



## 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.

```
1 Vec3 rayTrace( Ray const & rayStart )
2 {
3     Vec3 color;
4     color = rayTraceRecursive(rayStart, 1);
5     return color;
6 }
7
8 Vec3 rayTraceRecursive( Ray ray , int NRemainingBounces)
9 {
10    RaySceneIntersection raySceneIntersection = computeIntersection(ray, 4.9f);
11    Vec3 color;
12    if(raySceneIntersection.intersectionExists)
13    {
14        if(raySceneIntersection.typeOfIntersectedObject == ObjectType_Sphere)
15        {
16            color = spheres[raySceneIntersection.objectIndex].material.diffuse_material;
17        }
18        else if(raySceneIntersection.typeOfIntersectedObject == ObjectType_Square)
19        {
20            color = squares[raySceneIntersection.objectIndex].material.diffuse_material;
21        }
22    }
23    return color;
24 }
```

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.

```

1 RaySceneIntersection computeIntersection(Ray const & ray, float minDist)
2 {
3     RaySceneIntersection result;
4     float t = FLT_MAX;
5     for(size_t i = 0; i<spheres.size(); i++)
6     {
7         RaySphereIntersection intersection = spheres[i].intersect(ray);
8         if(intersection.intersectionExists && intersection.t < t && intersection.t >
minDist)
9         {
10             t = intersection.t;
11             result.intersectionExists = true;
12             result.typeOfIntersectedObject = ObjectType_Sphere;
13             result.objectIndex = i;
14             result.t = t;
15             result.raySphereIntersection = intersection;
16         }
17     }
18     for(size_t i = 0; i<squares.size(); i++)
19     {
20         RaySquareIntersection intersection = squares[i].intersect(ray);
21         if(intersection.intersectionExists && intersection.t < t && intersection.t >
minDist)
22         {
23             t = intersection.t;
24             result.intersectionExists = true;
25             result.typeOfIntersectedObject = ObjectType_Square;
26             result.objectIndex = i;
27             result.t = t;
28             result.raySquareIntersection = intersection;
29         }
30     }
31 }
32 return result;
33 }

```

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 :

$$t^2 d \cdot d + 2td \cdot (o - c) + \|o - c\|^2 - r^2 = 0 \quad (1)$$

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.

```

1 RaySphereIntersection intersect(const Ray &ray) const {
2     RaySphereIntersection rayinter;
3     Vec3 C0 = ray.origin() - m_center;
4     double a = Vec3::dot(ray.direction(), ray.direction());
5     double b = 2 * Vec3::dot(ray.direction(), C0);
6     double c = pow(C0.length(), 2) - pow(m_radius, 2);
7     double disc = pow(b,2) - 4*a*c;
8     if(disc < 0 || (2*a) == 0)
9     {
10        rayinter.intersectionExists = false;
11        return rayinter;
12    }
13    double t1 = (b*(-1) - sqrt(disc))/(2*a);
14    double t2 = (b*(-1) + sqrt(disc))/(2*a);
15    if(t1>0 && (t1<t2 || t2<0))
16    {
17        rayinter.intersectionExists = true;
18        rayinter.t = t1;
19        Vec3 point = ray.direction() * t1 + ray.origin();
20        rayinter.intersection = point;
21        Vec3 normal = point - m_center;
22        normal.normalize();
23        rayinter.normal = normal;
24    }
25    }
26    else{
27        rayinter.intersectionExists = true;
28        rayinter.t = t2;
29        Vec3 point = ray.direction() * t2 + ray.origin();
30        rayinter.intersection = point;
31        Vec3 normal = point - m_center;
32        normal.normalize();
33        rayinter.normal = normal;
34    }
35    }
36    return rayinter;
37 }

```

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

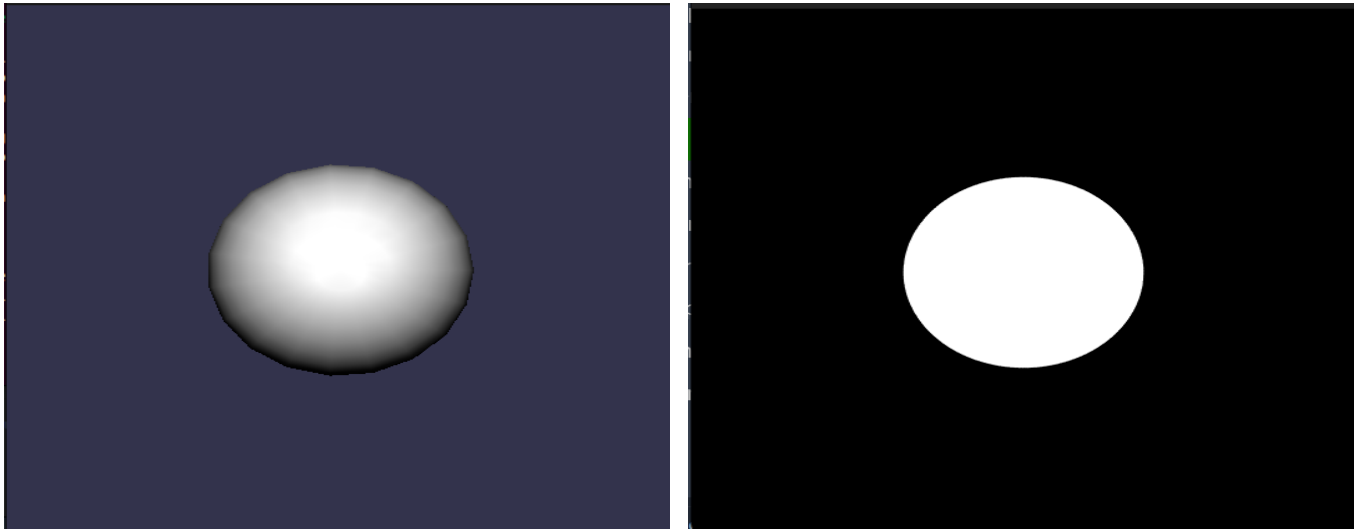


Figure 1: Sphère seule

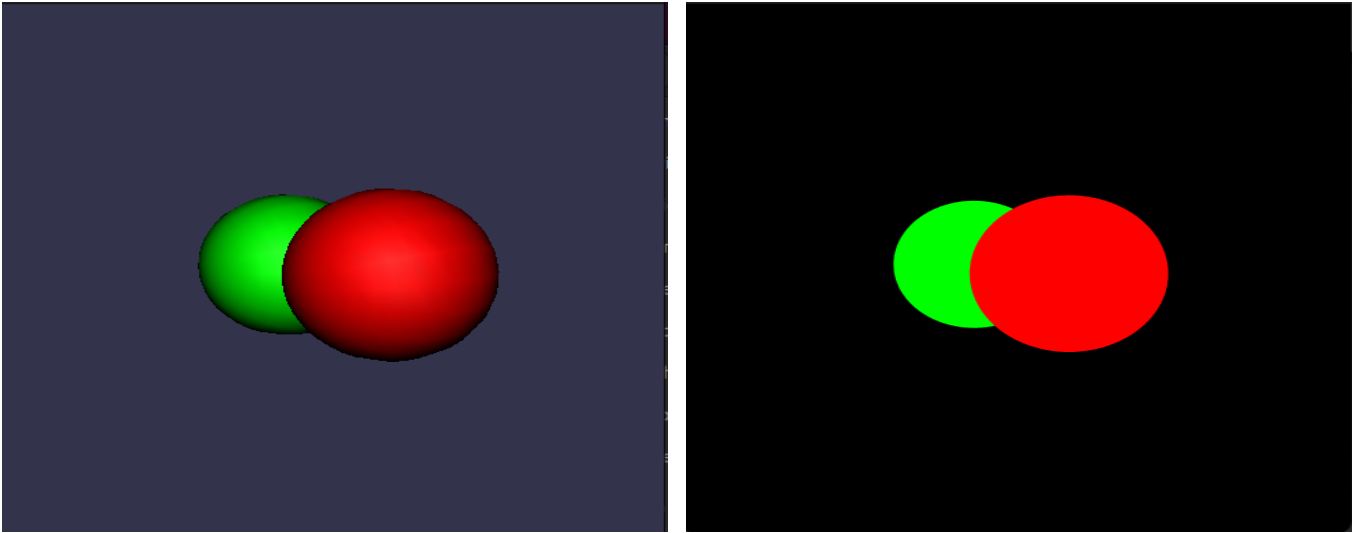


Figure 2: Deux sphères

### 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):

$$p = o - a \quad (2)$$

$$t = \frac{p \cdot n}{d \cdot n} \quad (3)$$

telle que :

- $a$  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.

```

1 bool isInSquare(Vec3 pos, Vec3 normal) const
2 {
3
4     Vec3 bottom = Vec3::cross(bottomRight() - bottomLeft(), pos - bottomLeft());
5     Vec3 right = Vec3::cross(upRight() - bottomRight(), pos - bottomRight());
6     Vec3 up = Vec3::cross(upLeft() - upRight(), pos - upRight());
7     Vec3 left = Vec3::cross(bottomLeft() - upLeft(), pos - upLeft());
8
9     bool direction = (Vec3::dot(bottom, normal) > 0);
10
11     bool sameDirection = (Vec3::dot(right, normal) > 0) == direction &&
12                          (Vec3::dot(left, normal) > 0) == direction &&
13                          (Vec3::dot(up, normal) > 0) == direction;

```

```

14     return sameDirection;
15 }

1 RaySquareIntersection intersect(const Ray &ray) const {
2     RaySquareIntersection intersection;
3     Vec3 p = bottomLeft() - ray.origin();
4     float t = ( Vec3::dot(p, normal()))/Vec3::dot(ray.direction(), normal());
5     Vec3 point = ray.direction() * t + ray.origin();
6     if(t>0 && isInSquare(point, normal()))
7     {
8         intersection.intersectionExists = true;
9         intersection.t = t;
10        intersection.intersection = point;
11        intersection.normal = normal();
12        intersection.normal.normalize();
13        return intersection;
14    }
15    else
16    {
17        intersection.intersectionExists = false;
18        return intersection;
19    }
20 }
21 }

```

On a donc le résultat suivant.

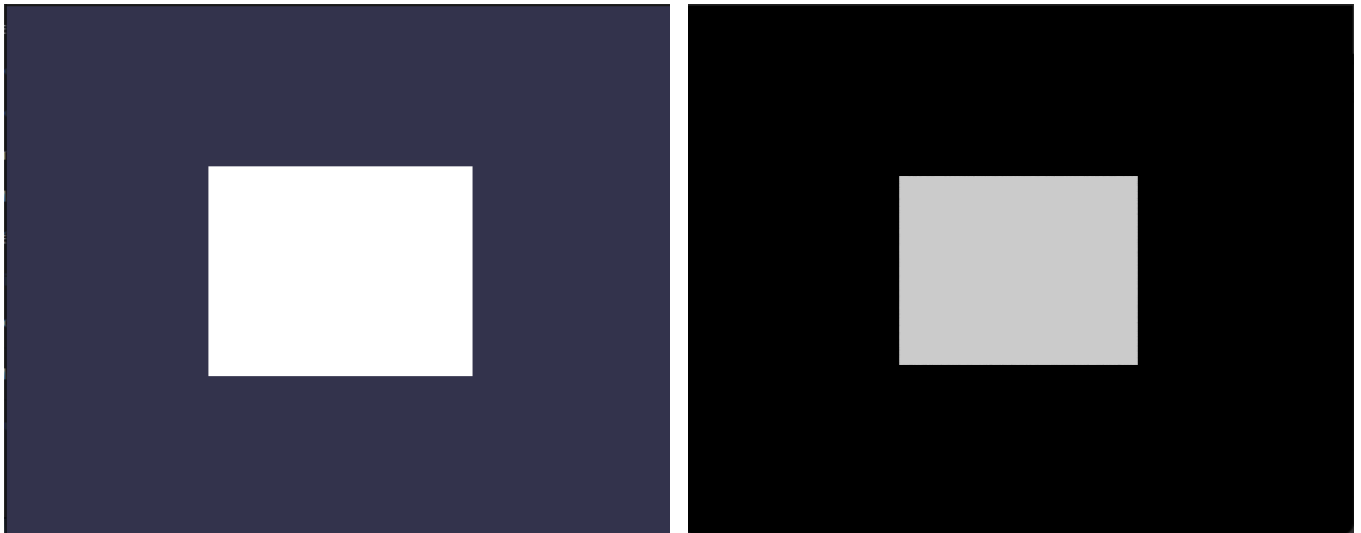


Figure 3: Carré

## 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.

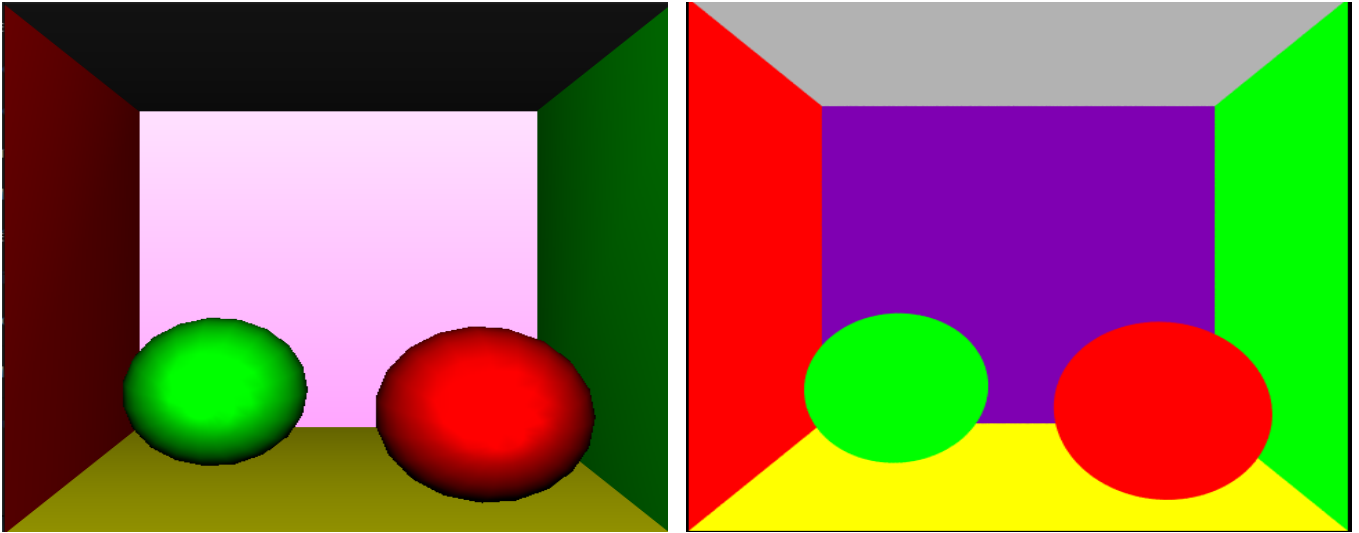


Figure 4: Boîte de Cornell

## 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.

```

1 Vec3 computePhongIllumination(
2     Ray ray,
3     Light light,
4     Vec3 normal,
5     Vec3 intersection,
6     Vec3 ambient_material,
7     Vec3 diffuse_material,
8     Vec3 specular_material,
9     double shininess)
10 {
11     Vec3 L = light.pos - intersection;
12     L.normalize();
13     Vec3 reflexion = 2 * (Vec3::dot(L, normal)) * normal - L;
14     reflexion.normalize();
15     Vec3 rayDir = ray.origin() - intersection;
16     rayDir.normalize();
17     float alpha = std::max(Vec3::dot(reflexion, rayDir), 0.f);
18     float theta = std::max(Vec3::dot(L, normal), 0.f);
19     Vec3 ambient = Vec3::compProduct(light.material, ambient_material);
20     Vec3 specular = Vec3::compProduct(
21         light.material,
22         specular_material) *
23         pow(alpha, shininess);
24     Vec3 diffuse = Vec3::compProduct(
25         light.material,
26         diffuse_material) * theta;
27
28     return ambient + specular + diffuse;
29 }

```

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

```
1 Vec3 rayTraceRecursive( Ray ray , int NRemainingBounces)
2 {
3     RaySceneIntersection raySceneIntersection = computeIntersection(ray, 4.9f);
4     Vec3 color;
5     if(raySceneIntersection.intersectionExists)
6     {
7         if(raySceneIntersection.typeOfIntersectedObject == ObjectType_Sphere)
8             color = computePhongIllumination(/*...*/);
9         else if(raySceneIntersection.typeOfIntersectedObject == ObjectType_Square)
10            color = computePhongIllumination(/*...*/);
11     }
12     return color;
13 }
```

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

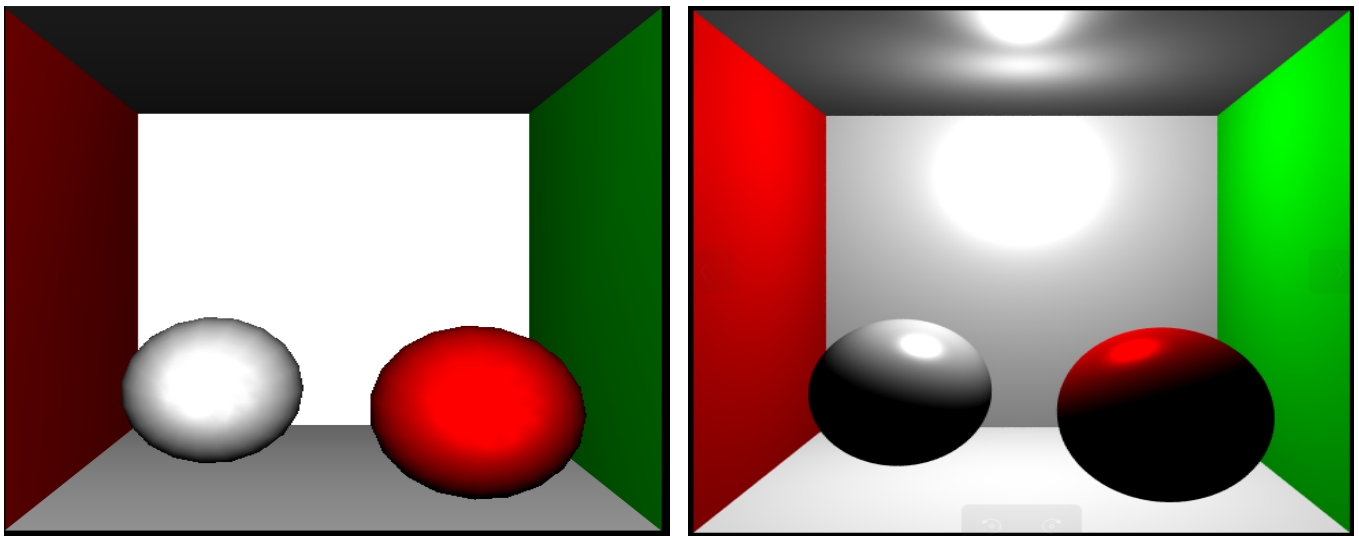


Figure 5: Illumination de Phong

### 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.

```
1 RaySceneIntersection computeShadowIntersection(Ray const & ray, float maxDist)
2 {
3     RaySceneIntersection result;
4     for(size_t i = 0; i<spheres.size(); ++i)
5     {
6         RaySphereIntersection intersection = spheres[i].intersect(ray);
7         if(intersection.intersectionExists && intersection.t > 0.001f && intersection.t <
8            maxDist)
9         {
10            result.intersectionExists = true;
11            return result;
12        }
13    }
```

```

12 }
13 for(size_t i = 0; i<squares.size(); ++i)
14 {
15     RaySquareIntersection intersection = squares[i].intersect(ray);
16     if(intersection.intersectionExists && intersection.t > 0.001f && intersection.t <
maxDist)
17     {
18         result.intersectionExists = true;
19         return result;
20     }
21 }
22
23 return result;
24 }
25
26 Vec3 computeHardShadow(Ray ray, Light light, Vec3 normal,
27     Vec3 intersection,
28     Vec3 ambient_material,
29     Vec3 diffuse_material,
30     Vec3 specular_material,
31     double shininess)
32 {
33     Vec3 L = light.pos - intersection;
34     float shadowMinDist = L.length();
35     L.normalize();
36     Ray shadowRay = Ray(intersection, L);
37     RaySceneIntersection shadowIntersection = computeShadowIntersection(shadowRay,
shadowMinDist);
38     if(shadowIntersection.intersectionExists)
39         return Vec3(0,0,0);
40     else
41         return computePhongIllumination(ray, light, normal, intersection,
ambient_material, diffuse_material, specular_material, shininess);
42 }
43 }

```

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

```

1 Vec3 rayTraceRecursive( Ray ray , int NRemainingBounces) {
2     RaySceneIntersection raySceneIntersection = computeIntersection(ray, 4.9f);
3     Vec3 color;
4     int light_number = 0;
5     if(raySceneIntersection.intersectionExists)
6     {
7
8         if(raySceneIntersection.typeOfIntersectedObject == ObjectType_Sphere)
9         {
10             for (int i = 0; i < lights.size(); ++i)
11             {
12                 color += computeHardShadow(/*...*/);
13             }
14         }
15         else if(raySceneIntersection.typeOfIntersectedObject == ObjectType_Square)
16         {
17
18             for (int i = 0; i < lights.size(); ++i)
19             {
20                 color += computeHardShadow(/*...*/);
21             }
22         }
23
24     }
25 }

```



```

25     return color;
26 }

```

On obtient finalement le résultat suivant

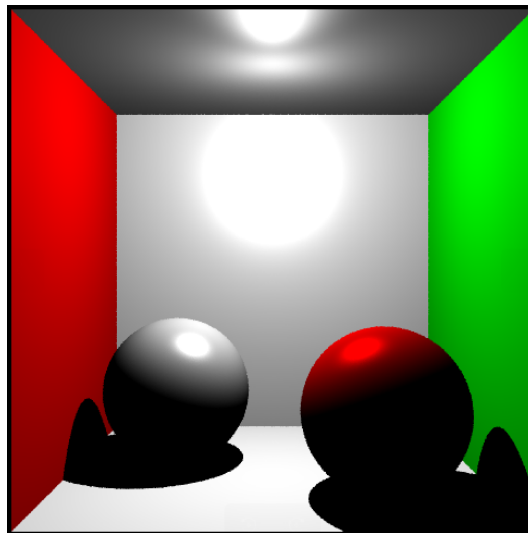


Figure 6: Calcul des ombres

### 3.3 Ombres Douces

Le calcul des ombres douces nécessite 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.

```

1 float computeSoftShadow(Light light, Vec3 intersection)
2 {
3
4     float counter = 0;
5     for(float k = 0; k<LIGHT_SAMPLES; k++)
6     {
7         float x =(float)(rand()/((float)(RAND_MAX / (light.radius))));
8         float y =(float)(rand()/((float)(RAND_MAX / (light.radius))));
9         float z =(float)(rand()/((float)(RAND_MAX / (light.radius))));
10        Vec3 pos = Vec3(light.pos[0] + x, light.pos[1] + y, light.pos[2] + z);
11
12        Vec3 L = pos - intersection;
13        L.normalize();
14        Ray shadowRay = Ray(intersection, L);
15        RaySceneIntersection shadowIntersection = computeShadowIntersection(shadowRay,
16        0.9f);
17        if(shadowIntersection.intersectionExists) counter++;
18    }
19    float shadow = counter/LIGHT_SAMPLES;
20    return (1.0f - shadow);
21 }

```

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

```

1
2 Vec3 rayTraceRecursive( Ray ray , int NRemainingBounces, float minDist)

```

```

3 {
4     RaySceneIntersection raySceneIntersection = computeIntersection(ray, minDist);
5     Vec3 color;
6     if(!raySceneIntersection.intersectionExists) return color;
7     if(raySceneIntersection.typeOfIntersectedObject == ObjectType_Sphere)
8     {
9         Vec3 normal = raySceneIntersection.raySphereIntersection.normal;
10        Vec3 point = raySceneIntersection.raySphereIntersection.intersection;
11        float intersected = 0;
12        for (int i = 0; i < lights.size(); ++i)
13        {
14            color = computePhongIllumination(/*...*/);
15            float shadow = computeSoftShadow(lights[i], point);
16            color *= shadow;
17        }
18    }
19
20    /* Repete pour chacun des types d'objets */
21 }

```

On obtient donc les résultats suivants.

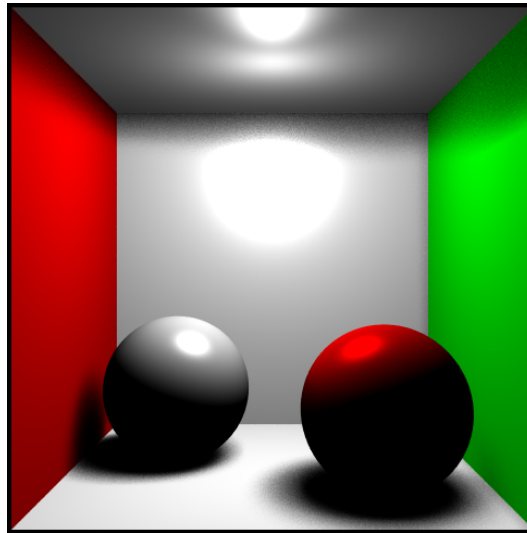


Figure 7: Ombres douces

## 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.

```

1 bool isParallelTo( Line const & L ) const {
2     bool result;
3     float dirDotNorm = Vec3::dot(L.direction(), normal());
4     result = (dirDotNorm == 0);
5     return result;
6 }
7 Vec3 getIntersectionPointWithSupportPlane( Line const & L, float &_t) const {
8     Vec3 result;
9     if(! isParallelTo(L))
10    {
11        _t = ( Vec3::dot((c0() - L.origin()), normal()))/Vec3::dot(L.direction(), normal
12        ());
13        if(_t>0) result = L.direction() * _t + L.origin();
14    }
15    return result;
16 }

```

#### 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 coordonnées barycentriques du point d'intersection, si ces coordonnées ont une valeur comprise entre 0 et 1 inclus et que leur somme vaut 1, alors le point est dans le triangle. *Coordonnées 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

```

1 void computeBarycentricCoordinates( Vec3 const & p , float & u0 , float & u1 , float &
2   u2 , bool & isInside) const {
3     Vec3 c0c1 = (c1() - c0());
4     Vec3 c1c2 = (c2() - c1());
5     Vec3 c2c0 = (c0() - c2());
6
7     Vec3 c0pc1normal = Vec3::cross(c0c1, (p-c0()));
8     Vec3 c1pc2normal = Vec3::cross(c1c2, (p-c1()));
9     Vec3 c2pc0normal = Vec3::cross(c2c0, (p-c2()));
10
11     isInside = (Vec3::dot(c1pc2normal, normal()) > 0) && (Vec3::dot(c0pc1normal, normal
12     ()) > 0) && (Vec3::dot(c2pc0normal, normal()) > 0);
13
14     float area0 = c1pc2normal.length()/2.f;
15     float area1 = c2pc0normal.length()/2.f;
16     float area2 = c0pc1normal.length()/2.f;
17
18     u0 = area0/this->area;
19     u1 = area1/this->area;
20     u2 = area2/this->area;
21 }

```

Finalement on a pour l'intersection rayon-triangle

```

1 RayTriangleIntersection getIntersection( Ray const & ray ) const {
2     RayTriangleIntersection result;

```

```

3 Line rayLine = (Line) ray;
4 if(isParallelTo(rayLine))
5 {
6     result.intersectionExists = false;
7     return result;
8 }
9 float t;
10 Vec3 planeIntersection = getIntersectionPointWithSupportPlane(rayLine, t);
11 if(t<0)
12 {
13     result.intersectionExists = false;
14     return result;
15 }
16 float w0,w1,w2;
17 bool isInside;
18 computeBarycentricCoordinates(planeIntersection, w0,w1,w2, isInside);
19 if(( 0.f<=w0 && w0<=1.f) &&
20    ( 0.f<=w1 && w1<=1.f) &&
21    ( 0.f<=w2 && w2<=1.f) && isInside)
22 {
23     result.intersectionExists = true;
24     result.t = t;
25     result.w0 = w0;
26     result.w1 = w1;
27     result.w2 = w2;
28     result.intersection = Vec3(planeIntersection[0] * w0, planeIntersection[1] * w1,
29     planeIntersection[2] * w2);
30     Vec3 norm = Vec3(normal()[0], normal()[1], normal()[2]);
31     norm.normalize();
32     result.normal = norm;
33 }
34 else
35 {
36     result.intersectionExists = false;
37     return result;
38 }
39 }

```

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.

```

1 RayTriangleIntersection intersect( Ray const & ray ) const {
2     RayTriangleIntersection closestIntersection;
3     float triangleScaling = 0.0001f;
4     closestIntersection.t = FLT_MAX;
5     for (int i = 0; i < triangles.size(); ++i)
6     {
7         MeshTriangle meshTriangle = triangles[i];
8         Triangle triangle = Triangle(vertices[triangles[i][0]].position *
9         triangleScaling,
10                                     vertices[triangles[i][1]].position *
11                                     triangleScaling,
12                                     vertices[triangles[i][2]].position *
13                                     triangleScaling);
14         RayTriangleIntersection intersection = triangle.getIntersection(ray);
15         if(intersection.intersectionExists && intersection.t<closestIntersection.t &&
16            intersection.t> 0.001f) closestIntersection = intersection;
17     }
18 }

```

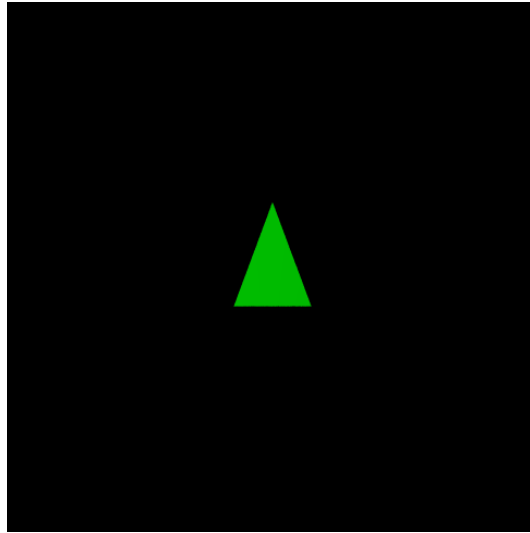


Figure 8: Triangle

```

13     }
14     return closestIntersection;
15 }

```

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

```

1 void setup_single_mesh(){
2     meshes.clear();
3     spheres.clear();
4     squares.clear();
5     lights.clear();
6     {
7         lights.resize( lights.size() + 1 );
8         Light & light = lights[lights.size() - 1];
9         light.pos = Vec3(-5,5,5);
10        light.radius = 2.5f;
11        light.powerCorrection = 2.f;
12        light.type = LightType_Spherical;
13        light.material = Vec3(1,1,1);
14        light.isInCamSpace = false;
15    }
16    {
17        meshes.resize( meshes.size() + 1 );
18        Mesh & mesh = meshes[meshes.size() - 1];
19        mesh.loadOFF("./data/suzanne.off");
20        mesh.centerAndScaleToUnit();
21        mesh.build_arrays();
22        mesh.material.type = Material_Diffuse_Blinn_Phong;
23        mesh.material.diffuse_material = Vec3( 0.,1.,0.);
24        mesh.material.specular_material = Vec3( 0.2,0.2,0.2 );
25        mesh.material.shininess = 20;
26    }
27 }

```

On obtient alors les résultats suivants.

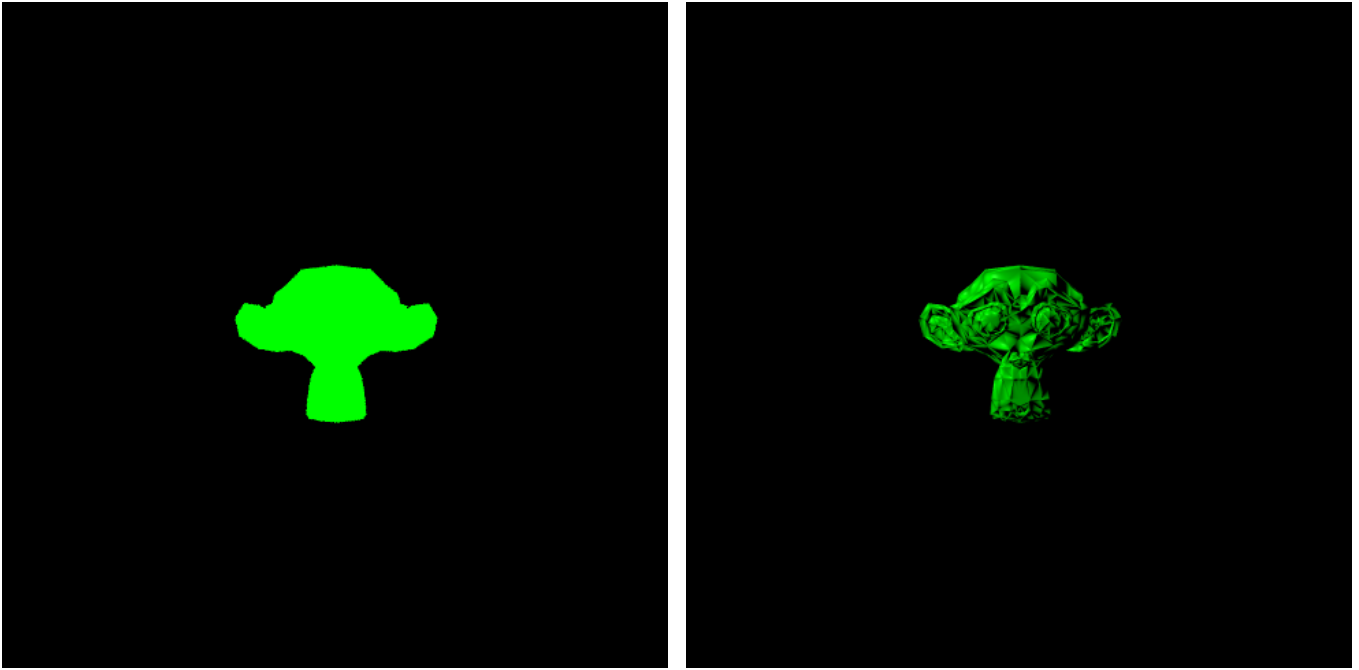


Figure 9: Suzanne

## 5 Phase finale

### 5.1 Sphère réfléchissante

Pour pouvoir rendre une sphère réfléchissante, 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 :

$$R = d - (2 * n * d.n); \quad (4)$$

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 :

```

1 Vec3 rayTraceRecursive( Ray ray , int NRemainingBounces , float minDist)
2 {
3     if(NRemainingBounces == 0) return Vec3();
4     /*...*/
5     if(spheres[raySceneIntersection.objectIndex].material.type == Material_Mirror)
6     {
7         Vec3 reflectionDir = ray.direction() - (2 * normal * Vec3::dot(ray.direction(),
8         normal));
9         reflectionDir.normalize();
10        Ray reflected = Ray(point, reflectionDir);
11        color += rayTraceRecursive(reflected, NRemainingBounces - 1, 0.0001f);
12    }
13    /*...*/
14 }
```

On obtient alors le résultat suivant :

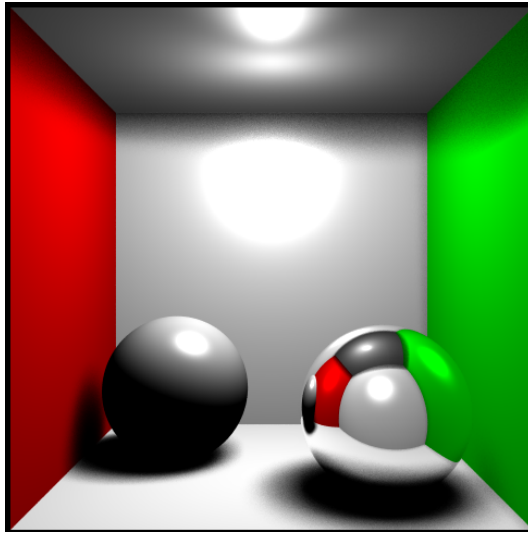


Figure 10: Sphère réfléchissante

## 6 Conclusion

Les valeurs des normales et des  $u,v$  ne sont pas encore interpolées pour les triangles. Le calcul de la réfraction est en cours de développement.