Ray tracing - Compte rendu

HAI719I - Programmation 3D

Laëtitia LAPORTE

16/01/2026

Sommaire

1 Introduction **4**

2 Phase 1 **5**
2.1 Calcul des Intersections de Rayon dans la Scène 5
2.1.1 Implémentation de rayTrace() 5
2.1.2 Implémentation de rayTraceRecursive() 6
2.2 Intersection rayon-sphère 7
2.2.1 Théorie 7
2.2.2 Implémentation 7
2.2.3 Rendu 8
2.3 Intersection rayon-carré 10
2.3.1 Théorie 10
2.3.2 Implémentation 10
2.3.3 Rendus 12
2.4 Gestion des Résultats d'Intersection 13
2.4.1 Implémentation de computeIntersection() 13
2.4.2 Rendus 14

3 Phase 2 **15**
3.1 Modèle de Phong 15
3.1.1 Mise à jour de la fonction raytrace_recursive() 15
3.1.2 Réflexion Ambiante 15
3.1.3 Réflexion Diffuse 16
3.1.4 Réflexion Spéculaire 17
3.1.5 Rendus 18
3.2 Ombres dures 19
3.2.1 Rendus 19

4 Rendus inattendus notables des premières phases **22**

5 Phase 3 **23**
5.1 Ombres douces 23
5.2 Maillages 3D 29
5.2.1 Classe Line 29
5.2.2 Classe Triangle 30
5.2.3 Classe Mesh 33
5.2.4 Utilisation de l'ntersection rayon-maillage dans la classe Scene 35
5.2.5 Sphère réfléchissante 37

6 Phase 4 **40**
6.1 Sphère transparente 40
6.2 Structure d'accélération : KdTree 45
6.2.1 Principe de fonctionnement 45

6.2.2 Construction du kd-tree dans un espace 3D 46
6.2.3 Ajout d'un chronomètre 47
6.3 Comparaison des versions de KDTree 48
6.3.1 Contexte et objectif 48
6.3.2 V1 – Partition simple sur l'axe le plus long 48
6.3.3 V2 – Centre de gravité des primitives 49
6.3.4 V3 – Centres basés sur les AABB précalculées 50
6.3.5 Tableau de comparaison des versions du Kd-Tree 50
6.3.6 Choix de la meilleure version du Kd-Tree 51
6.3.7 Améliorations possibles pour le futur 51

7 Rendus inattendus notables des dernières phases 52

1 Introduction

Ce projet, réalisé dans le cadre du cours HAI719I - Programmation 3D, vise à implémenter un moteur de rendu par lancer de rayons (raytracing).

Dans une première phase, l'objectif est de calculer les intersections rayon-objet et de déterminer les couleurs des objets en fonction de leurs propriétés diffusives.

Une seconde phase consiste à intégrer des effets d'éclairage réalistes, comme le modèle de Phong et les ombres dures, afin d'améliorer la perception de profondeur et de réalisme dans la scène.

La troisième phase se concentre sur l'amélioration de la simulation du lancer de rayons. Elle inclut la gestion de maillages 3D, le calcul des intersections entre rayons et objets, ainsi que l'intégration d'une sphère réfléchissante qui simule les effets de réflexion dans la scène.

La dernière phase du projet ajoute des effets de réfraction et d'accélération. Elle consiste à créer une sphère transparente pour simuler la réfraction de la lumière à travers des objets transparents, tout en implémentant une structure d'accélération, le KdTree, afin d'optimiser les performances du rendu en réduisant le temps de calcul des intersections.

Ce document détaille les différentes étapes de développement, les principes théoriques appliqués, et les résultats obtenus pour chaque phase du projet.

Vous pouvez retrouver la page du dépôt git grâce au lien suivant :
https://github.com/Laeti016934/RayTracing

2 Phase 1

2.1 Calcul des Intersections de Rayon dans la Scène

L'objectif de cette partie est de déterminer, pour un rayon donné, l'objet de la scène avec lequel il intersecte en premier pour ainsi renvoyer sa couleur.

2.1.1 Implémentation de rayTrace()

La fonction rayTrace() renvoie la couleur perçue en suivant les rayons dans la scène et en calculant leurs interactions avec les objets grâce à l'appel de la fonction rayTraceRecursive().

Listing 1: Fonction rayTrace()

```
Vec3 rayTrace( Ray const & rayStart ) {
    int initialBounceCount = 10;

    Vec3 color = rayTraceRecursive(rayStart,
        initialBounceCount);

    return color;
}
```

2.1.2 Implémentation de rayTraceRecursive()

Dans cette première phase, la fonction rayTraceRecursive() utilise directement le résultat de l'intersection pour assigner une couleur à l'objet intersecté (basée sur sa couleur diffuse). S'il n'y a aucune intersection, alors la couleur noir est renvoyée.

Listing 2: Fonction rayTraceRecursive()

```
Vec3 rayTraceRecursive( Ray ray , int NRemainingBounces ) {
    Vec3 color;

    //Calcule de l'intersection avec la scene
    RaySceneIntersection raySceneIntersection =
        computeIntersection(ray);

    if(!raySceneIntersection.intersectionExists){
        color = Vec3(0.,0.,0.);
        return color;
    }

    if(raySceneIntersection.intersectionExists){
        if(raySceneIntersection.typeOfIntersectedObject ==
            0){
            color = meshes[raySceneIntersection.objectIndex
                ].material.diffuse_material;

        }else if(raySceneIntersection.
            typeOfIntersectedObject == 1){
            color = spheres[raySceneIntersection.objectIndex
                ].material.diffuse_material;

        }else if(raySceneIntersection.
            typeOfIntersectedObject == 2){
            color = squares[raySceneIntersection.objectIndex
                ].material.diffuse_material;

        }
    }

    return color;
}
```

2.2 Intersection rayon-sphère

2.2.1 Théorie

Équation rayon-sphère :

$$||x + c||^2 - r^2 = 0$$

Avec :

- x : équation du rayon $r(t) = o + td$
- c : centre de la sphère
- r : rayon de la sphère

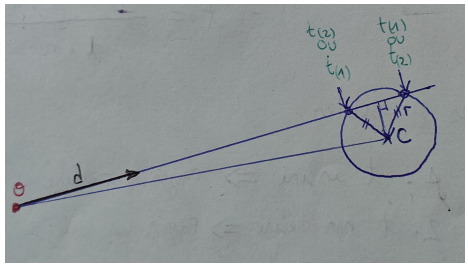

Figure 1: Schéma de l'intersection rayon-sphère

En remplaçant x par l'équation du rayon, on obtient une équation quadratique du second degré en t :

$$t^2 d \cdot d + 2td \cdot (o - c) + ||o - c||^2 - r^2 = 0$$

Avec :

- a = $d.d$
- b = $2.d.(o - c)$
- c = $||o - c||^2 - r^2$

2.2.2 Implémentation

Le code calcule les solutions de l'équation quadratique pour déterminer t, la distance au point d'intersection le long du rayon.
Tout d'abord, on calcule les coefficients.

Listing 3: Calcul des coefficients dans la fonction Intersect() de Sphere.h

```
float a = Vec3::dot(rayDirection, rayDirection);
float b = Vec3::dot(2 * rayDirection, (diff_Origin_Center));
float c = (diff_Origin_Center.norm() * diff_Origin_Center.
    norm()) - (m_radius * m_radius);
```

Ensuite, on analyse le discriminant $\Delta = b^2 - 4ac$.
Si :

- $\Delta > 0$: Deux intersections. Le plus petit $t > 0$ est choisi
- $\Delta = 0$: Une intersection tangente
- $\Delta < 0$: Aucune intersection

Enfin, si l'intersection existe, on calcule le point d'intersection $P = o + t.d$ ainsi que la normale $N = (P - c)/(||P - c||)$. Au contraire, si aucune intersection n'est trouvée, le booléen *intersectionExists* est à false.

Listing 4: Extrait de la fonction Intersect() de Sphere.h pour delta égale à 0

```
else if(delta == 0){
    //t = -b / 2a
    intersection.t = -b / (2 * a);
    intersection.intersectionExists = (intersection.t > 0);
    if (intersection.intersectionExists) {
        intersection.intersection = rayOrigin + intersection
            .t * rayDirection;
        intersection.normal = (intersection.intersection -
            m_center);
        intersection.normal.normalize();
    }
}
```

2.2.3 Rendu

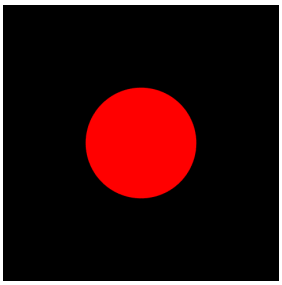

Figure 2: Rendu de la scène single_sphere

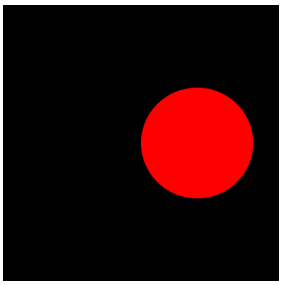

Figure 3: Rendu de la scène avec changement de position de l'objet

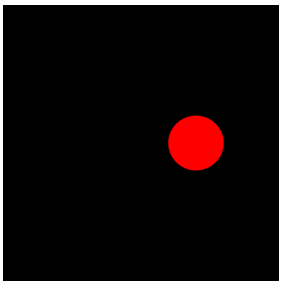

Figure 4: Rendu de la scène avec mise à l'échelle

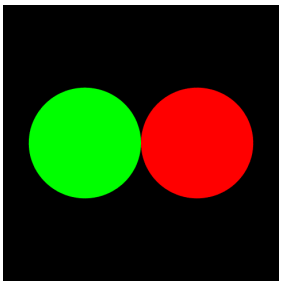

Figure 5: Rendu de la scène avec ajout d'une sphère

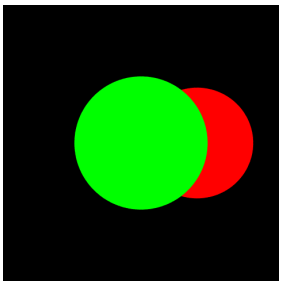

Figure 6: Rendu de la scène avec chevauchement des sphères

2.3 Intersection rayon-carré

2.3.1 Théorie

Équation rayon-plan :

$$x \cdot n - D = 0$$

Avec :

- x : point sur le plan
- n : vecteur normal au plan
- D : distance du plan par rapport à l'origine dans la direction n

Pour trouver le point d'intersection entre le rayon et le plan, on remplace x par l'équation du rayon. Ce qui permet de trouver la distance paramétrique t :

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

Avec o et d qui sont respectivement l'origine et la direction du rayon.

2.3.2 Implémentation

On commence par le calcul de la normale au plan.

Listing 5: Calcul de la normale au plan extrait de la fonction Intersect() de Square.h

```
//Largeur et hauteur du quad
Vec3 vec_width = (this->vertices[1].position - this->
    vertices[0].position);
Vec3 vec_height = (this->vertices[3].position - this->
    vertices[0].position);

//Normale du quad (perpendiculaire au plan)
Vec3 normal = Vec3::cross(vec_width , vec_height);
```

Cette normale est utilisée dans le calcul de t.

Listing 6: Calcul de t

```
//Distance t entre l'origine du rayon et le point d'
    intersection avec le plan du carre
float t = Vec3::dot(this->vertices[0].position - ray.origin
    (), normal) / Vec3::dot(ray.direction(), normal);
```

Ensuite, on fait les tests d'intersection avec le plan.

- Si $t < 0$: l'intersection est derrière la caméra donc pas d'intersection
- Sinon le point d'intersection est calculé ($P = o + td$)

Enfin, si le point d'intersection P existe, on vérifie qu'il soit bien compris dans le carré. Pour cela, on projette P dans les directions des vecteurs de la largeur et hauteur (respectivement proj_width et proj_height).
Et l'on vérifie que $0 < proj_width < width$ et que $0 < proj_height < height$.

Listing 7: Tests d'intersection et de limites du carré extrait de la fonction Intersect() dans Square.h

```
RaySquareIntersection intersect(const Ray &ray) const {
    //Intersection derriere la camera
    if(t < 0){
        intersection.intersectionExists = false;
    }
    //Intersection devant ou a l'origine de la camera
    if(t >= 0){
        Vec3 inter_pt = ray.origin() + t * ray.direction();
        Vec3 bottomLeftInterPt = inter_pt - this->vertices
            [0].position;

        float width = vec_width.length();
        float height = vec_height.length();

        float proj_width = Vec3::dot(bottomLeftInterPt,
            vec_width)/width;
        float proj_height = Vec3::dot(bottomLeftInterPt,
            vec_height)/height;

        if( 0 < proj_width && proj_width < width && 0 <
            proj_height && proj_height < height){
            intersection.intersectionExists = true;
            intersection.intersection = inter_pt;
            intersection.t = t;
            intersection.normal = normal;
            intersection.normal.normalize();
        }else{
            intersection.intersectionExists = false;
        }
    }
    return intersection;
```

2.3.3 Rendus

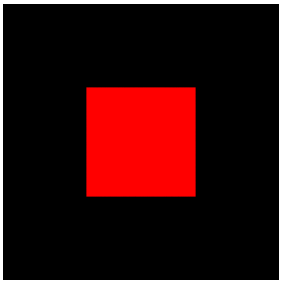

Figure 7: Rendu de la scène single_square

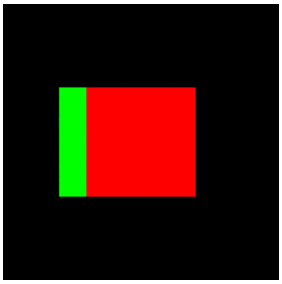

Figure 8: Rendu de la scène avec ajout d'un plan

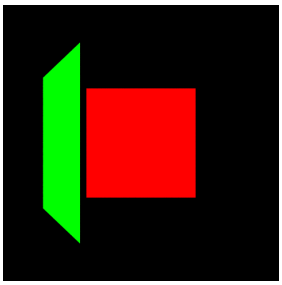

Figure 9: Rendu de la scène avec chanchement de position

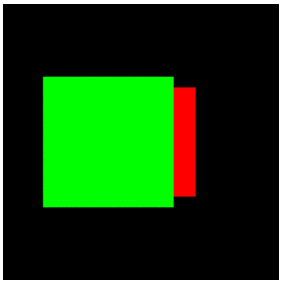

Figure 10: Rendu de la scène avec un second chanchement de position

2.4 Gestion des Résultats d'Intersection

2.4.1 Implémentation de computeIntersection()

Cette fonction itère sur tous les objets de la scène et calcule les intersections avec chaque type d'objet. Elle retourne une structure (RaySceneIntersection) contenant les informations sur l'intersection la plus proche.

Listing 8: Fonction computeIntersection()

```
RaySceneIntersection computeIntersection(Ray const & ray) {
        RaySceneIntersection result;

        for (int i = 0; i < meshes.size(); i++) {
            RayTriangleIntersection RSI = meshes[i].
                intersect(ray);
            if (RSI.intersectionExists && RSI.t < result.t)
                {
                result.rayMeshIntersection = RSI;
                result.intersectionExists = true;
                result.objectIndex = i;
                result.typeOfIntersectedObject = 0;
                result.t = RSI.t;
            }
        }

        for(int i = 0; i < spheres.size(); i++){
            RaySphereIntersection RSI = spheres[i].intersect
                (ray);
            if(RSI.intersectionExists == true && RSI.t <
                result.t ){
                result.raySphereIntersection = RSI;
                result.intersectionExists = true;
                result.objectIndex = i;
                result.typeOfIntersectedObject = 1;
                result.t = RSI.t;
            }
        }

        for (int i = 0; i < squares.size(); i++) {
            RaySquareIntersection RSI = squares[i].intersect
                (ray);
            if (RSI.intersectionExists && RSI.t < result.t)
                {
                result.raySquareIntersection = RSI;
                result.intersectionExists = true;
                result.objectIndex = i;
                result.typeOfIntersectedObject = 2;
                result.t = RSI.t;
            }
        }
```

```

        return result;
    }
```

2.4.2 Rendus

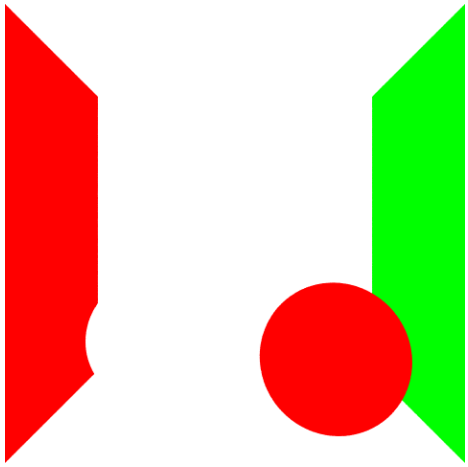

Figure 11: Rendu de la scène cornell_box avant les changements de couleur des objets

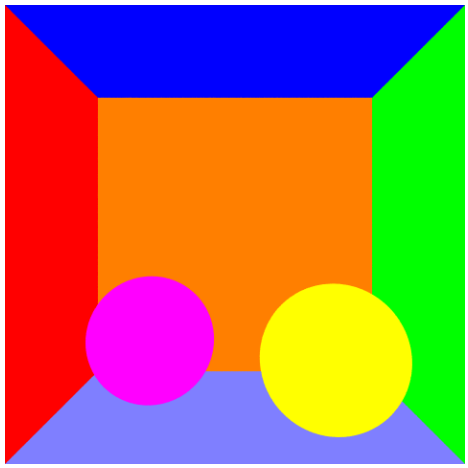

Figure 12: Rendu de la scène cornell_box après les changements de couleur des objets

3 Phase 2

Cette seconde phase a pour but de donner une apparence 3D aux objets de la scène grâce à l'implémentation de l'éclairage selon le modèle de Phong et des ombres, dans un premier temps sous leur forme dure.

3.1 Modèle de Phong

Ce modèle repose sur trois composantes principales : la réflexion ambiante, diffuse et spéculaire.

3.1.1 Mise à jour de la fonction raytrace_recursive()

Il faut effectuer la mise à jour de la fonction raytrace_recursive() pour inclure les effets d'illumination selon le modèle de Phong.

Listing 9: Mise à jour de la fonction rayTraceRecursive()

```
Vec3 rayTraceRecursive( Ray ray , int NRemainingBounces ) {
    Vec3 color;

    RaySceneIntersection raySceneIntersection =
        computeIntersection(ray);

    if(!raySceneIntersection.intersectionExists){
            color = Vec3(0.,0.,0.);
            return color;
    }

    if (raySceneIntersection.intersectionExists)
    {
        color = phong(raySceneIntersection);
    }

    return color;
}
```

3.1.2 Réflexion Ambiante

Le calcul de la réflexion ambiante est basé sur un coefficient global (Ka) et une lumière ambiante fixe définie par globalAmbientLight.

$$AmbientColor = Ka \times GlobalAmbientLight$$

Cette composante assure une luminosité minimale uniforme sur les objets, même sans source de lumière directe.

3.1.3 Réflexion Diffuse

La réflexion diffuse modélise l'éclairage distribué uniformément dans toutes les directions selon la loi de Lambert. La direction de la lumière est calculée en fonction de la position de la source lumineuse et du point d'intersection entre le rayon et l'objet.

$$Diffuse = I \times Kd \times (L \cdot N)$$

Où :

- I : intensité de la lumière diffuse
- Kd : coefficient de réflexion diffuse
- L : vecteur directionnel vers la lumière
- N : normale à la surface au point d'intersection

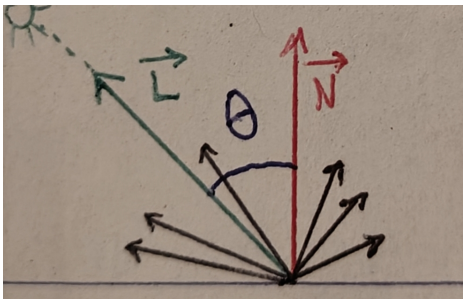

Figure 13: Schéma de la réflexion diffuse

3.1.4 Réflexion Spéculaire

La réflexion spéculaire dépend de la position de l'observateur et utilise le vecteur réfléchi (R) pour simuler la brillance sur la surface.

$$Specular = I \times Ks \times (R \cdot V)^{shininess}$$

Où :

- I : intensité de la lumière spéculaire
- Ks : coefficient de réflexion spéculaire
- R : vecteur de réflexion spéculaire
- V : vecteur directionnel de la vue (vers l'observateur)
- shininess : coefficient qui contrôle la taille et l'intensité du point lumineux spéculaire

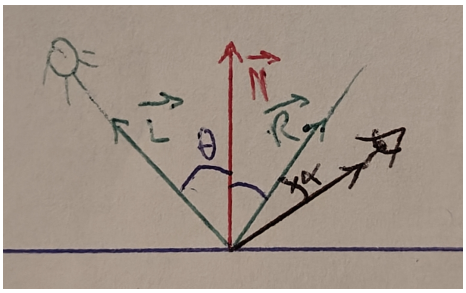

Figure 14: Schéma de la réflexion spéculaire

3.1.5 Rendus

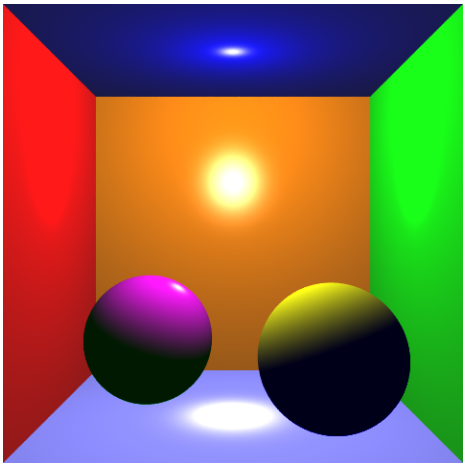

Figure 15: Rendu du modèle Phong

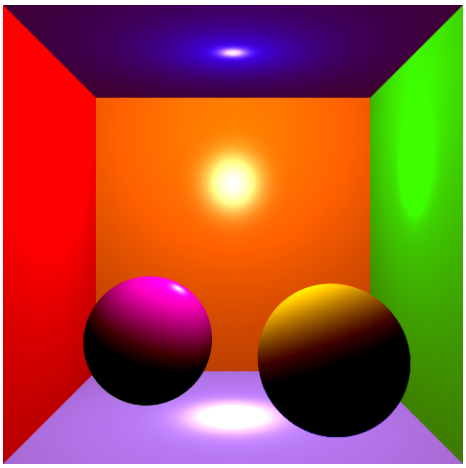

Figure 16: Rendu du modèle Phong avec une lumière rouge

3.2 Ombres dures

Une vérification a été ajoutée, à la fonction phong(), pour déterminer si un point d'intersection est ombragé en lançant un rayon secondaire (shadowRay) vers chaque source lumineuse. Si un objet bloque la lumière avant d'atteindre le point considéré, les contributions diffuse et spéculaire sont ignorées.

Listing 10: Vérification des ombres extrait de la fonction phong()

```
// Verifie si le point P est dans l'ombre de la lumiere
Ray shadowRay(P + N * 0.001, L); // Rayon legerement decale
    pour eviter l'autointersection
RaySceneIntersection shadowIntersection =
    computeIntersection(shadowRay);

if (shadowIntersection.intersectionExists &&
    shadowIntersection.t < (light.pos - P).length()) {
    // Si un objet bloque la lumiere, ignorer les
        contributions diffuse et speculaire
    continue;
}
```

3.2.1 Rendus

Figure 17: Rendu du modèle Phong avec ombres dures

J'ai voulu essayer de rajouter une seconde source de lumière et de modifier sa position pour observer les résultats obtenus.

Figure 18: Rendu du modèle Phong avec ombres dures avec ajout d'une seconde lumière, pos(-1.5, 1.0, 0.0)

Figure 19: Rendu du modèle Phong avec ombres dures avec ajout d'une seconde lumière, pos(-1.9, 1.5, -1.5)

Figure 20: Rendu du modèle Phong avec ombres dures avec ajout d'une seconde lumière, pos(-1.9, -1.9, 1.9)

Figure 21: Rendu du modèle Phong avec ombres dures avec ajout d'une seconde lumière rouge

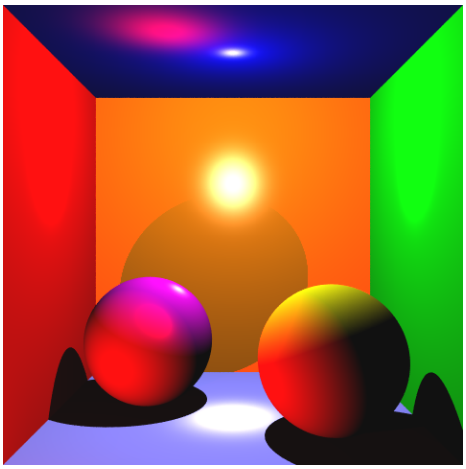

Figure 22: Rendu du modèle Phong avec ombres dures avec ajout d'une seconde lumière rouge, pos(-1.9, -1.9, 1.9)

4 Rendus inattendus notables des premières phases

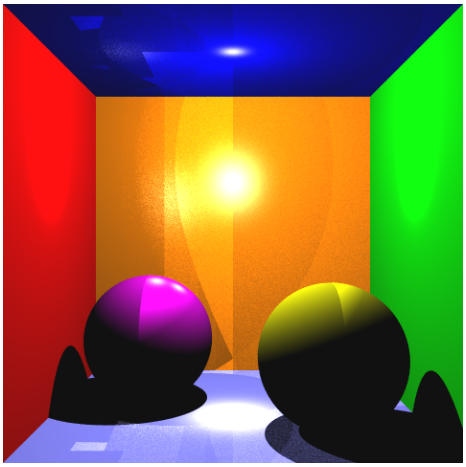

Figure 23: Rendu inattendus du modèle Phong avec ombres dures et deux sources de lumières

Ce rendu est sûrement dû à la position de la deuxième source de lumière (-2.0, 1.0, 0.0). La lumière doit être dans le mur de gauche.

5 Phase 3

5.1 Ombres douces

Dans cette partie, j'ai intégré des ombres douces au moteur de raytracing en utilisant des lumières étendues. Cela permet de simuler des transitions d'ombre progressives, comme celles observées dans le monde réel, où une source lumineuse a une surface non négligeable.

Ces ombres sont créées en tenant compte des zones d'ombre partielle et des zones totalement éclairées. Pour simuler ce phénomène, j'ai utilisé un échantillonnage sur la surface de la lumière étendue (de type Quad) et calculé les contributions lumineuses pour chaque échantillon.

Voici les étapes de la création des ombres douces :

1. **Ajout de lumières étendues :**
   Dans la scène cornell_box, j'ai modifié le type de la lumière par LighType_Quad et appelé la procédure nommée initialize_quad_light() qui initialise le paramètre quad de type Mesh de la structure Light.

Listing 11: Modification du Type de la lumière dans la scène cornell_box

```
    void setup_cornell_box(){
        meshes.clear();
        spheres.clear();
        squares.clear();
        lights.clear();

    // Lumiere de type Quad
    {
        lights.resize(lights.size() + 1);
        Light &light = lights.back();
        light.pos = Vec3(0., 1.5, 0.);
        light.type = LightType_Quad;
        light.material = Vec3(1, 1, 1);
        light.isInCamSpace = false;

        // Initialiser la geometrie du quad
        initialize_quad_light(light, 4.0f, 4.0f); //
            Taille 4x4
    }
    //...
```

Listing 12: Procédure initialize_quad_light()

```
void initialize_quad_light(Light &light, float width, float height) {
    if (light.type != LightType_Quad) {
        std::cerr << "La lumiere doit etre de type Quad pour utiliser cette fonction." << std::endl;
        return;
    }

    float halfWidth = width * 0.5f;
    float halfHeight = height * 0.5f;

    light.quad.vertices.clear();
    light.quad.vertices.push_back(MeshVertex(light.pos + Vec3(-halfWidth, -halfHeight, 0), Vec3(0, 0, 1))); // Bas gauche
    light.quad.vertices.push_back(MeshVertex(light.pos + Vec3(halfWidth, -halfHeight, 0), Vec3(0, 0, 1)));  // Bas droit
    light.quad.vertices.push_back(MeshVertex(light.pos + Vec3(halfWidth, halfHeight, 0), Vec3(0, 0, 1)));   // Haut droit
    light.quad.vertices.push_back(MeshVertex(light.pos + Vec3(-halfWidth, halfHeight, 0), Vec3(0, 0, 1)));  // Haut gauche

    light.quad.triangles.clear();
    light.quad.triangles.push_back(MeshTriangle(0, 1, 2)); // Triangle 1
    light.quad.triangles.push_back(MeshTriangle(0, 2, 3)); // Triangle 2

    light.quad.build_arrays();
}
```

2. **Échantillonnage sur la lumière étendue :**
La fonction samplePointOnLight() échantillonne aléatoirement des points sur la surface de la lumière.

Listing 13: Fonction samplePointOnLight()

```
        Vec3 samplePointOnLight(const Light &light) {
        if (light.type != LightType_Quad) {
            std::cerr << "Echantillonnage impossible : la lumiere n'est pas de type Quad." << std::endl;
            return light.pos; // Retourne simplement la position pour les autres types de lumiere.
        }

        // Generer des coordonnees barycentriques aleatoires
        float u = static_cast<float>(rand()) / RAND_MAX;
        float v = static_cast<float>(rand()) / RAND_MAX;

        // Assurer que u + v <= 1 pour le premier triangle
        if (u + v > 1.0f) {
            u = 1.0f - u;
            v = 1.0f - v;
        }

        const Mesh &quad = light.quad;

        // Calculer la position interpolee dans le triangle (0, 1, 2)
        Vec3 p0 = quad.vertices[quad.triangles[0][0]].position;
        Vec3 p1 = quad.vertices[quad.triangles[0][1]].position;
        Vec3 p2 = quad.vertices[quad.triangles[0][2]].position;

        return (1 - u - v) * p0 + u * p1 + v * p2;
    }
```

3. **Intégration dans le modèle d'éclairage Phong :**
   La fonction samplePointOnLight() est utilisée dans le modèle d'éclairage comme suit.

Listing 14: Intégration de samplePointOnLight() dans le modèle de Phong

```
Vec3 phong(RaySceneIntersection intersectionObjet) {
    //Initialisation

    //...

    // Parcours des sources de lumiere
    for (int i_Light = 0; i_Light < lights.size();
        i_Light++) {

        //...

        // Calcul des contributions pour chaque type
            d'objet

        //...

        // Pour les lumieres de type Quad
        if (light.type == LightType_Quad) {
            Vec3 summedColor(0.0f, 0.0f, 0.0f);
            int numSamples = 16; // Nombre d'
                echantillons pour la lumiere etendue

            // Moyenne des echantillons pour simuler
                des ombres douces
            for (int i = 0; i < numSamples; ++i) {
                // Echantillonnage d'un point sur la
                    lumiere
                Vec3 sampledLightPosition =
                    samplePointOnLight(light);

                // Calcul de la direction de la
                    lumiere
                L = (sampledLightPosition - P);
                L.normalize();

                // Calcul de la direction vers la
                    camera
                V = P * -1;
                V.normalize();

                // Calcul de la direction reflechie
                R = (2. * Vec3::dot(N, L) * N - L);
                R.normalize();

```

```
                    // Verifie l'ombre pour ce point echantillonne
                    Ray shadowRay(P + N * 0.001, L); // Rayon legerement decale pour eviter l'autointersection
                    RaySceneIntersection shadowIntersection = computeIntersection(shadowRay);

                    if (shadowIntersection.intersectionExists &&
                        shadowIntersection.t < (sampledLightPosition - P).length()) {
                        // Si un objet bloque la lumiere, ignorer cette contribution
                        continue;
                    }

                    // Ajout de la contribution diffuse et speculaire pour ce point echantillonne

                    //...

                    }
                }

                // Moyenne des contributions de la lumiere etendue

                //...

                // Ajout de la couleur calculee a la couleur finale

                //...

    }
```

Voici les résultats obtenus :

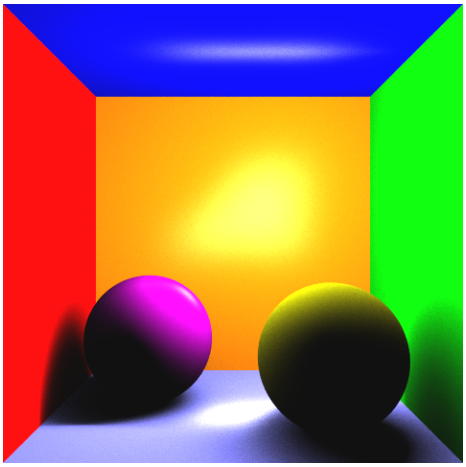

Figure 24: Rendu du modèle Phong avec ombres douces

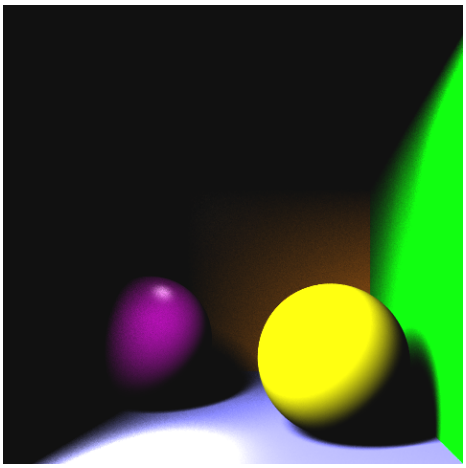

Figure 25: Rendu du modèle Phong avec ombres douces en déplaçant la lumière

5.2 Maillages 3D

Pour calculer l'intersection d'un rayon avec un maillage, il est essentiel de comprendre la structure de ce maillage. Un maillage est constitué d'un ensemble de triangles, et chaque triangle est formé de lignes.

Par conséquent, la première étape consiste à définir la classe **Line.h**, qui représente les lignes. Ensuite, nous définissons la classe **Triangle.h**, qui utilisera des objets de type Line, suivie de la création de la classe **Mesh.h** pour gérer l'ensemble du maillage. Enfin, la classe **Scene.h** permettra d'intégrer et de gérer l'ensemble du système, incluant le calcul des intersections.

5.2.1 Classe Line

La fonction **project()** permet de projeter un point **p** sur une droite définie par une origine **m_origin** et une direction **m_direction**.

Listing 15: Fonction project()

```
Vec3 project( Vec3 const & p ) const {
    Vec3 result;

    Vec3 OP = p - m_origin;
    float t = Vec3::dot(OP, m_direction);

    result = m_origin + t * m_direction;

    return result;
}
```

La fonction **squareDistance()** calcule la distance au carré entre un point **p** et la projection de ce point sur une droite définie par **m_origin** et **m_direction**.

Listing 16: Fonction squareDistance()

```
float squareDistance( Vec3 const & p ) const {
    float result;
    Vec3 projectedPoint = project(p);

    result = (p - projectedPoint).length();

    return result;
}
```

5.2.2 Classe Triangle

La fonction **projectOnSupportPlane()** projette un point **p** sur un plan support défini par un triangle, en utilisant un procédé de projection sur une droite liée à la normale du plan.

Listing 17: Fonction projectOnSupportPlane()

```
Vec3 projectOnSupportPlane( Vec3 const & p ) const {
    Vec3 result;

    Line line(m_c[0], m_normal);
    result = line.project(p);

    return result;
}
```

La fonction **squareDistanceToSupportPlane()** calcule la distance entre un point **p** et un plan support défini par un triangle, mais elle retourne la distance au carré plutôt que la distance réelle.

Listing 18: Fonction squareDistanceToSupportPlane()

```
float squareDistanceToSupportPlane( Vec3 const & p ) const {
    float result;

    Vec3 projection = projectOnSupportPlane(p);
    result = (p - projection).length();

    return result;
}
```

La fonction **isParallelTo()** vérifie si la normale d'un triangle est parallèle à la direction d'une ligne donnée **L**.

Listing 19: Fonction isParallelTo()

```
bool isParallelTo( Line const & L ) const {
    bool result;

    float dotProduct = Vec3::dot(m_normal, L.direction());

    result = std::abs(dotProduct) < 1e-6f;

    return result;
}
```

La fonction **getIntersectionPointWithSupportPlane()** calcule le point d'intersection entre une ligne **L** et le plan support défini par un triangle.

Listing 20: Fonction getIntersectionPointWithSupportPlane()

```
Vec3 getIntersectionPointWithSupportPlane( Line const & L )
    const {
    Vec3 result;

    if (isParallelTo(L)) {
        std::cerr << "La␣ligne␣est␣parallele␣au␣plan." <<
            std::endl;
        return Vec3(); // Retourne un point nul
    }

    float t = Vec3::dot(m_c[0] - L.origin(), m_normal) /
        Vec3::dot(L.direction(), m_normal);

    result = L.origin() + t * L.direction();

    return result;
}
```

La fonction **computeBarycentricCoordinates()** calcule les coordonnées barycentriques d'un point **p** par rapport à un triangle défini par ses trois sommets **m_c[0]**, **m_c[1]**, et **m_c[2]**.

Listing 21: Fonction computeBarycentricCoordinates()

```
void computeBarycentricCoordinates( Vec3 const & p , float &
     u0 , float & u1 , float & u2 ) const {
    // Vecteurs de l'element du triangle
    Vec3 v0 = m_c[1] - m_c[0];
    Vec3 v1 = m_c[2] - m_c[0];
    Vec3 v2 = p - m_c[0];

    // Calcul du determinant de la matrice pour l'inverser
        et obtenir les barycentriques
    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;

    // Calcul des coordonnees barycentriques
    u1 = (d11 * d20 - d01 * d21) / denom;
    u2 = (d00 * d21 - d01 * d20) / denom;
    u0 = 1.0f - u1 - u2;
}
```

La fonction **getIntersection()** détermine si un rayon **ray** intersecte un triangle défini par ses trois sommets **m_c[0]**, **m_c[1]**, et **m_c[2]**, et calcule les coordonnées barycentriques de l'intersection. Elle renvoie un objet **RayTriangleIntersection** qui contient les informations sur l'intersection, si elle existe.

Listing 22: Fonction getIntersection()

```
RayTriangleIntersection getIntersection( Ray const & ray ) const {
    RayTriangleIntersection result;
    result.intersectionExists = false;

    // 1) check that the ray is not parallel to the triangle:
    float denominator = Vec3::dot(m_normal, ray.direction());
    if (fabs(denominator) < 1e-6f) {
        return result; // Pas d'intersection, rayon parallele au triangle
    }

    // 2) check that the triangle is "in front of" the ray:
    Vec3 edgeToRayOrigin = m_c[0] - ray.origin();
    float t = Vec3::dot(m_normal, edgeToRayOrigin) / denominator;

    if (t < 1e-6f) { // Ajouter une petite tolerance
        return result; // Intersection derriere ou trop proche
    }

    // 3) check that the intersection point is inside the triangle:
    // CONVENTION: compute u,v such that p = w0*c0 + w1*c1 + w2*c2, check that 0 <= w0,w1,w2 <= 1
    float u0, u1, u2;
    Vec3 intersectionPoint = ray.origin() + t * ray.direction();
    computeBarycentricCoordinates(intersectionPoint, u0, u1, u2);

    // Verifier que le point est a l'interieur du triangle
    if (u0 < 0 || u1 < 0 || u2 < 0) {
        return result; // Intersection en dehors du triangle
    }

    // 4) Finally, if all conditions were met, then there is an intersection! :
    result.intersectionExists = true;
    result.t = t;
```

```
    result.w0 = u0;
    result.w1 = u1;
    result.w2 = u2;
    result.intersection = intersectionPoint;
    result.normal = m_normal;

    return result;
}
```

5.2.3 Classe Mesh

La fonction **intersect()** effectue une recherche d'intersection entre un rayon (**ray**) et un ensemble de triangles (**triangles**). Elle retourne l'intersection la plus proche sous forme d'un objet **RayTriangleIntersection**.

Listing 23: Fonction intersect()

```
RayTriangleIntersection intersect( Ray const & ray ) const {
    RayTriangleIntersection closestIntersection;
    closestIntersection.intersectionExists = false;
    closestIntersection.t = FLT_MAX;
    closestIntersection.w0 = closestIntersection.w1 =
        closestIntersection.w2 = 0.0f;
    closestIntersection.tIndex = -1;  // Indique qu'aucun
        triangle n'est encore selectionne.
    closestIntersection.intersection = Vec3(0, 0, 0);
    closestIntersection.normal = Vec3(0, 0, 0);

    float triangleScaling = 1.000001f; // Echelle pour
        eviter les problemes numeriques.
    Vec3 rayDirection = ray.direction();
    Vec3 rayOrigin = ray.origin();

    // Parcourir chaque triangle
    for (size_t i = 0; i < triangles.size(); ++i) {
        const auto &triangle = triangles[i];
        // Recuperer les positions des sommets
        const Vec3 &v0 = vertices[triangle[0]].position *
            triangleScaling;
        const Vec3 &v1 = vertices[triangle[1]].position *
            triangleScaling;
        const Vec3 &v2 = vertices[triangle[2]].position *
            triangleScaling;

        // Calcule les aretes du triangle
        Vec3 edge1 = v1 - v0;
        Vec3 edge2 = v2 - v0;

        // Calcul du determinant
        Vec3 h = Vec3::cross(rayDirection, edge2);
```

```
        float a = Vec3::dot(edge1, h);

        // Si le determinant est proche de 0, le rayon est parallele au triangle
        if (fabs(a) < 1e-6f) continue;

        float f = 1.0f / a;
        Vec3 s = rayOrigin - v0;
        float u = f * Vec3::dot(s, h);

        // Verifie si u est en dehors de la plage [0, 1]
        if (u < 0.0f || u > 1.0f) continue;

        Vec3 q = Vec3::cross(s, edge1);
        float v = f * Vec3::dot(rayDirection, q);

        // Verifie si v est en dehors de la plage [0, 1] ou si u + v > 1
        if (v < 0.0f || u + v > 1.0f) continue;

        // Calcul de t pour l'intersection
        float t = f * Vec3::dot(edge2, q);

        // Verifie si l'intersection est devant le rayon
        if (t > 1e-6f && t < closestIntersection.t) {
            closestIntersection.t = t;
            closestIntersection.intersectionExists = true;
            closestIntersection.tIndex = i;
            closestIntersection.intersection = ray.origin() + ray.direction() * t;

            // Interpole les normales et autres attributs
            closestIntersection.normal =
                (1.0f - u - v) * vertices[triangle[0]].normal +
                u * vertices[triangle[1]].normal +
                v * vertices[triangle[2]].normal;

            closestIntersection.normal.normalize();

            closestIntersection.w0 = 1.0f - u - v;
            closestIntersection.w1 = u;
            closestIntersection.w2 = v;
        }
    }

    return closestIntersection;
}
```

5.2.4 Utilisation de l'ntersection rayon-maillage dans la classe Scene

La fonction **intersect()** de la classe Mesh est ensuite utilisée dans la fonction **computeIntersection()** de la classe Scene.
Il suffit ensuite de charger le maillage dans une scène comme suit.

Listing 24: Initialisation et chargement du fichier star.off dans la scène

```
{
    meshes.resize(meshes.size() + 1);
    Mesh &mesh = meshes.back();

    // Chemin vers le fichier OFF
    std::string offFilePath = "img/star.off";

    // Chargement du fichier OFF
    mesh.loadOFF(offFilePath);

    // Recentre et met a l'echelle le maillage
    mesh.centerAndScaleToUnit();

    // Recalcule les normales des sommets
    mesh.recomputeNormals();

    // Hauteur minimale du maillage (axe Y)
    float minY = std::numeric_limits<float>::max();
    for (const auto &vertex : mesh.vertices) {
        if (vertex.position[1] < minY) {
            minY = vertex.position[1];
        }
    }

    // Translation verticale pour aligner le maillage sur le
        sol (y = 0)
    Vec3 translation(0.0, -minY - 1.8, 0.0);
    for (auto &vertex : mesh.vertices) {
        vertex.position += translation;
    }

    std::cout << "Maillage␣charge␣" << offFilePath << std::
        endl;
}
```

Figure 26: Rendu du maillage star.off

5.2.5 Sphère réfléchissante

Pour que la sphère soit réfléchissante tel un miroir, j'ai modifié la fonction rayTraceRecursive() pour qu'elle prenne en compte le type de matériaux existants (ici Material_Mirror, et plus tard, Material_Glass pour la sphère transparente).

Voici les principales étapes :

1. **Accès aux propriétés de la sphère**
   - **spheres[raySceneIntersection.objectIndex]** permet de récupérer la sphère concernée
   - On extrait son matériau **sph.material** pour vérifier s'il est du type Material_Mirror
2. **Calcul de la réflexion**

   Celle-ci est déterminée par la formule :

   $$R = I - 2 \cdot (I \cdot N) \cdot N$$

   Où :

   - **I**, la direction du rayon incident
   - **N**, la normale à la surface au point d'intersection
   - **R**, la direction du rayon réfléchi
3. **Prévention des auto-intersections**

   Le point $P + N \cdot 0.001$ décale légèrement le rayon réfléchi pour éviter qu'il réintersecte immédiatement la surface.
4. **Appel récursif**

   Un nouveau rayon réfléchi est lancé (**reflectedRay**) avec une profondeur de récursion réduite (NRemainingBounces - 1).
5. **Ajout de la couleur réfléchie**

   La couleur réfléchie calculée est pondérée par le matériau spéculaire de la sphère (**sph.material.specular_material**).
6. **Clamp de la couleur**

   On restreint la valeur de la couleur dans l'intervalle [0, 1] pour éviter des dépassements numériques.

Listing 25: Gestion du matériau réfléchissant dans rayTraceRecursive()

```
    Sphere &sph = spheres[raySceneIntersection.objectIndex];

    Vec3 P = raySceneIntersection.raySphereIntersection.
        intersection;
    Vec3 N = raySceneIntersection.raySphereIntersection.
        normal;
    Vec3 I = ray.direction();

    if (sph.material.type == Material_Mirror) {

        Vec3 R = I - 2.0f * Vec3::dot(I, N) * N;

        Ray reflectedRay(P + N * 0.001f, R);

        Vec3 reflectedColor = rayTraceRecursive(reflectedRay
            , NRemainingBounces - 1);

        color[0] += reflectedColor[0] * sph.material.
            specular_material[0];
        color[1] += reflectedColor[1] * sph.material.
            specular_material[1];
        color[2] += reflectedColor[2] * sph.material.
            specular_material[2];

        color = Vec3::clamp(color, 0.0f, 1.0f);
    }
```

Figure 27: Rendu de la sphère réfléchissante

6 Phase 4

6.1 Sphère transparente

Pour donner cet effet de transparence à la sphère, j'ai modifié la fonction rayTraceRecursive() pour qu'elle prenne en compte le type de matériaux existants (ici Material_Glass).

Voici les étapes principales :

1. **Propriétés de la sphère**

   Comme pour le miroir, on récupère la sphère et ses propriétés (**sph.material**).

2. **Indices de réfraction**

   Deux indices sont définis :

   - **n1** : Indice de réfraction du milieu incident (air, n1=1.0)
   - **n2** : Indice de réfraction du matériau de la sphère (**sph.material.index_medium**).

3. **Gestion des rayons sortants**

   Si $I \cdot N > 0$ (le rayon sort de la sphère) :

   - Les indices **n1** et **n2** sont échangés
   - La normale **N** est inversée

4. **Calcul des coefficients de Fresnel**

   - La réflexion et la réfraction dépendent des coefficients de Fresnel. Ceux-ci déterminent la proportion d'énergie réfléchie (Rs) et réfractée (T)
   - Si $sin^2\theta_t > 1$, il y a réflexion totale interne

5. **Calcul du rayon réfléchi**

   Identique à celui du miroir. Un rayon réfléchi est calculé et lancé récursivement.

6. **Calcul du rayon réfracté** (s'il n'y a pas réflexion totale interne)

   La direction du rayon réfracté est donnée par la loi de Snell :

   $$T = (I - N \cdot cos\theta_i) \cdot \frac{n_1}{n_2} - N \cdot cos\theta_t$$

   Où $cos\theta_t = \sqrt{1 - sin\theta_t}$

7. **Combinaison des couleurs**

   La couleur finale est une interpolation entre :

   - La couleur réfléchie $fresnelReflectance \cdot reflectedColor$
   - La couleur réfractée $(1 - fresnelReflectance) \cdot refractedColor$

8. **Clamp final**

   Comme pour le miroir, la couleur est restreinte dans [0, 1].

Listing 26: Gestion du matériau transparent dans rayTraceRecursive()

```
if (sph.material.type == Material_Glass) {
    Vec3 P = raySceneIntersection.raySphereIntersection.intersection; // Point d'intersection
    Vec3 N = raySceneIntersection.raySphereIntersection.normal; // Normale
    Vec3 I = ray.direction(); // Direction incidente

    // Indices de refraction
    float n1 = 1.0f; // Air
    float n2 = sph.material.index_medium; // Indice du verre

    // Si le rayon est sortant, inverser les indices et la normale
    if (Vec3::dot(I, N) > 0.0f) {
        std::swap(n1, n2);
        N = N * -1.0f; // Inverser manuellement la normale
    }

    // Calcul des coefficients de Fresnel
    float cosI = fabs(Vec3::dot(I, N));
    float sinT2 = (n1 / n2) * (n1 / n2) * (1.0f - cosI * cosI);
    float fresnelReflectance;

    if (sinT2 > 1.0f) {
        // Reflexion totale interne
        fresnelReflectance = 1.0f;
    } else {
        float cosT = sqrt(1.0f - sinT2);
        float Rs = pow((n1 * cosI - n2 * cosT) / (n1 * cosI + n2 * cosT), 2);
        float Rp = pow((n1 * cosT - n2 * cosI) / (n1 * cosT + n2 * cosI), 2);
        fresnelReflectance = (Rs + Rp) / 2.0f;
    }

    // Rayon reflechi
    Vec3 R = I - 2.0f * Vec3::dot(I, N) * N; // Direction reflechie
    Ray reflectedRay(P + N * 0.001f, R); // Deplacement pour eviter les auto-intersections
    Vec3 reflectedColor = rayTraceRecursive(reflectedRay, NRemainingBounces - 1);

    // Rayon refracte (si pas de reflexion totale interne)
    Vec3 refractedColor(0.0f, 0.0f, 0.0f);
    if (fresnelReflectance < 1.0f) {
        Vec3 T = (I - N * cosI) * (n1 / n2) - N * sqrt(1.0f
```

```
            - sinT2); // Direction refractee
        Ray refractedRay(P - N * 0.001f, T); // Deplacement leger vers l'interieur
        refractedColor = rayTraceRecursive(refractedRay, NRemainingBounces - 1);
    }

    // Combiner les couleurs
    color = fresnelReflectance * reflectedColor + (1.0f - fresnelReflectance) * refractedColor;
    color = Vec3::clamp(color, 0.0f, 1.0f); // Eviter les depassements
}
```

Figure 28: Rendu de la sphère transparente sans améliorations

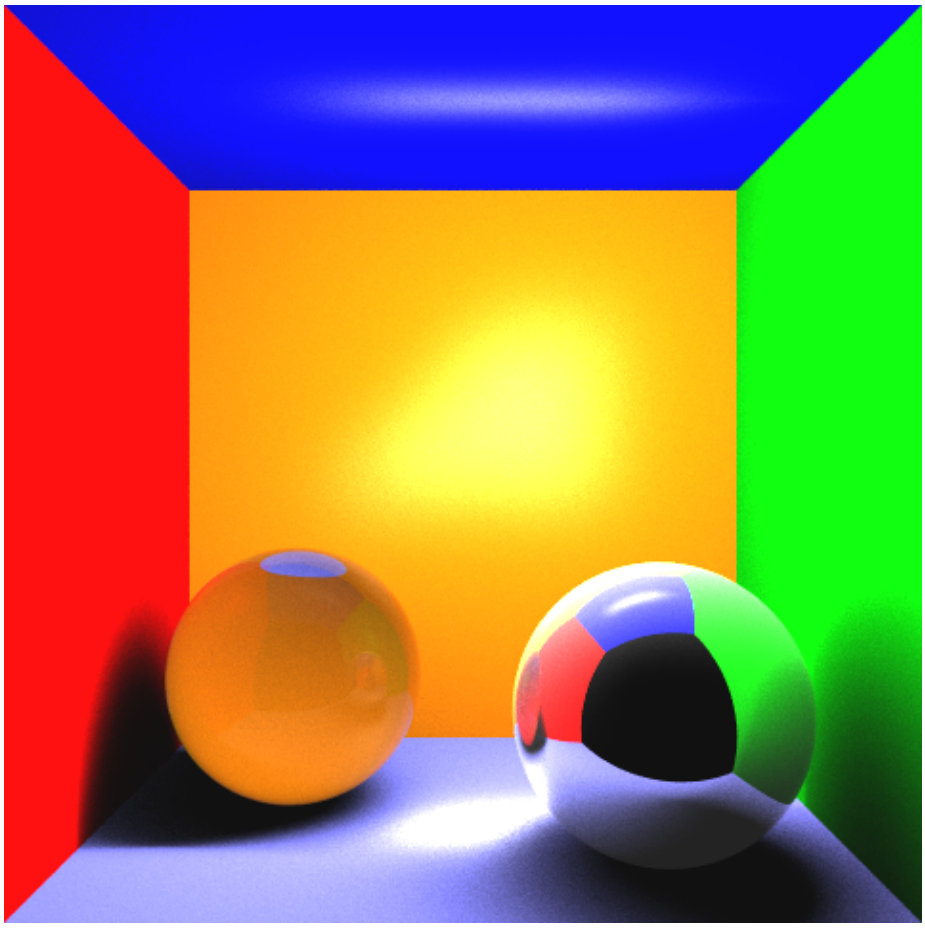

Figure 29: Rendu de la sphère transparente avec améliorations

6.2 Structure d'accélération : KdTree

J'ai d'abord fait mes recherches pour un environnement 2D afin de mieux comprendre le principe du KdTree avant de passer dans un environnement 3D. Voici ce que j'ai appris.

6.2.1 Principe de fonctionnement

Le kd-tree est une structure d'arbre binaire où chaque nœud divise l'espace selon un plan orthogonal à l'un des axes principaux (X, Y ou Z). Cette division permet de réduire le nombre d'objets à tester lors de la recherche d'intersections.

1. **Construction**
   - Les objets de la scène sont organisés hiérarchiquement selon leur position dans l'espace
   - À chaque nœud, l'espace est divisé en deux sous-espaces par un plan de séparation
   - Les nœuds feuille contiennent les objets d'un sous-espace, tandis que les nœuds internes contiennent des informations sur le plan de séparation
2. **Traversée**
   - Lorsqu'un rayon traverse l'arbre, seules les régions pertinentes (nœuds dont les boîtes englobantes intersectent le rayon) sont explorées
   - Cela réduit considérablement le nombre de tests d'intersection nécessaires

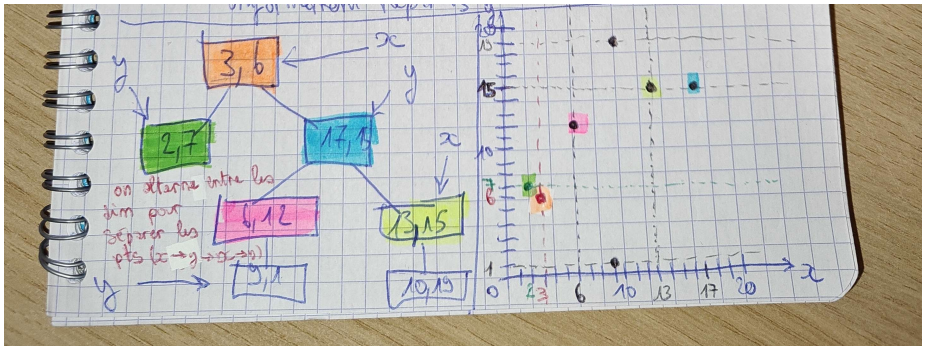

Figure 30: Exemple d'une structure de l'arbre KD ainsi que sa représentation géométrique (points et coupes qu'il effectue dans le plan)

6.2.2 Construction du kd-tree dans un espace 3D

Étapes principales :

1. **Boîte englobante AABB**
   La boîte englobante globale de la scène est obtenue en recherchant, pour chaque axe X, Y et Z, les valeurs minimales et maximales parmi l'ensemble des sommets de tous les triangles de la scène. Cette boîte, alignée sur les axes, définit le volume racine du KD-tree et sert à initialiser le test d'intersection rayon–boîte.

2. **Structure d'un Noeud**
   - *boundingBox* : AABB du noeud
   - *splittingAxis* : Axe de division (0=X,1=Y,2=Z)
   - *cuttingPos* : Position de coupe
   - *leftChild* et *rightChild* : Pointeurs vers les enfants
   - *primitives* : Liste de primitives si feuille
   - *leaf* : Booléen pour savoir si le nœud est terminal

3. **Structure du KD-Tree**
   - *root* : Racine de l'arbre
   - *maxDepth* : Limite de profondeur pour éviter récursions infinies
   - *maxPrimitivesPerLeaf* : Nombre maximum de primitives dans une feuille
   - *epsilon* : Taille minimale de l'AABB pour couper

4. **Cas d'arrêt (→ créer une feuille)**
   - Primitives $\leq$ *maxPrimitivesPerLeaf*
   - Profondeur $\geq$ *maxDepth*
   - Taille de l'AABB $<$ *epsilon*
   - Partition dégénérée :
     - Aucune séparation réelle des primitives
     - Une des listes des primitives est vide
     - Duplication totale des primitives

5. **Choix du plan de division**
   - L'axe de coupe est déterminé selon la plus grande dimension de l'AABB
   - La position de coupe est choisie en coupant au milieu de la boîte englobante sur l'axe déterminé

6. **Récursion**

   - Les objets sont divisés en deux sous-ensembles, assignés aux enfants gauche et droit
   - La construction se poursuit récursivement jusqu'à ce que les conditions d'arrêt soient remplies

6.2.3 Ajout d'un chronomètre

Avant d'implémenter le KD-tree, j'ai intégré un chronomètre permettant de mesurer le temps d'exécution du rendu, afin de pouvoir quantifier précisément le gain de performance apporté par la structure.

```
gMini: a minimal OpenGL/GLUT application
for 3D graphics.
Author : Tamy Boubekeur (http://www.labri.fr/~boubek)

Usage : ./gmini [<file.off>]
Keyboard commands
-----------------
 ?: Print help
 w: Toggle Wireframe Mode
 g: Toggle Gouraud Shading Mode
 f: Toggle full screen mode
 <drag>+<left button>: rotate model
 <drag>+<right button>: move model
 <drag>+<middle button>: zoom
 q, <esc>: Quit

Maillage chargé : img/star.off
Ray tracing a 480 x 480 image
Progress: 100% - Done!
Image saved to './rendu.ppm' in 100413 ms.
```

Figure 31: Temps d'exécution du rendu du maillage étoile

6.3 Comparaison des versions de KDTree

6.3.1 Contexte et objectif

Le KDTree a été introduit pour accélérer le raytracing en réduisant le nombre de tests d'intersection entre rayons et primitives. Trois versions de la fonction **buildNode** ont été développées et testées avec différentes configurations de paramètres (*maxDepth*, *maxPrimitivesPerLeaf*, *epsilon*) afin d'évaluer les performances.

6.3.2 V1 – Partition simple sur l'axe le plus long

Description :

- Coupe chaque nœud au milieu de l'axe le plus long
- Les primitives qui traversent la coupe sont dupliquées dans les deux enfants
- Gère les cas dégénérés (feuilles trop petites, partition impossible)

Observations :

- Très robuste, évite les partitions dégénérées
- Dupliquer les primitives dans les deux enfants entraîne un coût supplémentaire pour la récursion et le stockage
- Temps de rendu : 149 295 ms (plus lent que le rendu sans KDTree qui est à 100413 ms)

Analyse :

- L'overhead dû à la duplication et à la gestion des nœuds internes est plus important que le gain apporté par la réduction du nombre de tests
- Paramètres par défaut ($maxDepth = 15$, $maxPrimitivesPerLeaf = 30$) réduisent légèrement le temps mais restent sous-optimaux

6.3.3 V2 – Centre de gravité des primitives

Description :

- Coupe selon le centre moyen des primitives sur l'axe choisi (approche SAH-lite)
- Plus intelligente que V1, car la coupe suit la répartition réelle des primitives
- La duplication est supprimée : chaque primitive est placée dans un seul enfant selon le centre

Observations :

- Temps de rendu : 107 854 ms (beaucoup plus rapide que V1)
- Les nœuds sont mieux équilibrés, ce qui réduit la profondeur et les tests inutiles

Analyse :

- V2 améliore clairement les performances en évitant la duplication et en rapprochant les partitions de la distribution réelle des primitives
- Cependant, certaines partitions restent dégénérées si toutes les primitives ont des centres très proches, ce qui peut créer des feuilles très grandes

6.3.4 V3 – Centres basés sur les AABB précalculées

Description :

- Coupe selon le centre des AABB de chaque primitive, plutôt que le centre des points/centre de gravité
- L'AABB des meshes est précalculée et utilisée pour accélérer le calcul des min/max
- Partitionnement basé sur le centre de l'AABB pour chaque primitive, mais toujours pas de duplication

Observations :

- Temps de rendu : 105 792 ms (le plus rapide parmi les trois)
- La prise en compte des AABB réduit le calcul des min/max par primitive et permet des partitions plus cohérentes

Analyse :

- L'utilisation d'AABB pré-calculées réduit le coût de construction du KDTree et améliore l'équilibre des nœuds
- Les performances dépendent fortement du choix des paramètres (*maxDepth*, *maxPrimitivesPerLeaf*, *epsilon*)
- La fusion des petites feuilles avec *epsilon* = 0.01f évite des nœuds trop petits qui ne filtreraient pas efficacement les primitives

6.3.5 Tableau de comparaison des versions du Kd-Tree

Version	Coupe	Dup.	Centre	Temps (ms)	Observations
V1	Milieu axe long	Oui	Milieu AABB	149 295	Beaucoup de duplication, trop de nœuds, plus lent que sans KDTree.
V2	Centre gravité	Non	Gravité	107 854	Partitions mieux équilibrées, rendu plus rapide, quelques feuilles dégénérées.
V3	Centre AABB	Non	Centre AABB	105 792	Meilleur compromis : équilibrage efficace et calcul AABB plus rapide.

Table 1: Comparaison compacte des versions de KDTree

6.3.6 Choix de la meilleure version du Kd-Tree

V3 est retenue comme version finale pour le rendu :

- Partition plus cohérente grâce aux AABB
- Pas de duplication → moins de nœuds et moins d'intersections inutiles
- Temps de rendu le plus faible

Paramètres retenus pour cette version :

- $maxDepth = 15$
- $maxPrimitivesPerLeaf = 30$
- $epsilon = 0.01\text{f}$

6.3.7 Améliorations possibles pour le futur

1. SAH complet (Surface Area Heuristic) pour calculer la position de coupe qui minimise le coût attendu des intersections
2. Bounding Volume Hierarchy (BVH) par feuille pour réduire encore le nombre de tests dans les grandes feuilles
3. Fusion des nœuds dégénérés et équilibrage plus intelligent des enfants
4. Optimisation mémoire : éviter la duplication des primitives et utiliser des indices au lieu de copies complètes

7 Rendus inattendus notables des dernières phases

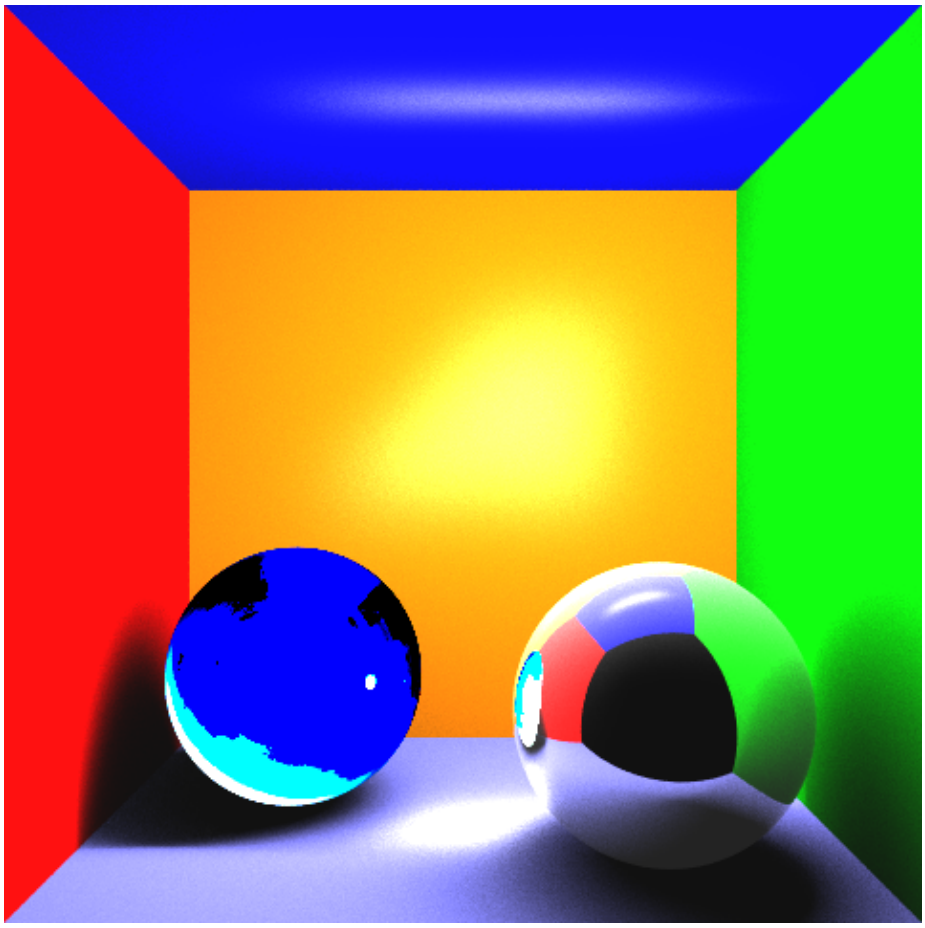

Figure 32: Erreur de rendu de la sphère transparente