vec3 rayTraceRecursive( Ray ray , int NRemainingBounces)
{
    RaySceneIntersection raySceneIntersection = computeIntersection(ray, 4.9f);
    Vec3 color;
    if(raySceneIntersection.intersectionExists)
    {
         if(raySceneIntersection.typeOfIntersectedObject == ObjectType_Sphere)
            for (int i = 0; i < lights.size(); ++i)
                    color += phong_sphere(ray, lights[i], raySceneIntersection.
    raySphereIntersection, raySceneIntersection.objectIndex);
        else if(raySceneIntersection.typeOfIntersectedObject == ObjectType_Square)
            for (int i = 0; i < lights.size(); ++i)
              color += phong_square(ray, lights[i], raySceneIntersection.
    raySquareIntersection, raySceneIntersection.objectIndex);
    }
    return color;
}
```

On obtient donc le résultat suivant

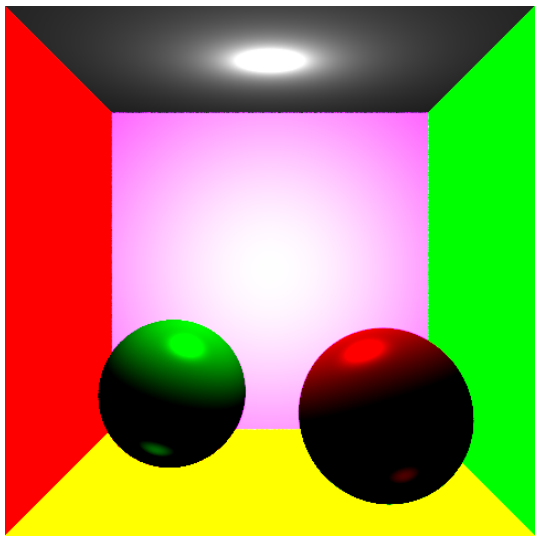

Figure 5: Illumination de Phong

3.2 Calculateur d'ombre

Le calculateur d'ombre calcule tire un rayon entre le point d'intersection et la source de lumière. Si le rayon intersecte un objet, alors l'objet est noir est point d'intersection initial. La fonction qui calcule

l'intersection prends cette fois-ci en paramètre une valeur maximale de t de façon à ne pas calculer une intersection au dela de la source lumineuse.

```
RaySceneIntersection computeShadowIntersection(Ray const & ray, float maxDist)
{
  RaySceneIntersection result;
  for(size_t i = 0; i<spheres.size(); ++i)
  {
    RaySphereIntersection intersection = spheres[i].intersect(ray);
    if(intersection.intersectionExists && intersection.t > 0.001f && intersection.t <
   maxDist)
    {
      result.intersectionExists = true;
      return result;
    }
  }
  for(size_t i = 0; i<squares.size(); ++i)
  {
    RaySquareIntersection intersection = squares[i].intersect(ray);
    if(intersection.intersectionExists && intersection.t > 0.001f && intersection.t <
   maxDist)
    {
      result.intersectionExists = true;
      return result;
    }
  }

  return result;
}
```

La fonction rayTraceRecursive se voit donc à nouveau modifiée pour ajouter cette fonctionnalité.

```
Vec3 rayTraceRecursive( Ray ray , int NRemainingBounces) {
    RaySceneIntersection raySceneIntersection = computeIntersection(ray, 4.9f);
    Vec3 color;
    int light_number = 0;
    if(raySceneIntersection.intersectionExists)
    {

        if(raySceneIntersection.typeOfIntersectedObject == ObjectType_Sphere)
        {
          for (int i = 0; i < lights.size(); ++i)
          {
            Vec3 L = lights[i].pos - raySceneIntersection.raySphereIntersection.
   intersection;
            float shadowMaxDist = L.length();
            L.normalize();
            Ray shadowRay = Ray(raySceneIntersection.raySphereIntersection.intersection,
    L);
            RaySceneIntersection shadowIntersection = computeShadowIntersection(
   shadowRay, shadowMaxDist);
            if(shadowIntersection.intersectionExists)
          {
            return Vec3(0,0,0);
          }
                else
                {
                  color += phong_sphere(ray, lights[i], raySceneIntersection.
   raySphereIntersection, raySceneIntersection.objectIndex);

                }
          }
        }
```

```
28          else if(raySceneIntersection.typeOfIntersectedObject == ObjectType_Square)
29          {
30
31            for (int i = 0; i < lights.size(); ++i)
32            {
33              {
34                  Vec3 L = lights[i].pos - raySceneIntersection.raySquareIntersection.
    intersection;
35                  float shadowMaxDist = L.length();
36                  L.normalize();
37                  Ray shadowRay = Ray(raySceneIntersection.raySquareIntersection.
    intersection, L);
38                  RaySceneIntersection shadowIntersection = computeShadowIntersection(
    shadowRay, shadowMaxDist);
39
40                    if(shadowIntersection.intersectionExists)
41                {
42                  return Vec3(0,0,0);
43                }
44                  else
45                  {
46                    color += phong_square(ray, lights[i], raySceneIntersection.
    raySquareIntersection, raySceneIntersection.objectIndex);
47
48                  }
49              }
50            }
51          }
52
53      }
54      return color;
55  }
```

On obtient finalement le résultat suivant

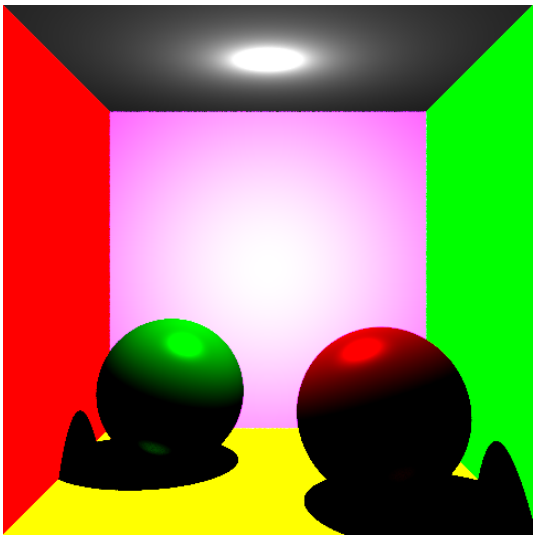

Figure 6: Calcul des ombres

4 Conclusion

Finalement, les ombres adoucies n'ont pas encore été calculées par manque de temps. Cela sera ajouté plus tard.