Programmation 3D

Raytracer phase 3

Louis Jean
Master 1 IMAGINE
Université de Montpellier

6 décembre 2023

Table des matières

1 Introduction

Poursuivant l'aventure du raytracer, la troisième phase de ce projet a mis l'accent sur le peaufinage de la finesse des interactions lumineuses avec les surfaces complexes. Au programme : le calcul minutieux des intersections des rayons avec des maillages, l'interpolation des valeurs de sommets pour un rendu nuancé et la création d'une sphère réfléchissante. J'en ai aussi profité pour faire du nettoyage dans mon code.

2 Intersection rayon-triangle

Premièrement, afin de pouvoir charger un maillage 3D (au format **off**), il a fallu calculer l'intersection rayon-triangle. Après avoir calculé l'intersection du rayon avec le plan support du triangle, je suis ensuite passé aux coordonnées barycentriques. Pour cela, j'ai d'abord tenté d'implémenter la méthode vue dans le cours en utilisant la méthode **area** du fichier **Triangle.h**, en vain. J'ai alors choisi la méthode algébrique en m'inspirant de cet article : `https://codeplea.com/triangular-interpolation`.

```
1  Vec3 getIntersectionPointWithSupportPlane(Line const & L)
       ↪  const {
2          Vec3 result = Vec3(-100.0f,-100.0f,-100.0f);
3          if (isParallelTo(L)) return result;
4          Vec3 P0 = L.origin();
5          Vec3 P1 = m_c[0];
6          Vec3 D = L.direction();
7          float t;
8          t = Vec3::dot(P1 - P0,m_normal) / Vec3::dot(D,
               ↪ m_normal);
9          result = P0 + t * D;
10         return result;
11 }
```

Figure 1: Calcul de l'intersection entre le rayon et le plan support du triangle

```
void computeBarycentricCoordinates(Vec3 const &p, float &
    ↪ u0, float &u1, float &u2) const {
        Vec3 A = m_c[0];
        Vec3 B = m_c[1];
        Vec3 C = m_c[2];
        Vec3 v0 = B - A;
        Vec3 v1 = C - A;
        Vec3 v2 = p - A;
        float d00 = Vec3::dot(v0, v0);
        float d01 = Vec3::dot(v0, v1);
        float d11 = Vec3::dot(v1, v1);
        float d20 = Vec3::dot(v2, v0);
        float d21 = Vec3::dot(v2, v1);
        float denom = d00 * d11 - d01 * d01;
        u1 = (d11 * d20 - d01 * d21) / denom;
        u2 = (d00 * d21 - d01 * d20) / denom;
        u0 = 1.0f - u1 - u2;
}
```

Figure 2: Calcul des coordonnées barycentriques pour un point $\mathbf{p}$ donné

```
RayTriangleIntersection getIntersection(Ray const & ray)
    ↪ const {
     RayTriangleIntersection result;
     result.intersectionExists = false;
     Vec3 intersectionPoint =
        ↪ getIntersectionPointWithSupportPlane(ray);
     Vec3 originToIntersection = intersectionPoint - ray.
        ↪ origin();
     if(Vec3::dot(ray.direction(),originToIntersection) <
        ↪ 0) return result;
     float u0,u1,u2;
     computeBarycentricCoordinates(intersectionPoint, u0,
        ↪ u1, u2);
     if(u0 < 0 || u0 > 1 || u1 < 0 || u1 > 1 || u2 < 0 ||
        ↪ u2 > 1) return result;
     result.intersectionExists = true;
     result.intersection = intersectionPoint;
     result.w0 = u0;
     result.w1 = u1;
     result.w2 = u2;
     result.t = originToIntersection.norm();
     Vec3 normal = Vec3::cross(m_c[1] - m_c[0], m_c[2] -
        ↪ m_c[0]);
     normal.normalize();
     result.normal = normal;
     return result;
}
```

Figure 3: Calcul de l'intersection rayon-triangle

```
RayTriangleIntersection intersect( Ray const & ray )
    ↪ const {
     RayTriangleIntersection closestIntersection;
     closestIntersection.t = FLT_MAX;
     float triangleScaling = 1.000001;
     int nbTriangles = triangles.size();
     for(int i = 0; i < nbTriangles; i++) {
         unsigned int index0 = triangles_array[3 * i];
         unsigned int index1 = triangles_array[3 * i + 1];
         unsigned int index2 = triangles_array[3 * i + 2];
         MeshVertex vertex0 = vertices[index0];
         MeshVertex vertex1 = vertices[index1];
         MeshVertex vertex2 = vertices[index2];
         Triangle currentTriangle = Triangle(vertex0.
            ↪ position*triangleScaling,vertex1.position*
            ↪ triangleScaling,vertex2.position*
            ↪ triangleScaling);
         RayTriangleIntersection currentIntersection =
            ↪ currentTriangle.getIntersection(ray);
         if(currentIntersection.intersectionExists &&
            ↪ currentIntersection.t < closestIntersection
            ↪ .t) {
             closestIntersection = currentIntersection;
             currentIntersection.tIndex = i;
         }
     }
     return closestIntersection;
}
```

Figure 4: Gestion de l'intersection rayon-triangle dans **Mesh.h**

Après un simple parcours des triangles du maillage considéré, voici ce que j'ai obtenu.

(a) Triangle simple

(b) Maillage d'un cylindre

(c) Maillage d'un blob

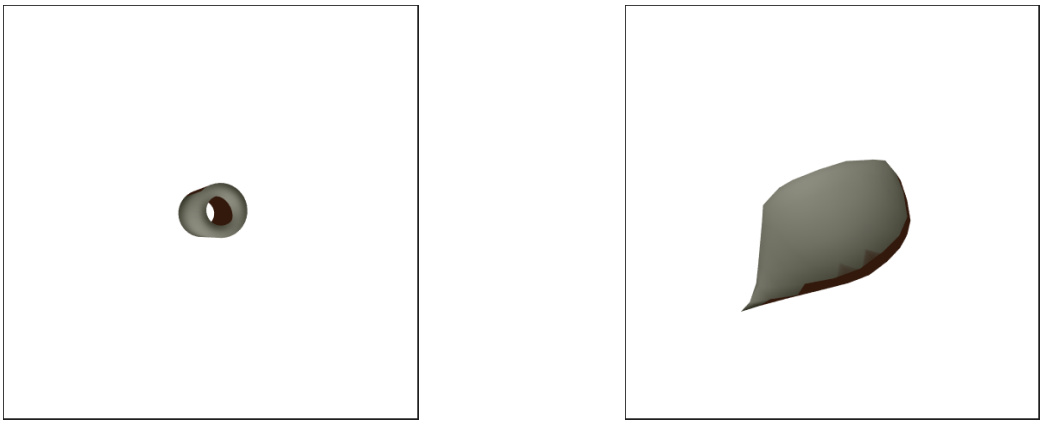
Figure 5: Rendus de maillages grâce au lancer de rayons

3 Interpolation des valeurs des sommets

Pour obtenir un rendu plus convaincant et réaliste, j'ai interpolé les valeurs des normales dans les maillages. Voici ce que j'ai rajouté dans la condition de la fonction **intersect**.

```
1 Vec3 interpolatedNormal = currentIntersection.w0 *
    ↪ vertex0.normal + currentIntersection.w1 * vertex1.
    ↪ normal + currentIntersection.w2 * vertex2.normal;
2 interpolatedNormal.normalize();
3 closestIntersection.normal = interpolatedNormal;
```

Figure 6: Interpolation des normales



(a) Maillage d'un cylindre

(b) Maillage d'un blob

Figure 7: Rendus de maillages grâce au lancer de rayons avec interpolation des normales

Je n'ai pas trouvé l'intêret d'interpoler les couleurs des sommets sachant que l'on définit un **Material** par maillage. Pour les coordonnées de textures (**uv**), j'implémenterai l'interpolation lorsque je commencerai à m'intéresser aux textures.

4 Surfaces réfléchissantes

Afin d'obtenir un objet totalement réfléchissant (donc un miroir), j'ai créé une condition sur le type du **Material** considéré dans **Scene.h**, en jouant avec l'appel récursif à **rayTraceRecursive**. Si le point tapé est de type miroir, alors il prend la couleur qu'ira taper son rayon réfléchi.

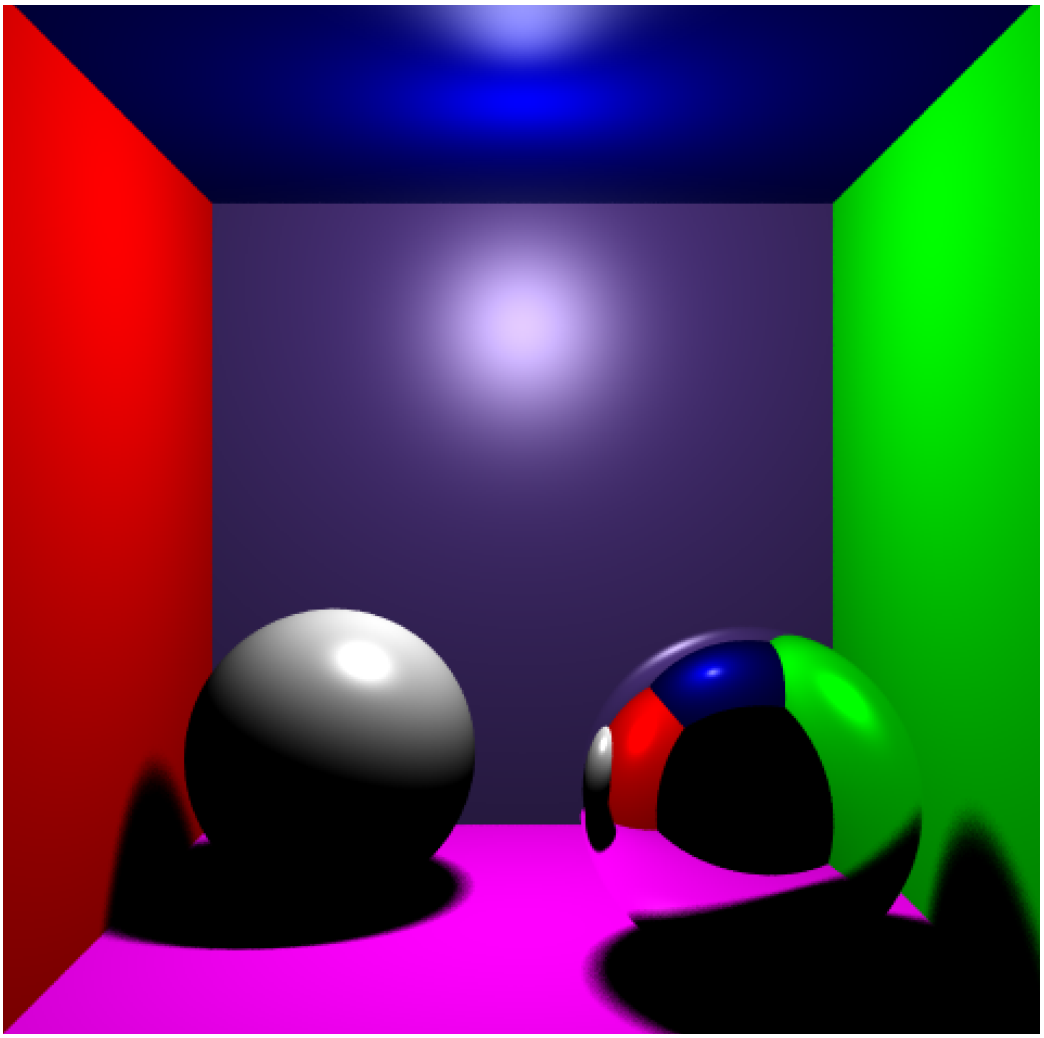


Figure 8: Sphère miroir

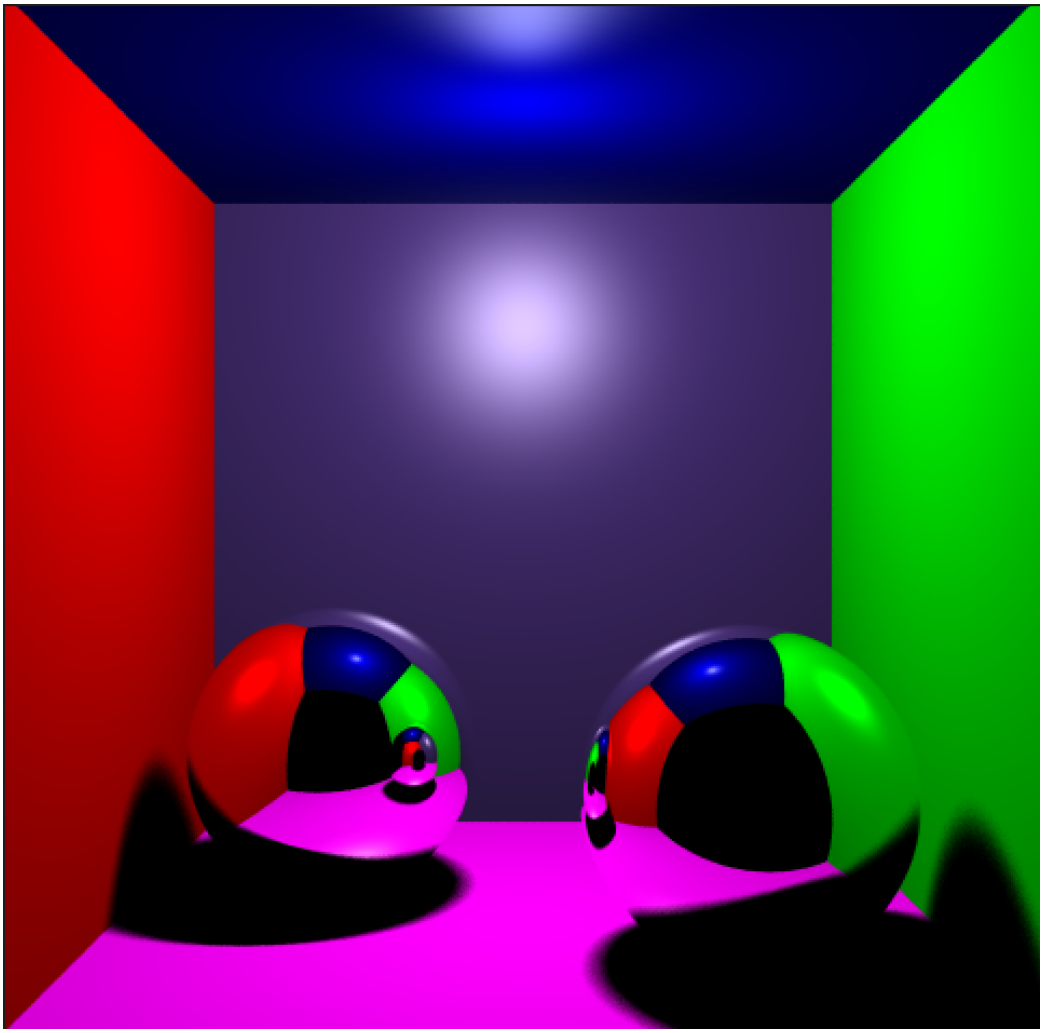


Figure 9: Deux sphères miroir

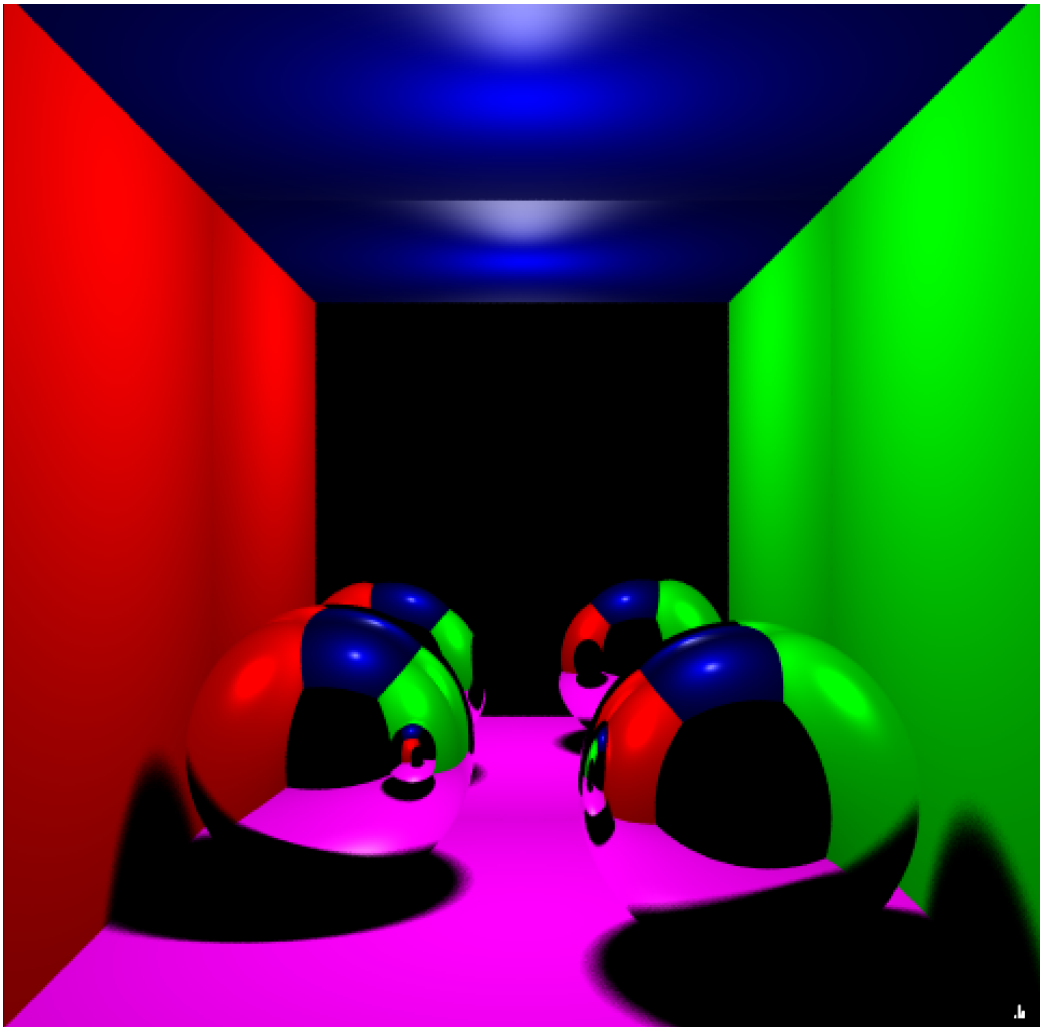


Figure 10: Deux sphères miroir et un mur miroir

5 Conclusion

La phase 3 du projet de ray tracing a marqué un progrès significatif avec l'intégration réussie d'intersections de maillages, d'interpolation de normales et de la création de sphères réfléchissantes. Ces avancées ont non seulement amélioré le réalisme visuel des rendus mais ont également posé les bases pour des développements plus poussés. L'expérience acquise et les défis relevés préparent le terrain pour l'étape suivante, qui consistera à améliorer les performances de l'application à l'aide d'une structure d'accélération, et qui mettra en place la réfraction des rayons.

Merci pour le temps et l'attention que vous avez consacrés à la lecture de ce compte-rendu.