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.

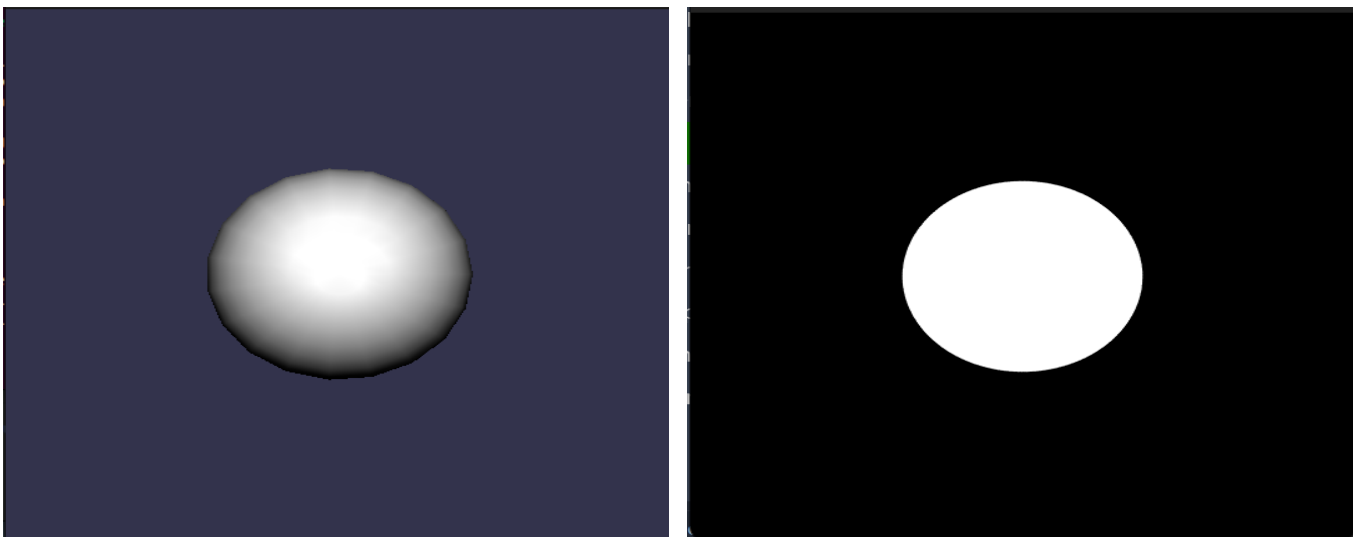

Figure 1: Sphère seule

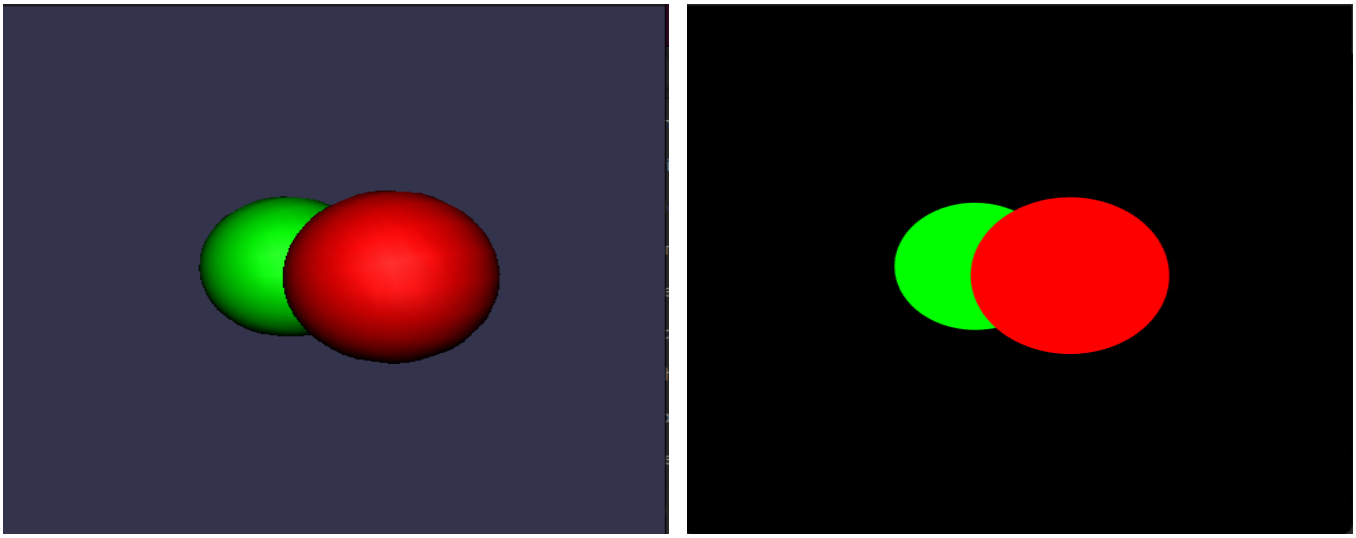

Figure 2: Deux sphères

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):

$$p = o - a \tag{2}$$

$$t = \frac{p.n}{d.n} \tag{3}$$

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 les 4 sommets du carré. Si le produit scalaire entre la normale au carré et chacun de ces vecteurs est supérieur à 0 alors le point est dans le carré.

```
bool isInSquare(Vec3 pos, Vec3 normal) const
{

    Vec3 bottom = Vec3::cross(bottomRight() - bottomLeft(), pos - bottomLeft());
    Vec3 right = Vec3::cross(upRight() - bottomRight(), pos - bottomRight());
    Vec3 up = Vec3::cross(upLeft() - upRight(), pos - upRight());
    Vec3 left = Vec3::cross(bottomLeft() - upLeft(), pos - upLeft());

    bool direction = (Vec3::dot(bottom, normal) > 0);

    bool sameDirection = (Vec3::dot(right, normal) > 0) == direction &&
                         (Vec3::dot(left, normal)  > 0) == direction &&
                         (Vec3::dot(up, normal)    > 0) == direction;
```

```
    return sameDirection;
}
```

```
RaySquareIntersection intersect(const Ray &ray) const {
        RaySquareIntersection intersection;
        Vec3 p = bottomLeft() - ray.origin();
        float t = ( Vec3::dot(p, normal()))/Vec3::dot(ray.direction(), normal());
        Vec3 point = ray.direction() * t + ray.origin();
        if(t>0 && isInSquare(point, normal()))
        {
            intersection.intersectionExists = true;
            intersection.t = t;
            intersection.intersection = point;
            intersection.normal = normal();
            intersection.normal.normalize();
            return intersection;
        }
        else
        {
            intersection.intersectionExists = false;
            return intersection;
        }

    }
```

On a donc le résultat suivant.

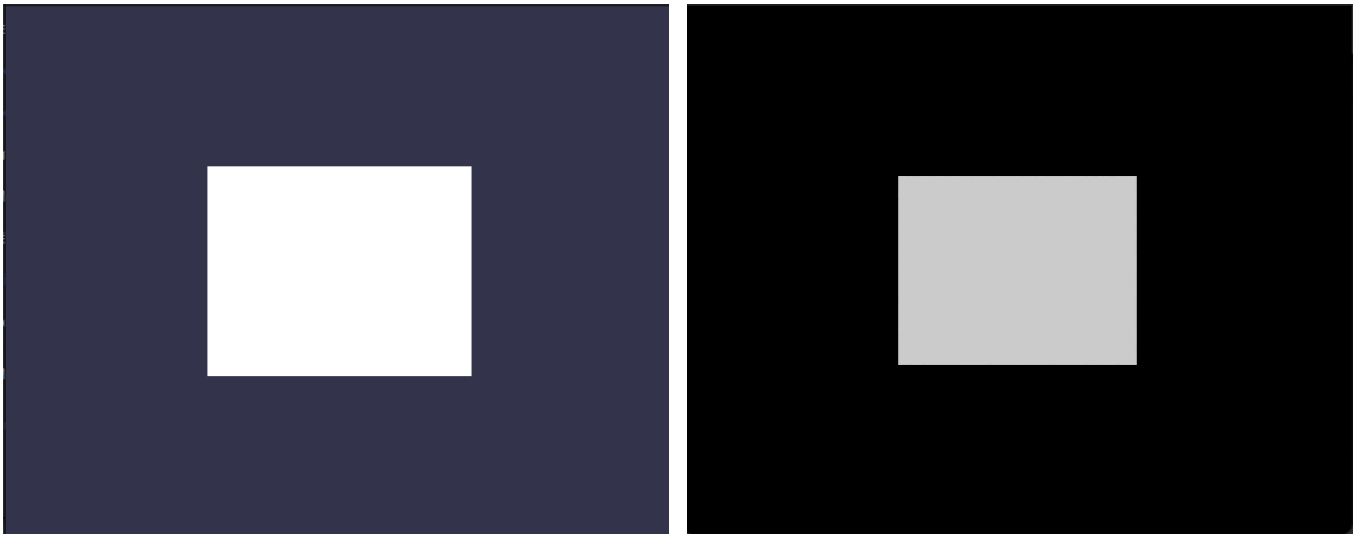

Figure 3: Carré

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.

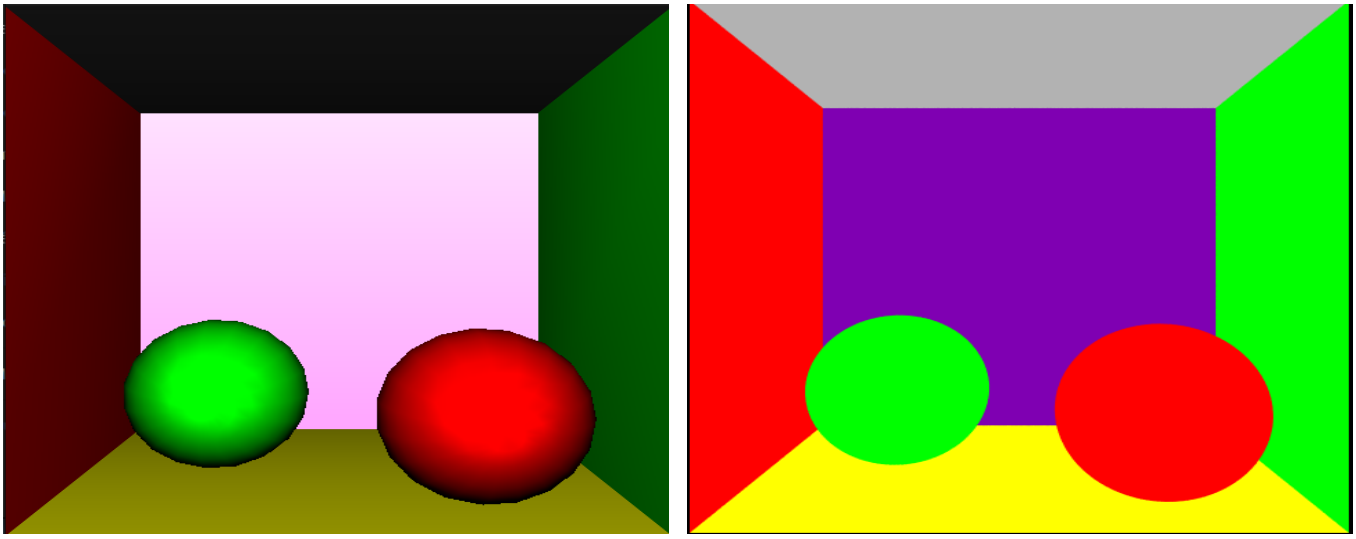

Figure 4: Boîte de Cornell

3 Phase 2

3.1 Illumination de Phong

Le calcul de l'illumination de Phong est réalisé dans une fonction dont les paramètres sont la lumière, le rayon, la normale à l'intersection, le point d'intersection, et les 4 attributs du matériau. On reprends les formules du cours pour les calculs.

```
Vec3 computePhongIllumination(
        Ray ray,
        Light light,
        Vec3 normal,
        Vec3 intersection,
        Vec3 ambient_material,
        Vec3 diffuse_material,
        Vec3 specular_material,
        double shininess)
{
    Vec3 L = light.pos - intersection;
    L.normalize();
    Vec3 reflexion = 2 * (Vec3::dot(L, normal)) * normal - L;
    reflexion.normalize();
    Vec3 rayDir = ray.origin() - intersection;
    rayDir.normalize();
    float alpha = std::max(Vec3::dot(reflexion, rayDir), 0.f);
    float theta = std::max(Vec3::dot(L,normal), 0.f);
    Vec3 ambient = Vec3::compProduct(light.material, ambient_material);
    Vec3 specular = Vec3::compProduct(
                    light.material,
                    specular_material) *
                    pow(alpha, shininess);
    Vec3 diffuse=Vec3::compProduct(
                    light.material,
                    diffuse_material)*theta;

    return ambient + specular + diffuse;
}
```

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)
                color = computePhongIllumination(/*...*/);
        else if(raySceneIntersection.typeOfIntersectedObject == ObjectType_Square)
              color = computePhongIllumination(/*...*/);
    }
    return color;
}
```

Pour avoir un meilleur comparatif des résultats et des attendus, nous utiliserons à présent la coloration "originale" de la boîte de cornell.

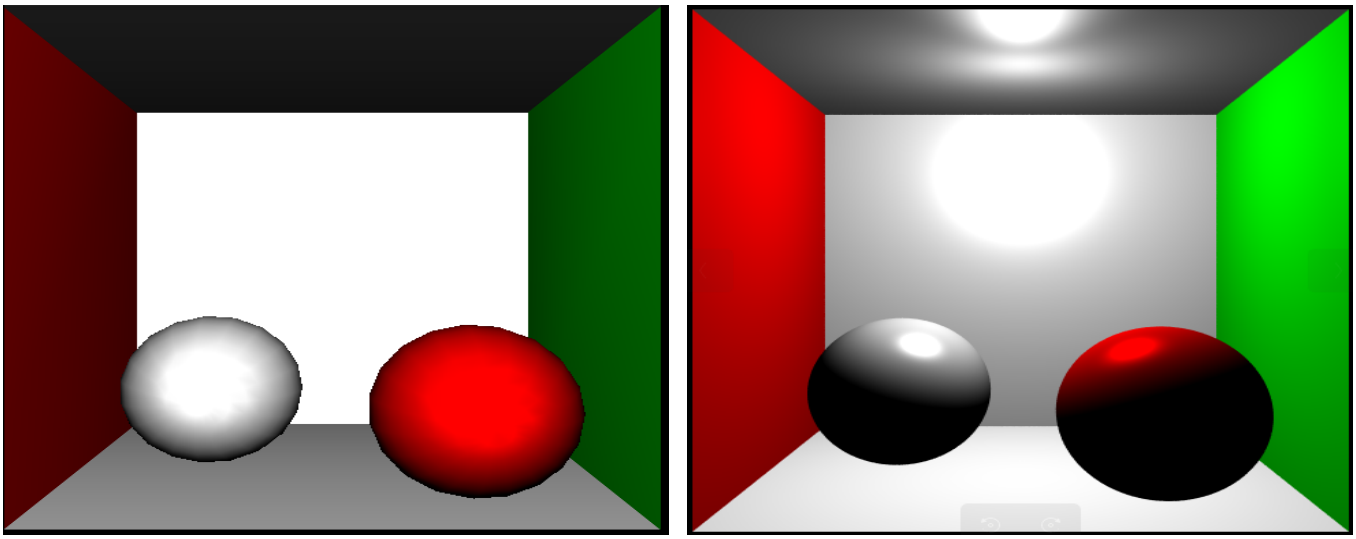

Figure 5: Illumination de Phong

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;
}

Vec3 computeHardShadow(Ray ray, Light light, Vec3 normal,
        Vec3 intersection,
        Vec3 ambient_material,
        Vec3 diffuse_material,
        Vec3 specular_material,
        double shininess)
{
    Vec3 L = light.pos - intersection;
    float shadowMinDist = L.length();
    L.normalize();
    Ray shadowRay = Ray(intersection, L);
    RaySceneIntersection shadowIntersection = computeShadowIntersection(shadowRay,
   shadowMinDist);
    if(shadowIntersection.intersectionExists)
        return Vec3(0,0,0);
    else
        return computePhongIllumination(ray, light, normal, intersection,
   ambient_material, diffuse_material, specular_material, shininess);

}
```

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)
          {
            color += computeHardShadow(/*...*/);
          }
        }
        else if(raySceneIntersection.typeOfIntersectedObject == ObjectType_Square)
        {

          for (int i = 0; i < lights.size(); ++i)
          {
            color += computeHardShadow(/*...*/);
          }
        }

    }
```

```
    return color;
}
```

On obtient finalement le résultat suivant

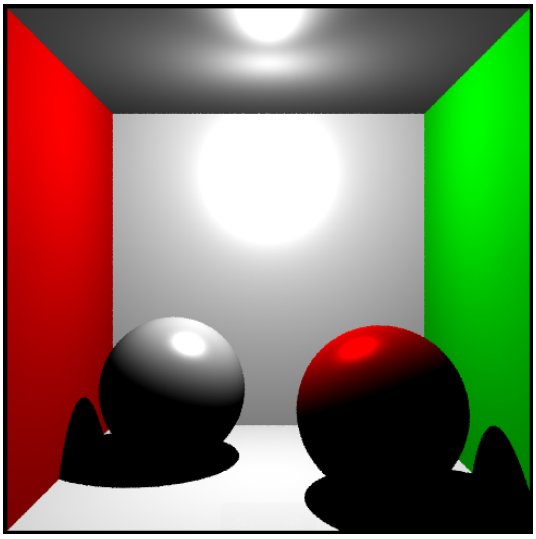

Figure 6: Calcul des ombres

3.3 Ombres Douces

Le calcul des ombres douces nécéssite l'échantillonnage de la source de lumière (simulation d'une source étendue). Pour chaque échantillon, un rayon depuis le point d'intersection sera tiré. On pondère finalement la couleur du pixel au point d'intersection initial par le pourcentage de rayons ayant atteint la source lumineuse depuis ce point. La fonction computeSoftShadow retourne donc ce pourcentage.

```
float computeSoftShadow(Light light, Vec3 intersection)
{

    float counter = 0;
    for(float k = 0; k<LIGHT_SAMPLES; k++)
    {
        float x =(float)(rand()/(float)(RAND_MAX / (light.radius)));
        float y =(float)(rand()/(float)(RAND_MAX / (light.radius)));
        float z =(float)(rand()/(float)(RAND_MAX / (light.radius)));
        Vec3 pos = Vec3(light.pos[0] + x, light.pos[1] + y, light.pos[2] + z);

        Vec3 L = pos - intersection;
        L.normalize();
        Ray shadowRay = Ray(intersection, L);
        RaySceneIntersection shadowIntersection = computeShadowIntersection(shadowRay,
    0.9f);
        if(shadowIntersection.intersectionExists) counter++;

    }
    float shadow = counter/LIGHT_SAMPLES;
    return (1.0f - shadow);
}
```

On modifie ensuite la fonction rayTraceRecursive pour la pondération de la couleur.

```

Vec3 rayTraceRecursive( Ray ray , int NRemainingBounces, float minDist)
```

```
{
    RaySceneIntersection raySceneIntersection = computeIntersection(ray, minDist);
    Vec3 color;
    if(!raySceneIntersection.intersectionExists) return color;
    if(raySceneIntersection.typeOfIntersectedObject == ObjectType_Sphere)
    {
        Vec3 normal = raySceneIntersection.raySphereIntersection.normal;
        Vec3 point = raySceneIntersection.raySphereIntersection.intersection;
        float intersected = 0;
        for (int i = 0; i < lights.size(); ++i)
        {
            color = computePhongIllumination(/*...*/);
            float shadow = computeSoftShadow(lights[i], point);
            color *= shadow;
        }
    }

    /* Repete pour chacun des types d'objets */
}
```

On obtient donc les résultats suivants.

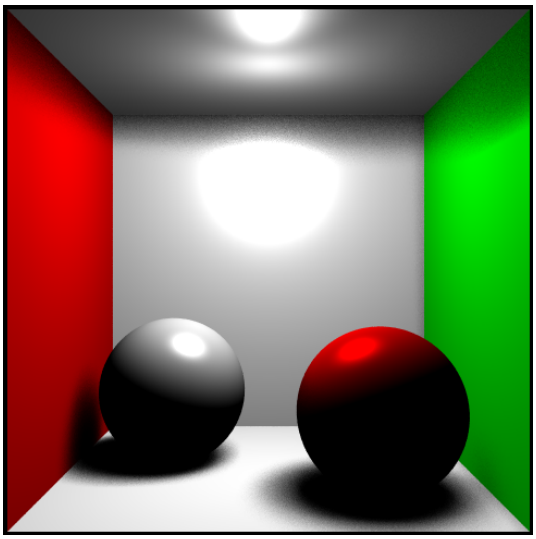

Figure 7: Ombres douces

4 Phase 3

4.1 Intersection Rayon-Triangle

Pour pouvoir ajouter un maillage à la scène, il faut d'abord implémenter le calcul de l'intersection entre un rayon et un triangle (une face du maillage). Ce calcul se décompose en plusieurs étapes :

- Calcul d'intersection rayon plan
- Calcul de l'appartenance au triangle

4.1.1 Intersection Rayon-Plan

Nous avons déjà calculé l'intersection rayon plan lors du calcul de l'intersection rayon-carré. En revanche, comme nous cherchons à calculer l'intersection avec un maillage, il se peut que certaines faces

soient parallèles au rayon. La classe Triangle implémente donc la fonction IsParallelTo qui effectue cette vérification.

```
bool isParallelTo( Line const & L ) const {
    bool result;
    float dirDotNorm = Vec3::dot(L.direction(), normal());
    result = (dirDotNorm == 0);
    return result;
}
Vec3 getIntersectionPointWithSupportPlane( Line const & L, float &_t) const {
    Vec3 result;
    if(! isParallelTo(L))
    {
        _t = ( Vec3::dot((c0() - L.origin()), normal()))/Vec3::dot(L.direction(), normal
    ());
        if(_t>0) result = L.direction() * _t + L.origin();
    }
    return result;
}
```

4.1.2 Appartenance au Triangle

Une fois que l'on sait que le rayon intersecte le plan porteur du triangle, il faut vérifier que le point d'intersection se situe dans le triangle. Pour cela on calcule les coordonées barycentriques du point d'intersection, si ces coordonées ont une valeur comprise entre 0 et 1 inclus et que leur somme vaut 1, alors le point est dans le triangle. *Coordonées barycentriques*
Le calcul des coordonnées barycentriques s'effectue de la façon suivante :

- Calcul des vecteurs directeurs des arêtes du triangle
- Calcul des vecteurs directeurs du point d'intersection à chacun des sommets
- Calcul des aires

```
void computeBarycentricCoordinates( Vec3 const & p , float & u0 , float & u1 , float &
    u2 , bool & isInside) const {
    Vec3 c0c1 = (c1() - c0());
    Vec3 c1c2 = (c2() - c1());
    Vec3 c2c0 = (c0() - c2());

    Vec3 c0pc1normal = Vec3::cross(c0c1, (p-c0()));
    Vec3 c1pc2normal = Vec3::cross(c1c2, (p-c1()));
    Vec3 c2pc0normal = Vec3::cross(c2c0, (p-c2()));

    isInside = (Vec3::dot(c1pc2normal, normal()) > 0) && (Vec3::dot(c0pc1normal, normal
    ()) > 0) && (Vec3::dot(c2pc0normal, normal()) > 0);

    float area0 = c1pc2normal.length()/2.f;
    float area1 = c2pc0normal.length()/2.f;
    float area2 = c0pc1normal.length()/2.f;

    u0 = area0/this->area;
    u1 = area1/this->area;
    u2 = area2/this->area;
}
```

Finalement on a pour l'intersection rayon-triangle

```
RayTriangleIntersection getIntersection( Ray const & ray ) const {
RayTriangleIntersection result;
```

```
Line rayLine = (Line) ray;
if(isParallelTo(rayLine))
{
    result.intersectionExists = false;
    return result;
}
float t;
Vec3 planeIntersection = getIntersectionPointWithSupportPlane(rayLine, t);
if(t<0)
{
    result.intersectionExists = false;
    return result;
}
float w0,w1,w2;
bool isInside;
computeBarycentricCoordinates(planeIntersection, w0,w1,w2, isInside);
if(( 0.f<=w0 && w0<=1.f) &&
   ( 0.f<=w1 && w1<=1.f)  &&
   ( 0.f<=w2 && w2<=1.f) && isInside)
{
    result.intersectionExists = true;
    result.t = t;
    result.w0 = w0;
    result.w1 = w1;
    result.w2 = w2;
    result.intersection = Vec3(planeIntersection[0] * w0, planeIntersection[1] * w1,
    planeIntersection[2] * w2);
    Vec3 norm = Vec3(normal()[0], normal()[1], normal()[2]);
    norm.normalize();
    result.normal = norm;
}
else
{
    result.intersectionExists = false;
    return result;
}
return result;
}
```

On obtient donc les résultats suivants

4.2 Intersection Rayon-Maillage

Pour calculer l'intersection entre le rayon et le maillage, on crée, pour chacune des faces du maillage, un triangle avec lequel on calculera l'intersection.

```
RayTriangleIntersection intersect( Ray const & ray ) const {
    RayTriangleIntersection closestIntersection;
    float triangleScaling = 0.0001f;
    closestIntersection.t = FLT_MAX;
    for (int i = 0; i < triangles.size(); ++i)
    {
        MeshTriangle meshTriangle = triangles[i];
        Triangle triangle = Triangle(vertices[triangles[i][0]].position *
    triangleScaling,
                                     vertices[triangles[i][1]].position *
    triangleScaling,
                                     vertices[triangles[i][2]].position *
    triangleScaling);
        RayTriangleIntersection intersection = triangle.getIntersection(ray);
        if(intersection.intersectionExists && intersection.t<closestIntersection.t &&
    intersection.t> 0.001f) closestIntersection = intersection;
```

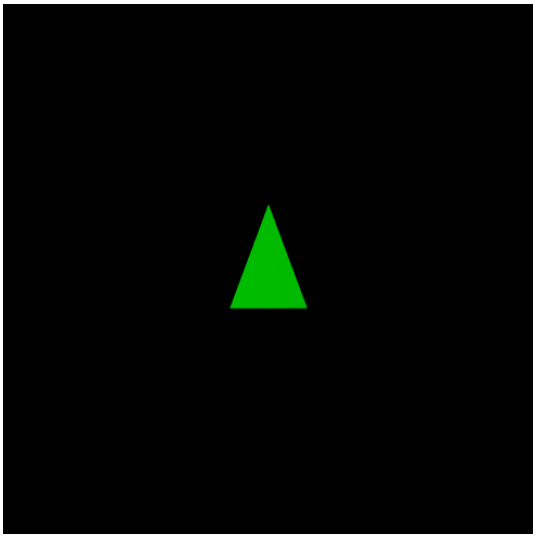

Figure 8: Triangle

```
    }
    return closestIntersection;
}
```

On charge alors un modèle suzanne dans la scène décrite par la méthode suivante.

```
void setup_single_mesh(){
    meshes.clear();
    spheres.clear();
    squares.clear();
    lights.clear();
     {
        lights.resize( lights.size() + 1 );
        Light & light = lights[lights.size() - 1];
        light.pos = Vec3(-5,5,5);
        light.radius = 2.5f;
        light.powerCorrection = 2.f;
        light.type = LightType_Spherical;
        light.material = Vec3(1,1,1);
        light.isInCamSpace = false;
    }
    {
        meshes.resize( meshes.size() + 1);
        Mesh & mesh = meshes[meshes.size() - 1];
        mesh.loadOFF("./data/suzanne.off");
        mesh.centerAndScaleToUnit();
        mesh.build_arrays();
        mesh.material.type = Material_Diffuse_Blinn_Phong;
        mesh.material.diffuse_material = Vec3( 0.,1.,0.);
        mesh.material.specular_material = Vec3( 0.2,0.2,0.2 );
        mesh.material.shininess = 20;
    }
}
```

On obtient alors les résultats suivants.

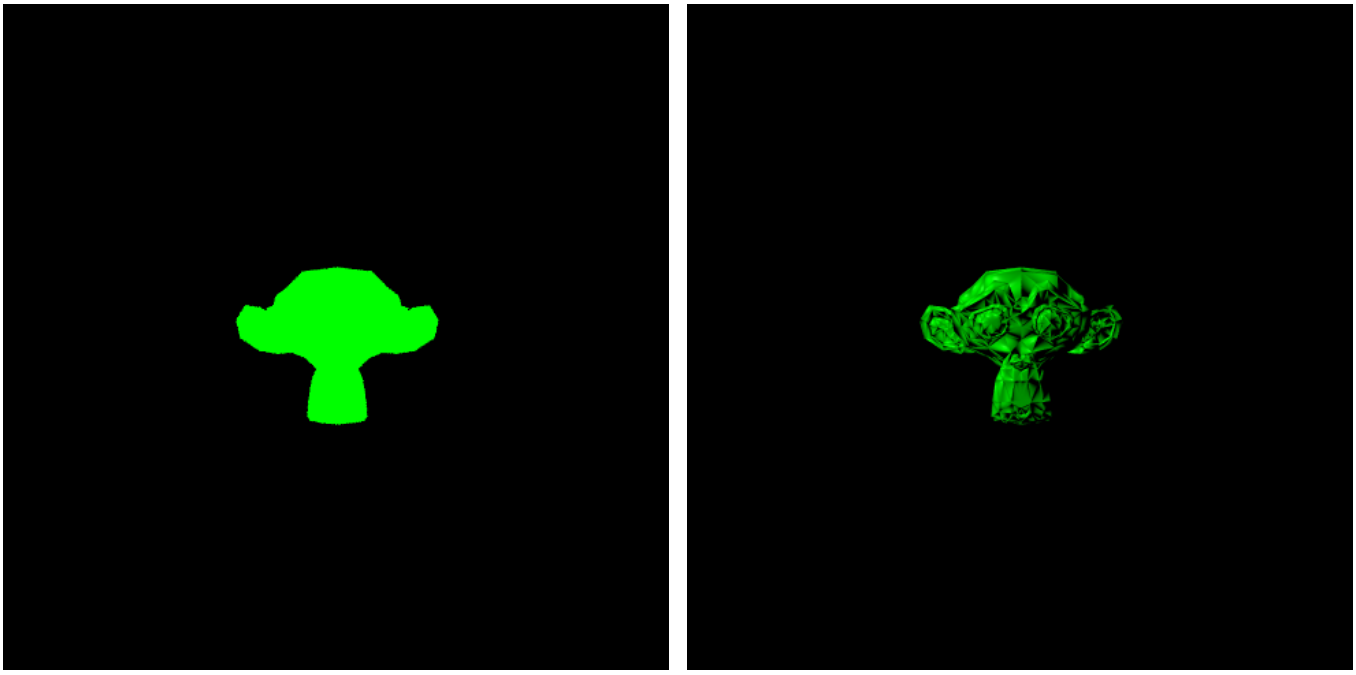

Figure 9: Suzanne

5 Phase finale

5.1 Sphère réfléchissante

Pour pouvoir rendre une sphère rélféchissante, il faut à partir du point d'intersection avec la sphère, calculer le vecteur directeur du rayon réfléchi et tirer ce rayon. Si on note R le vecteur directeur du rayon réfléchi, on a :

$$R = d - (2 * n * d.n); \tag{4}$$

tel que d est la direction du premier rayon (partant de la caméra) et n la normale à la surface. On ajoute donc la récursivité à la fonction rayTraceRecursive et on a :

```
 Vec3 rayTraceRecursive( Ray ray , int NRemainingBounces, float minDist)
{
    if(NRemainingBounces == 0) return Vec3();
    /*...*/
    if(spheres[raySceneIntersection.objectIndex].material.type == Material_Mirror)
    {
        Vec3 reflectionDir = ray.direction() - (2 * normal * Vec3::dot(ray.direction(),
   normal));
        reflectionDir.normalize();
        Ray reflected = Ray(point, reflectionDir);
        color += rayTraceRecursive(reflected, NRemainingBounces - 1, 0.0001f);
    }
    /*...*/
}
```

On obtient alors le résultat suivant :

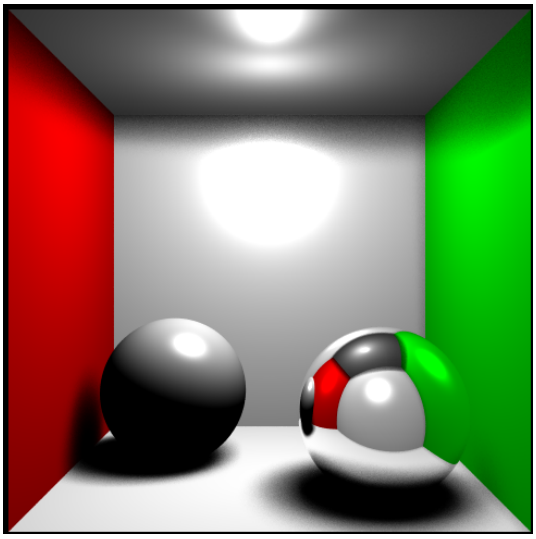

Figure 10: Sphère réflechissante

6 Refraction

Le calcul de la réfraction sur une sphère se décompose en deux parties. En effet, on peut distinguer deux "types" de réfractions. La première que l'on dira "classique" utilise la loi de Snell-Descartes pour calculer la direction du rayon réfracté. On détaillera ensuite la formule pour l'autre type de réfraction donnant un effet "lentille".

6.1 Réfraction 1

Ce calcul est réalisé dans la fonction suivante :

```
Vec3 getRefractedRayDir(Vec3 point, Vec3 normal, Vec3 rayDir, double index_medium)
{
    double cosIncident = Vec3::dot(rayDir, normal);
    double n1, n2;
    if(index_medium == 1.0) return rayDir;
    if(cosIncident > 0.0)
    {
        n1 = index_medium;
        n2 = 1.0;
        cosIncident = -cosIncident;
        normal = -1 * normal;
    }
    else{
        n1 = 1.0;
        n2 = index_medium;
    }
    double n = n1/n2;
    double sinTheta = n *(1 - cosIncident * cosIncident);
    Vec3 refractDir;
    if(sinTheta<1.0)
    {
        double cosTheta = sqrt(1.0 - sinTheta * sinTheta);
        refractDir = n * rayDir + (n * cosIncident - cosTheta) * normal;
        refractDir.normalize();
    }
    return refractDir;
}
```

Enfin, on ajoute à la fonction récursive de lancer de rayons les lignes suivantes :

```
//...
else if (spheres[raySceneIntersection.objectIndex].material.type == Material_Glass)
{
    Vec3 refractDir = getRefractedRayDir(point, normal, ray.direction(), spheres[
    raySceneIntersection.objectIndex].material.index_medium);
    if(refractDir.length()>0)
    {
        Ray refracted = Ray(point, refractDir);
        color += rayTraceRecursive(refracted, NRemainingBounces - 1, 0.0001f);
    }
}
//...
```

On obtient alors les résultats suivants.

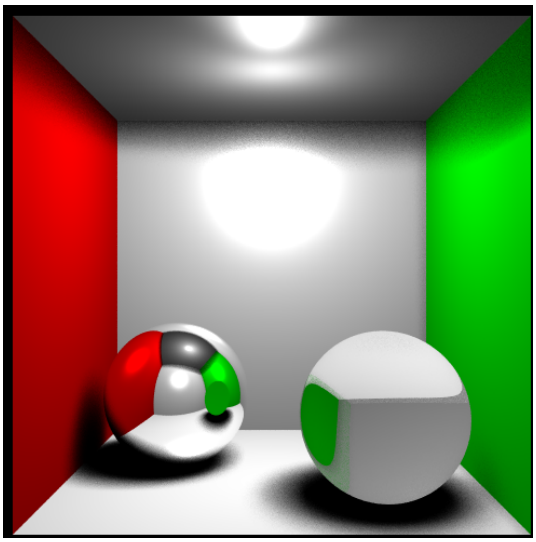

Figure 11: Sphère Verre (index medium = 1.4)

6.2 Réfraction 2

Le vecteur directeur du rayon réfracté donnant un effet "lentille" est calculé dans la fonction suivante :

```
Vec3 getTransparentDir(Vec3 point, Vec3 normal, Vec3 raydir, float index_medium)
{
    float dot = Vec3::dot(raydir, normal);
    float prod = (index_medium * index_medium) * (dot * dot);

    // create new ray
    Vec3 ray_inv_d;
    ray_inv_d = (index_medium * raydir) + (index_medium * (dot + sqrt(prod)) * normal);
    ray_inv_d.normalize();

    return ray_inv_d;
}
```

Enfin, on ajoute à la fonction récursive de lancer de rayons les lignes suivantes :

```
//...
else if (spheres[raySceneIntersection.objectIndex].material.type == Material_Transparent
    )
{
```

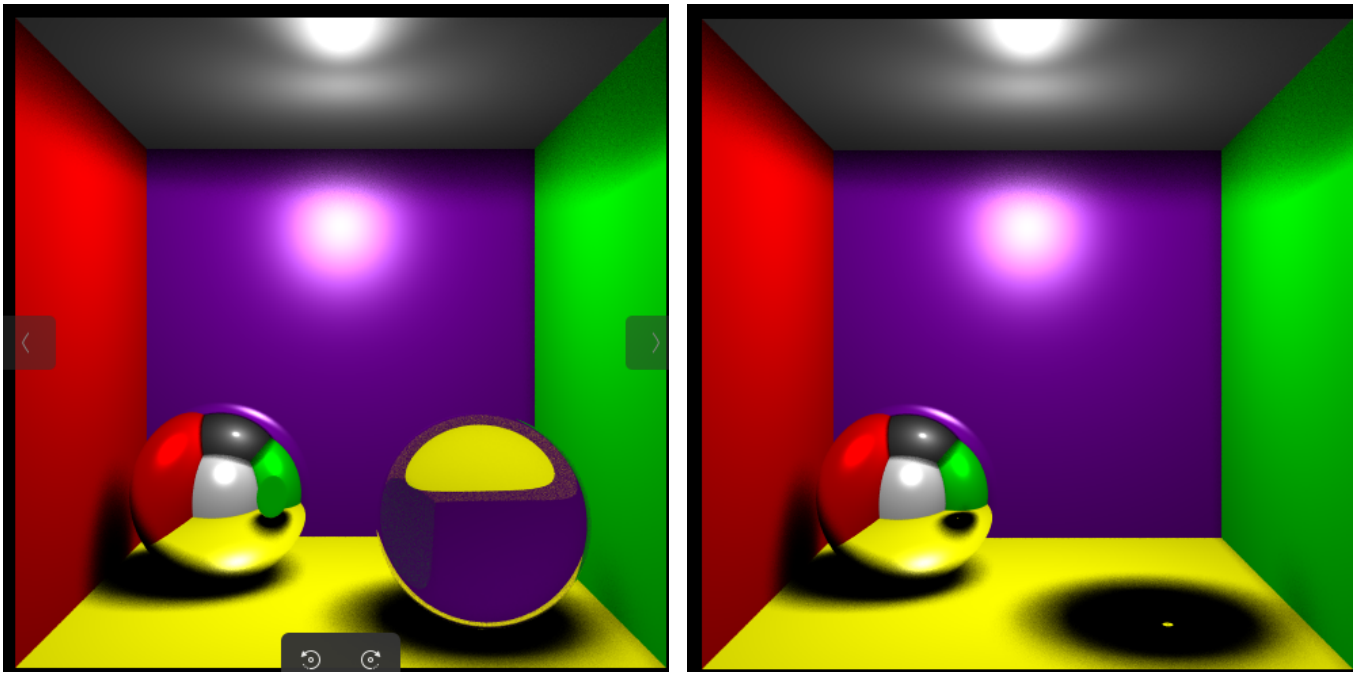

Figure 12: Sphère Verre indexes medium 1.4 (gauche) 1 (droite)

```
    Vec3 transDir = getTransparentDir(point, normal, ray.direction(), spheres[
    raySceneIntersection.objectIndex].material.index_medium);
    if(transDir.length()>0)
    {
        Ray transparentRay = Ray(point, transDir);
        color += rayTraceRecursive(transparentRay, NRemainingBounces - 1, 0.0001f);
    }
}
//...
```

On obtient alors les résultats suivants.

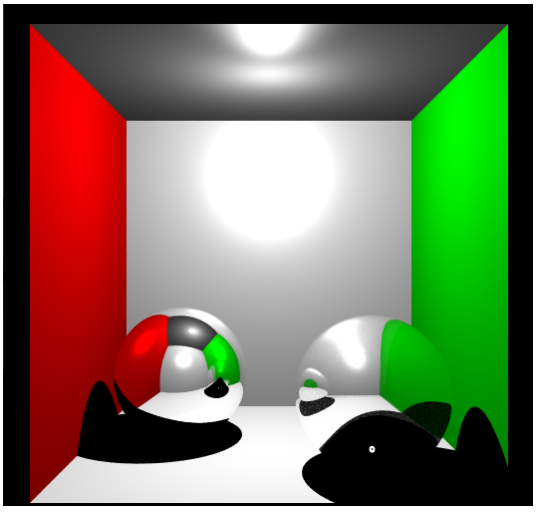

Figure 13: Sphère Verre effet "lentille"

7 Structure d'accélération

7.1 Boîte englobante

Pour tenter d'accélérer notre processus de lancer de rayon (surtout dans le cas du maillage), on peut dans un premier temps utiliser la boîte englobante. On calcule la boite englobante du maillage, et on vérifie si le rayon intersecte cette boîte plutôt que de calculer les intersections (ou non) avec chacun des triangles du maillage. Le calcul des bornes minimum et maximum de la boîte englobante d'un maillage est donné par la fonction suivante :

```
std::vector<Vec3> boardingBox() const
{
    Vec3 bbmin, bbmax;
    bbmin = Vec3(positions_array[0], positions_array[1], positions_array[2]);
    bbmax = Vec3(positions_array[0], positions_array[1], positions_array[2]);

    for(size_t i = 3; i<positions_array.size(); i+=3)
    {
        for (int j = 0; j < 3; ++j)
        {
            if(bbmin[j] > positions_array[i+j]) bbmin[j] = positions_array[i+j];
            if(bbmax[j] < positions_array[i+j]) bbmax[j] = positions_array[i+j];
        }
    }
    bbmin -= Vec3(0.01, 0.01, 0.01);
    bbmax += Vec3(0.01, 0.01, 0.01);

    std::vector< Vec3 > boardingbox;

    boardingbox.push_back(bbmin);
    boardingbox.push_back(bbmax);

    return boardingbox;
}
```

Nous cherchons alors à savoir si un rayon intersecte cette boîte, et pour cela, nous disposons de la fonction suivante :

```
bool isInBoardingBox(Vec3 origin, Vec3 dir) const
{
    float tmin, tmax, tYmin, tYmax, tZmin, tZmax;
    tmin = (boardingbox[0][0] - origin[0])/dir[0]; // tXmin;
    tmax = (boardingbox[boardingbox.size()-1][0] - origin[0])/dir[0]; // tXmax;
    tYmin = (boardingbox[0][1] - origin[1])/dir[1];
    tYmax = (boardingbox[boardingbox.size()-1][1] - origin[1])/dir[1];
    tZmin = (boardingbox[0][2] - origin[2])/dir[2];
    tZmax = (boardingbox[boardingbox.size()-1][2] - origin[2])/dir[2];

    if(tmin > tmax) std::swap(tmin, tmax); // swap value if tmin is not min;
    if(tYmin > tYmax) std::swap(tYmin, tYmax); //same
    if(tZmin > tZmax) std::swap(tZmin, tZmax); //same
    if(tmin > tYmax || tmax < tYmin) return false;
    if(tmin < tYmin) tmin = tYmin;
    if(tmax > tYmax) tmax = tYmax;
    if(tmin > tZmax || tmax < tZmin) return false;
    return true;
}
```

Finalement, lorsqu'on compare les temps de rendus, on passe d'environ 600 secondes à 194 secondes pour le rendu de la boîte de cornell comprenant le maillage de suzanne.

7.2 Kd-Tree

L'arbre Kd est une structure de donnée nous permettant de partitionner notre maillage et de traiter beaucoup plus rapidement l'intersection d'un rayon avec ce dernier. Chaque noeud de l'arbre est défini par :

- sa boîte englobante
- son fils gauche et son fils droit
- une liste d'index des triangles (dans le maillage) contenus dans la boîte englobante
- l'axe sur lequel la partition de l'espace est réalisée

L'arbre est donc défini inductivement, et chaque niveau de profondeur partitionne l'espace sur un axe (cycle axe des x, axe des y puis axe des z). Enfin, un algorithme de parcours de l'arbre permet de calculer l'intersection d'un rayon avec l'arbre. L'algorithme retourne une liste indexée de triangles, utilisée dans le calcul d'intersection entre le maillage et le rayon, et s'écrit ainsi :

```
/*
Iterates over tree and add index of intersected triangle in the given set
*/
std::vector<int> intersectRecursive(Ray const& ray, Node* node)
{
    if (node->childLeft && KdTreeUtils::intersectNode(ray, node->childLeft))
    {

        if (node->childRight && KdTreeUtils::intersectNode(ray, node->childRight))
        {

            std::vector<int> result = intersectRecursive(ray, node->childLeft);
            std::vector<int> second = intersectRecursive(ray, node->childRight);

            result.insert(result.end(), second.begin(), second.end());
            return result;
        }
        else
            return intersectRecursive(ray, node->childLeft);
    }
    if (node->childRight && KdTreeUtils::intersectNode(ray, node->childRight))
    {
        return intersectRecursive(ray, node->childRight);
    }
    totale des triangles
    return node->triangles;

}
```

Malheureusement, certainement par manque de temps, quelques bugs n'ont pas été résolus. En effet, les intersections sont bien calculés dans le cas d'un tétrahèdre et d'un cube, mais quand il s'agit de suzanne, le rendu présente des erreurs.

8 Ajout de Texture

Lors de nos calculs d'intersection avec les sphères ou les quadrilatères, nous pouvons ajouter le calcul des coordonées de texture (au point d'intersection) de façon à pouvoir plaquer une texture sur nos objets (visibles uniquement lors du rendu).

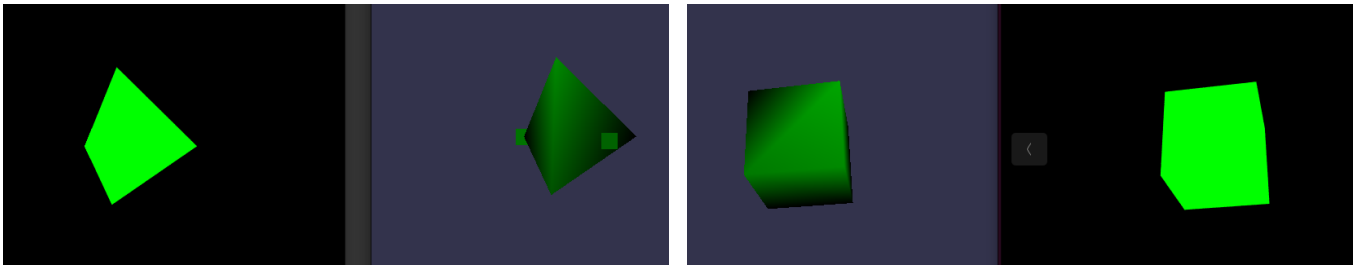

Figure 14: Rendu du tétrahèdre et du cube avec l'utilisation de l'arbre Kd

Figure 15: Rendu erroné du modèle de suzanne avec l'abre Kd

8.1 Chargement d'une texture

Le chargement d'une texture est implémenté dans les fonction de imageLoader, nous avons cependant ajouté un type de Matériau (Material Texture) et quelques méthodes au fichier Material.h, nous permettant de charger une texture.

```
void loadTexture2DFromFilePath(const std::string& path) {
   ppmLoader::ImageRGB image;
   ppmLoader::load_ppm(image, path);
   texture = image;
}

Vec3 sampleTexture(const float u, const float v) {
    Vec3 color;
    int i = floor(u*this->texture.w);
    int j = floor((1-v)*this->texture.h);
    int index = (i+j*this->texture.w);
    ppmLoader::RGB rgb = this->texture.data[index];
    color = Vec3((float)rgb.r/255.0, (float)rgb.g/255.0, (float)rgb.b/255.0);
    return color;
}
```

Une fois la texture chargée, il faut calculer les coordonnées de textures.

8.2 Coordonnées de textures (Intersection Rayon-Carré

Pour calculer les coordonnées de textures dans un carré, nous définissons deux axes et projetons le point d'intersection sur ces axes. Les axes sont dirigés par les vecteurs définis par les coins inférieur-gauche et

supérieur-gauche (axe vertical) et inférieur-gauche et inférieur-droit (axe horizontal). Les coordonnées de textures sont donc données par les formules suivantes, en notant x le vecteur directeur de l'axe horizontal et y le vecteur directeur de l'axe vertical et p le vecteur d'un point du carré au point d'intersection.

$$\begin{aligned} u &= \frac{||\frac{p.x}{||x||^2}x||}{||x||} \\ v &= \frac{||\frac{p.y}{||y||^2}y||}{||y||} \end{aligned} \tag{5}$$

8.3 Coordonnées de textures (Intersection Rayon-Sphère

Pour calculer les coordonnées de textures dans une sphère, on dispose déjà d'une fonction permettant de passer des coordonnées euclidiennes au coordonnées sphériques. De ces coordonnées, que l'on notera ϑet φ, on donne les coordonnées de textures u et v par la formule suivante :

$$\begin{aligned} u &= 0.5 + \frac{\theta}{2\pi} \\ v &= 0.5 - \frac{\phi}{\pi} \end{aligned} \tag{6}$$

8.4 Résultats

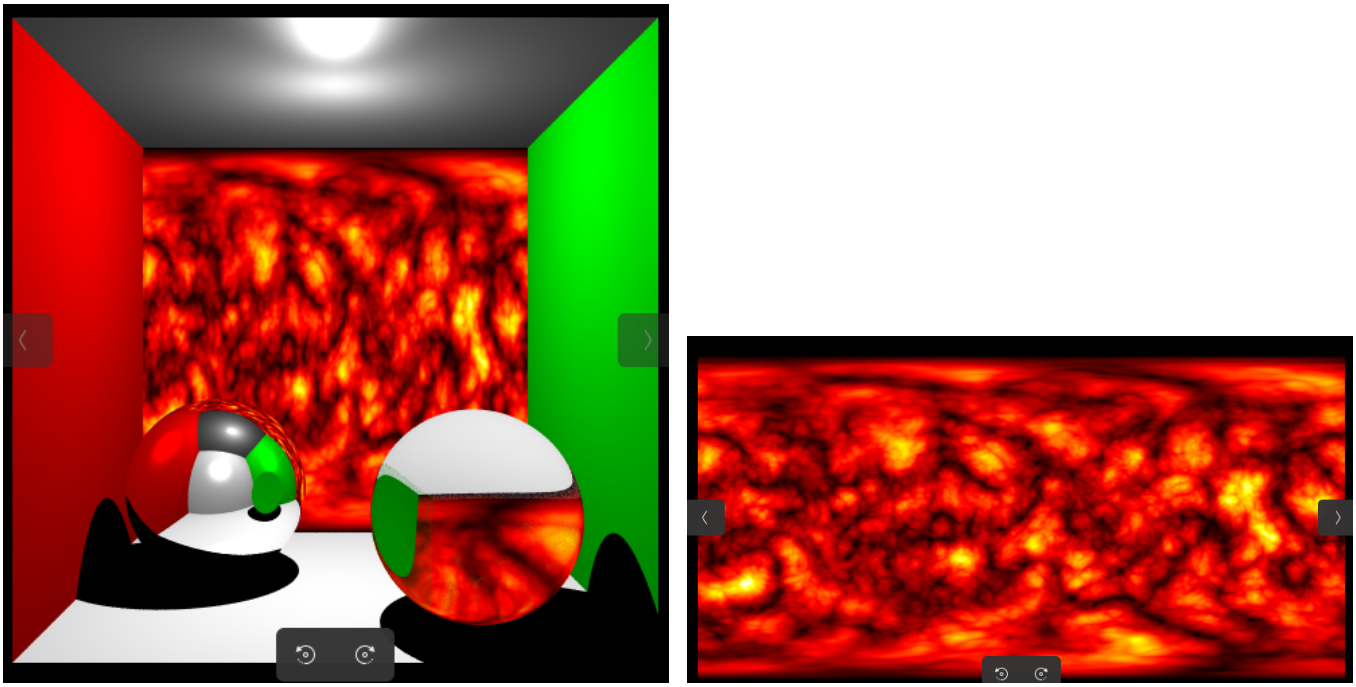

Figure 16: Texture plaquée sur le mur du fond de la boîte de la boite de cornell

9 Conclusion

Malgré certains bugs et erreurs encore non résolues, ce projet nous a appris de nombreuses notions d'informatique graphique et de rendu. Le temps investi dans ce projet nous motive à continuer le travail

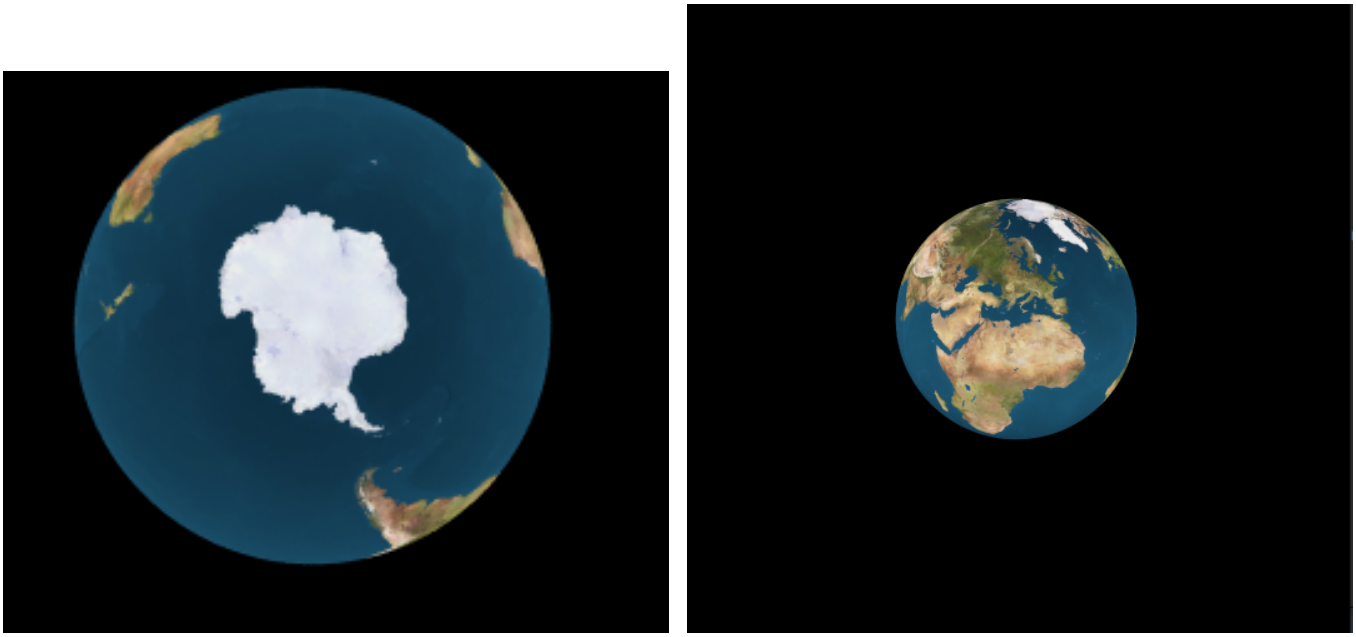

Figure 17: Texture "map monde" plaquée sur une sphère

dans le but de découvrir et expérimenter les notions encore nouvelles induite par la réalisation des tâches optionnelles notamment.