



UNIVERSITÉ  
DE MONTPELLIER



HAI719I – Programmation 3D

---

## Projet de lancer de rayon

---

Andrew Mansour

Faculté des Sciences de Montpellier

2024/2025  
M1 Imagine

# Table des matières

<b>1 Première Phase</b>	<b>1</b>
1.1 Fonctions de Scène . . . . .	1
1.2 Sphère . . . . .	2
1.3 Carré . . . . .	3
<b>2 Seconde Phase</b>	<b>5</b>
2.1 Shader de Phong . . . . .	5
2.2 Ombres simples . . . . .	7
2.3 Ombres douces . . . . .	7
<b>3 Troisième Phase</b>	<b>11</b>
3.1 3D Mesh . . . . .	11
3.1.1 Triangle intersection . . . . .	11
3.1.2 Mesh intersection . . . . .	13
3.2 Interpolation de sommets (Incomplet) . . . . .	14
3.3 Réflexion . . . . .	15
<b>4 Quatrième Phase</b>	<b>16</b>
4.1 Réfraction . . . . .	16
4.2 Structure d'accélération : Kd-Tree . . . . .	18
4.2.1 Bounding Box . . . . .	18
4.2.2 Kd-Tree . . . . .	19
4.2.3 Results . . . . .	22
<b>5 Phase Finale</b>	<b>22</b>
5.1 Motion Blur . . . . .	22
5.2 Rendus en plus . . . . .	25

# 1 Première Phase

## 1.1 Fonctions de Scène

La fonction computeIntersection calcule les intersections et retourne une Intersection de scène avec toutes ses valeurs nécessaires. Le résultat retourné sera l'intersection la plus proche de la camera trouvée en prenant la valeur t la plus petite (avec t étant le point d'intersection).

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

Listing 1 – Compute Intersection

La fonction rayTraceRecursive a pour but de calculer récursivement la couleur des objets, le nombre de bounce va simplement augmenter les valeurs des couleurs. Avec un nombre de bounce assez élevé, on peut obtenir une image blanche.

```
1 Vec3 rayTraceRecursive( Ray ray , int NRemainingBounces ) {
2     if (NRemainingBounces <= 0) {
3         return Vec3(0., 0., 0.); // Return black if no more bounces
4     }
5
6     RaySceneIntersection raySceneIntersection = computeIntersection(ray);
7     Vec3 color(0., 0., 0.);
```

```

9 if (raySceneIntersection.intersectionExists) {
10     switch (raySceneIntersection.typeOfIntersectedObject) {
11         case 0:
12             color = spheres[raySceneIntersection.objectIndex].material.
13                 diffuse_material;
14             break;
15         case 1:
16             color = squares[raySceneIntersection.objectIndex].material.
17                 diffuse_material;
18             break;
19     }
20
21     Vec3 recursiveColor = rayTraceRecursive(ray, NRemainingBounces - 1);
22     color = color + recursiveColor;
23
24 } else {
25     color = Vec3(1., 1., 1.);
26 }
27
28 return color;
29 }
```

Listing 2 – Raytrace recursive

Cette fonction appelle simplement rayTraceRecursive et retourne la couleur.

```

1
2 Vec3 rayTrace( Ray const & rayStart ) {
3     Vec3 color;
4     color = rayTraceRecursive(rayStart , 1);
5     return color;
6 }
```

Listing 3 – Raytrace

## 1.2 Sphère

L'intersection d'une sphère est calculée en utilisant la formule du cours.

```

1 RaySphereIntersection intersect(const Ray &ray) const {
2     RaySphereIntersection intersection;
3     //TODO calcul l'intersection rayon sphere
4     intersection.intersectionExists = false;
5     Vec3 o = ray.origin();
6     Vec3 d = ray.direction();
7     float a = Vec3::dot(d,d);
8     float b = Vec3::dot(d , (o - m_center)) * 2;
9     float c = Vec3::dot((o-m_center),(o-m_center)) - pow(m_radius , 2);
10    float determinant = pow(b,2) - (4 * a * c);
11    if(determinant > 0){
12        intersection.intersectionExists = true;
13        float x1 = (-b - sqrt(determinant))/2*a;
14        float x2 = (-b + sqrt(determinant))/2*a;
```

```

15     if(x1 < 0 || x2 < 0 ){
16         intersection.t = std::max(x1,x2);
17     }
18     else{
19         intersection.t = std::min(x1,x2);
20     }
21 }
22 return intersection;
23 }
```

Listing 4 – Modification dans le switch

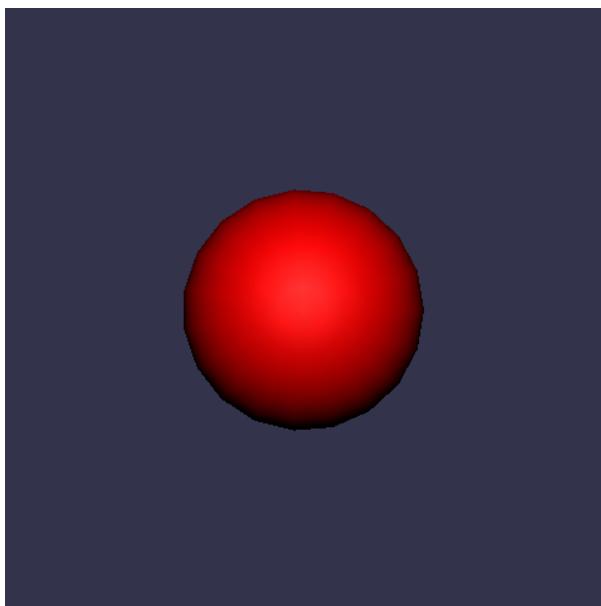


FIGURE 1 – Maillage de la sphère

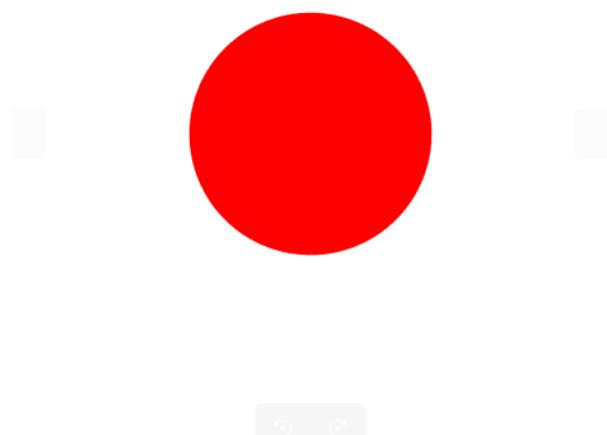


FIGURE 2 – Sphère générée par Raytracing

### 1.3 Carré

Les premières quelques lignes de la fonction servent à corriger une erreur de normales des carrés qui a notamment un impact pour la scène de la cornell box, comme indiqué dans la ligne commentée, cette solution a été trouvée par un camarade.

Source : <https://computergraphics.stackexchange.com/questions/8418/get-intersection-ray-with>

```

1 RaySquareIntersection intersect(const Ray &ray) const {
2     RaySquareIntersection intersection;
3
4     //Solution trouvée par un camarade pour régler les soucis d'affichage de la
5     //cornell box
6     Vec3 m_bottom_left = vertices[0].position;
7     Vec3 m_right_vector = vertices[1].position - vertices[0].position;
8     Vec3 m_up_vector = vertices[3].position - vertices[0].position;
9     Vec3 m_normal = Vec3::cross(m_right_vector, m_up_vector);
10    m_normal.normalize();
```

```

11     Vec3 o = ray.origin();
12     Vec3 d = ray.direction();
13     Vec3 n = m_normal;
14     Vec3 pos = m_bottom_left;
15     float denom = Vec3::dot(d,n);
16     if(denom >= 0){
17         intersection.intersectionExists = false;
18         return intersection;
19     }
20     float t = Vec3::dot((pos - o),n) / denom;
21     Vec3 intersectPoint = o + t*d;
22     Vec3 v = intersectPoint - pos;
23     Vec3 e1 = m_right_vector;
24     Vec3 e2 = m_up_vector;
25     float width = e1.length();
26     float height = e2.length();
27     float u1 = Vec3::dot(v,e1) / width;
28     float u2 = Vec3::dot(v,e2) / height;
29
30     if( (u1 < width && u1 > 0.f) && (u2 < height && u2 > 0.f) && t > 0.f){
31         intersection.intersectionExists = true;
32         intersection.t = t;
33         intersection.u = u1;
34         intersection.v = u2;
35         intersection.intersection = intersectPoint;
36         intersection.normal = m_normal;
37         return intersection;
38     }
39     intersection.intersectionExists = false;
40     return intersection;
41 }
```

Listing 5 – Modification dans le switch

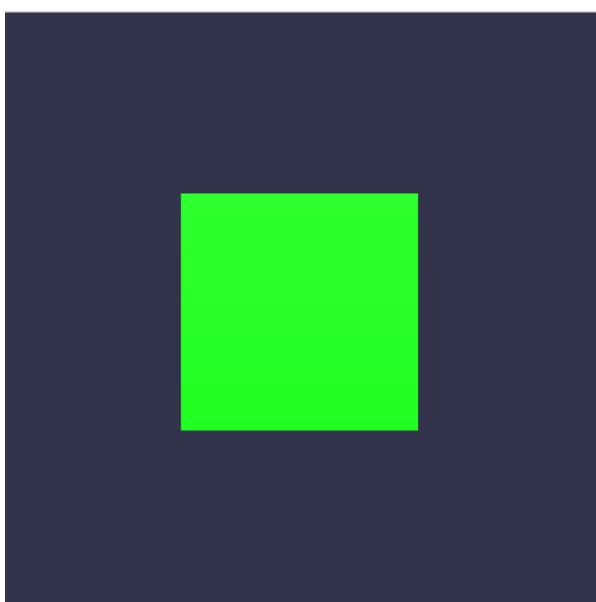


FIGURE 3 – Maillage du carré

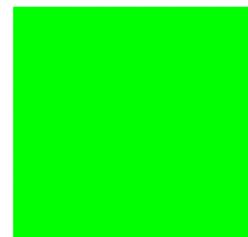


FIGURE 4 – Carré générée par Raytracing

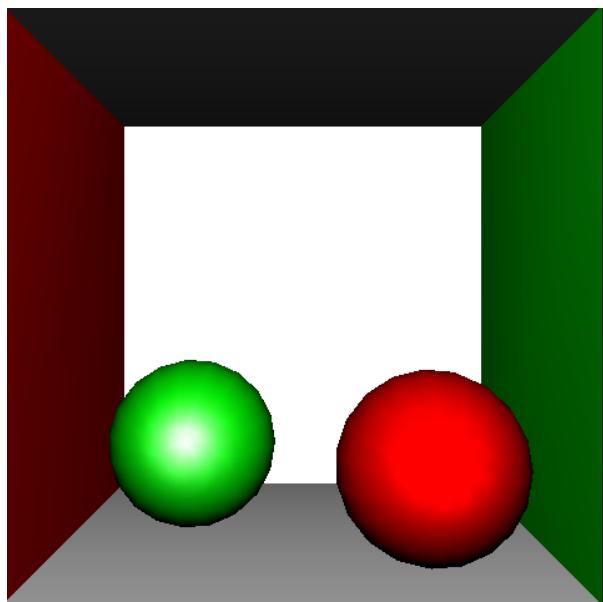
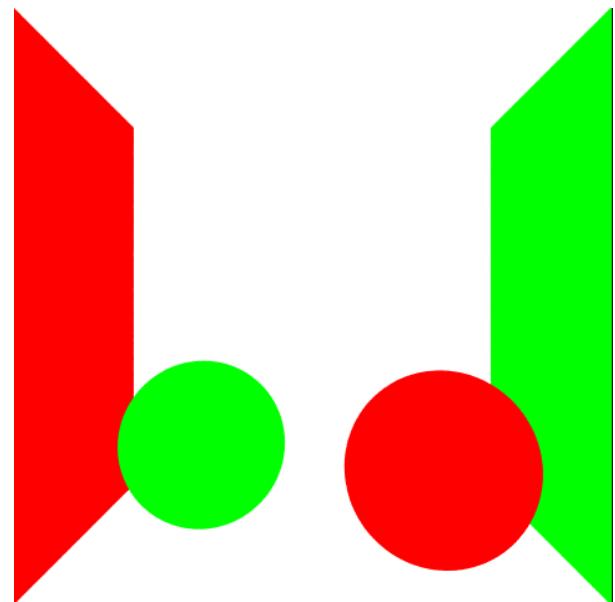


FIGURE 5 – Maillage de la Cornell Box sans cou-



leurs

FIGURE 6 – Cornell box sans courleurs générée par Raytracing

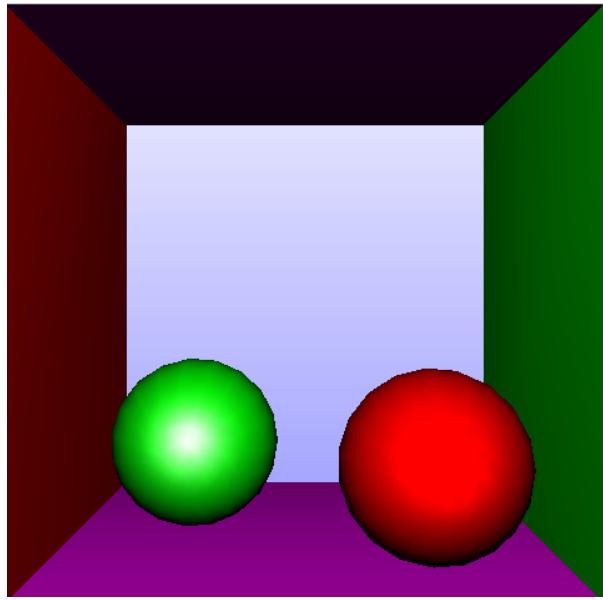
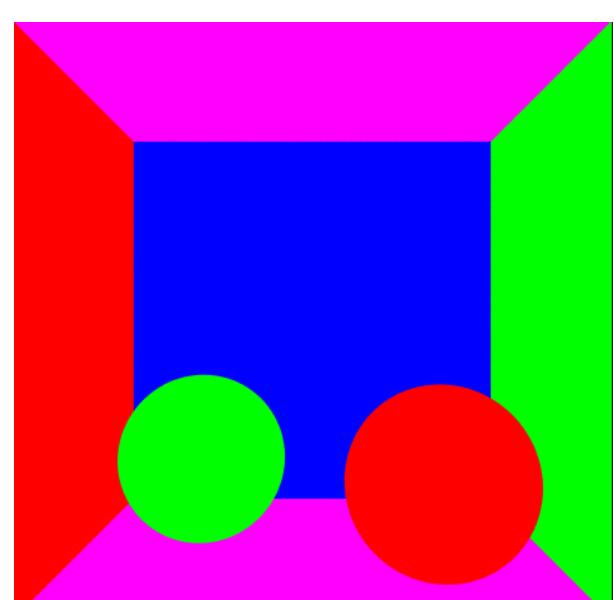


FIGURE 7 – Maillage de la Cornell Box avec cou-



leurs

FIGURE 8 – Cornell box générée par Raytracing

## 2 Seconde Phase

### 2.1 Shader de Phong

Les valeurs de Phong (shininess , specular et diffuse) présentes dans meshes[i].material sont récupérées et utilisées pour calculer le rendu avec un shader de Phong pour chaque source lumineuse de la scène. Les calculs d'intensité spéculaire, diffuse et ambiante ainsi que l'équation de Phong

ont été faits à partir de formules vues en cours.

```
1 Vec3 specular(0., 0., 0.);
2 Vec3 diffuse(0., 0., 0.);
3 Vec3 ambient(0., 0., 0.);
4 float shininess = 0;
5
6 Vec3 N = Vec3(0., 0., 0.);
7 Vec3 intersectionPoint = Vec3(0., 0., 0.);
8
9 switch (raySceneIntersection.typeOfIntersectedObject) {
10     case 0:
11         diffuse = spheres[raySceneIntersection.objectIndex].material.
12             diffuse_material;
13             //... meme chose pour les autres attributs
14     case 1:
15         diffuse = squares[raySceneIntersection.objectIndex].material.
16             diffuse_material;
17             //... meme chose pour les autres attributs
18 }
```

Listing 6 – Obtenir les valeurs pour Phong

```
1 for (int i = 0; i < lights.size(); i++) {
2     //Dir to light
3     Vec3 L = lights[i].pos - intersectionPoint;
4     L.normalize();
5
6     //Ambient
7     Vec3 Isa = Vec3(0.1, 0.1, 0.1);
8     Vec3 Ka = ambient;
9     Vec3 Ia = Vec3::compProduct(Isa, Ka);
10
11    //Diffuse
12    Vec3 Id = Vec3::compProduct(lights[i].material, diffuse) * std::max(0.f
13        , Vec3::dot(N, L)) ;
14
15    //Specular
16    Vec3 V = ray.origin() - intersectionPoint;
17    V.normalize();
18    Vec3 R = 2 * std::max(0.f, Vec3::dot(N, L)) * N - L;
19    Vec3 Iss = lights[i].material;
20    Vec3 Ks = specular;
21    Vec3 Is = Vec3::compProduct(Iss, Ks) * pow(std::max(0.f, Vec3::dot(R, V
22        )), shininess) ;
23
24    color += Ia + Id + Is;
25 }
```

Listing 7 – Equation de Phong

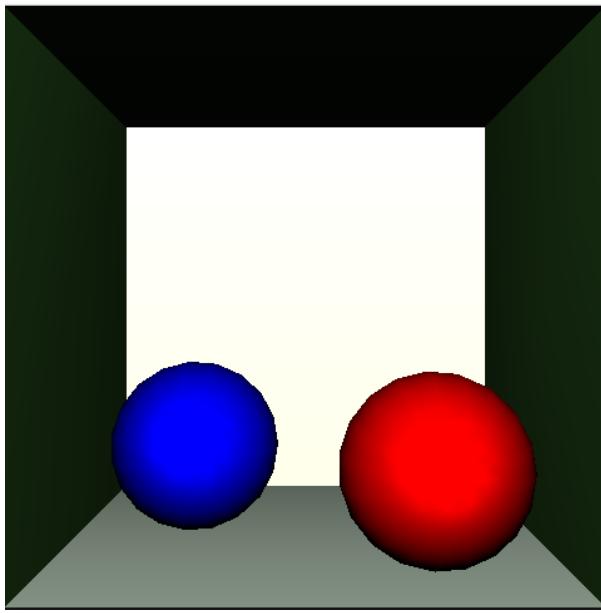


FIGURE 9 – Maillage de la cornell box

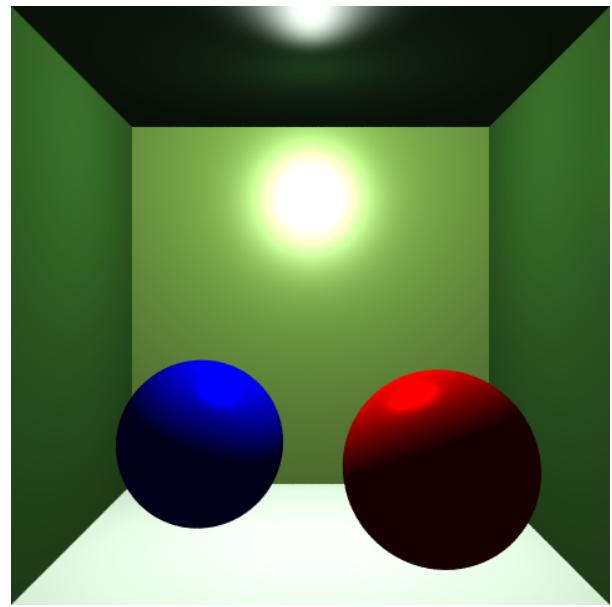


FIGURE 10 – Cornell box avec shading de Phong

## 2.2 Ombres simples

Les ombres simples qui sont implémentées ici sont définies comme-ci : Si le point d'intersection est dans l'ombre, alors il n'est pas éclairé. Pour ce faire, on lance un rayon d'ombre, légèrement décalé pour éviter que l'objet s'ombre lui-même. On va vérifier qu'il y existe une intersection, et que son point d'intersection est plus proche que la source lumineuse. Cette vérification implique que l'objet est obstrué par un autre, si c'est le cas, on met sa couleur en noir.

```

1 for(int i = 0 ; i < lights.size() ; i++){
2     Vec3 L = lights[i].pos - intersectionPoint;
3     L.normalize();
4
5     //Ombres simples
6
7     Ray shadowRay(intersectionPoint + N * 0.001f, L);
8     RaySceneIntersection shadowIntersection = computeIntersection(shadowRay);
9     float lightDistance = (lights[i].pos - intersectionPoint).length();
10
11    if (shadowIntersection.intersectionExists && shadowIntersection.t <
12        lightDistance && shadowIntersection.t > 0) {
13        color = Vec3(0., 0., 0.);
14        continue;
15    }

```

Listing 8 – Modifications de RaytraceRecursive pour les ombres

## 2.3 Ombres douces

L'approche utilisée ici pour appliquer des ombres douces est d'envoyer plusieurs ray d'ombre dans un cone vers la lumière pour essayer de simuler la façon donc la lumière se propage au contact

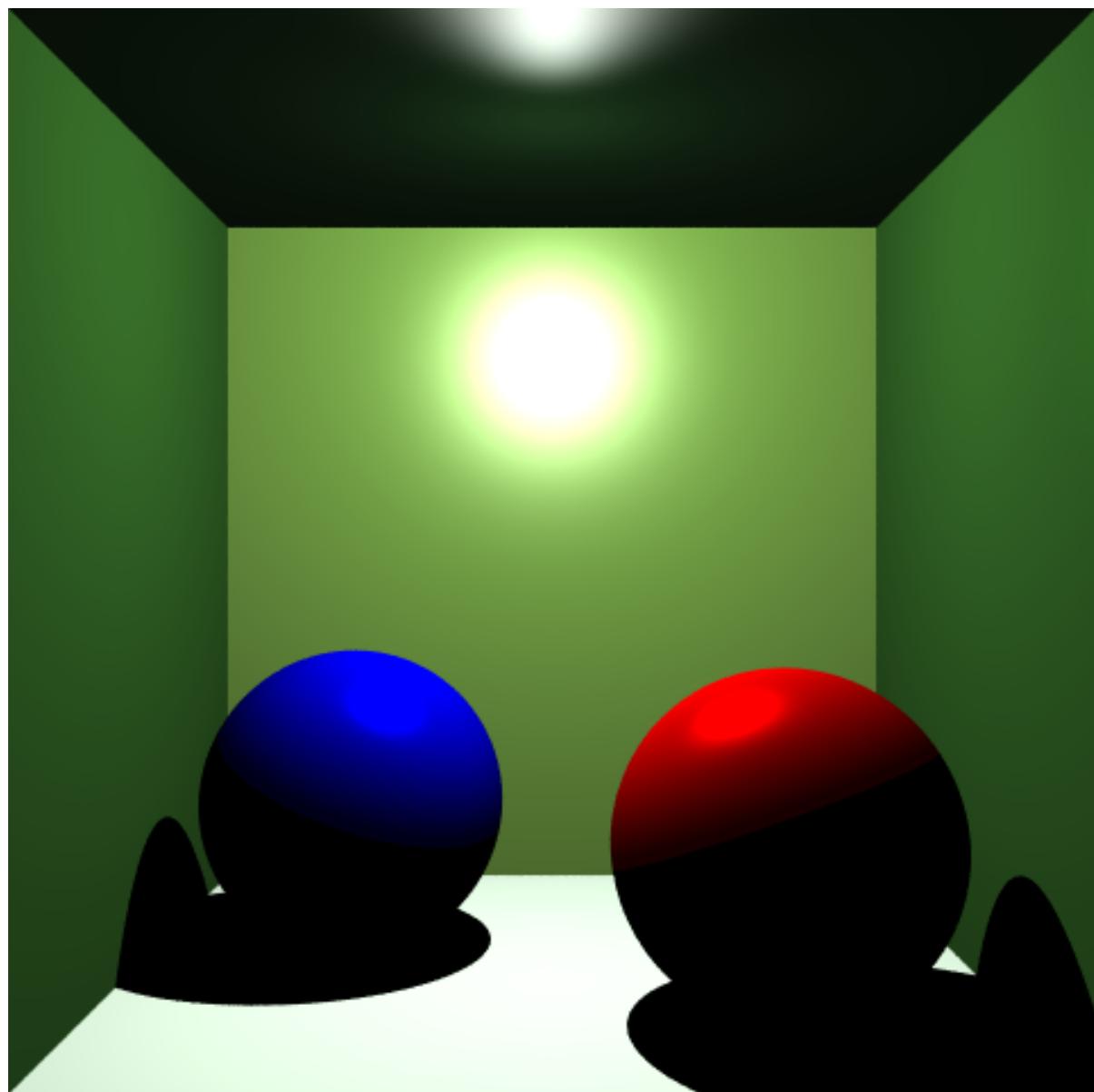


FIGURE 11 – Hard shadows

d'un object pour créer une ombre. Il est important de préciser que le cône ne fonctionne que sur une source lumineuse circulaire. On commence par trouver un vecteur perpendiculaire à la source lumineuse, calculé en faisant le produit vectoriel entre le vecteur direction du point d'intersection et la lumière, avec un vecteur vertical. Ensuite, on calcule le vecteur qui nous permet d'aller au bord de la source lumineuse et on fait le cosinus du produit vectoriel des deux vecteurs trouvés pour obtenir l'angle du cône dans lequel on souhaite envoyer la lumière. Lorsqu'on a le cône, on envoie un nombre choisi de ray (plus on envoie de ray, moins il y aura de bruit mais plus le temps de calcul est long) à des positions aléatoires dans le cône, si le ray d'ombre est obstrué, on augmente l'intensité de l'ombre. L'intensité est divisée par le nombre de ray envoyés pour obtenir un résultat plus lisse. Ce facteur va être appliqué aux intensités spéculaires et diffuses du pixel.

#### Sources :

- <https://medium.com/@alexander.wester/ray-tracing-soft-shadows-in-real-time-a53b836d1>
- *Ray tracing in One Weekend - Peter Shirley*

```

1 //Soft shadows
2 Vec3 perpendicular = Vec3::cross(L , Vec3(0.f , 1.f , 0.f));
3
4 //If up then perpendicular (1,0,0)
5 if(perpendicular.norm() == 0){
6     perpendicular = Vec3(1.f , 0.f , 0.f);
7 }
8
9 //Go to the edge of the lightsource to get the angle
10 Vec3 toLightEdge = (lights[i].pos + perpendicular * lights[i].radius) -
11     intersectionPoint;
12 toLightEdge.normalize();
13
14 //Get the angle (only works cuz we have circular lights)
15 float angle = acos(Vec3::dot(L , toLightEdge));
16
17 int numShadowRays = 16; //to go slow and pretty
18 //int numShadowRays = 5; //to go fast brrrr
19 float shadowFactor = 0.0f;
20
21 for (int j = 0; j < numShadowRays; j++) {
22     Vec3 randomDir = random_in_cone(L, angle);
23
24     Ray shadowRay(intersectionPoint + N * 0.001f, randomDir);
25
26     RaySceneIntersection shadowIntersection = computeIntersection(shadowRay
27         );
28     if (!shadowIntersection.intersectionExists || shadowIntersection.t >= (
29         lights[i].pos - intersectionPoint).length()) {
30         shadowFactor += 1.f;
31     }
32 }
33
34 shadowFactor /= numShadowRays;

```

Listing 9 – Modifications de RayTraceRecursive pour les ombres douces

```

1 Vec3 Id = Vec3::compProduct(lights[i].material , diffuse) * std::max(0.f ,
2     Vec3::dot(N, L)) * shadowFactor;

```

Listing 10 – Ombres appliquées à l'intensité spéculaire

```

1   Vec3 Is = Vec3::compProduct(Iss, Ks) * pow(std::max(0.f, Vec3::dot(R, V)),
    shininess) * shadowFactor;

```

Listing 11 – Ombres appliquées à l'intensité spéculaire

L'envoi de plus de ray d'ombre réduit fortement le bruitage présent dans cette ombre, mais augmente fortement le temps nécessaire pour générer l'image. L'envoi de 128 ray prend environ 200s et l'envoi d'une seule prend moins de 15s.

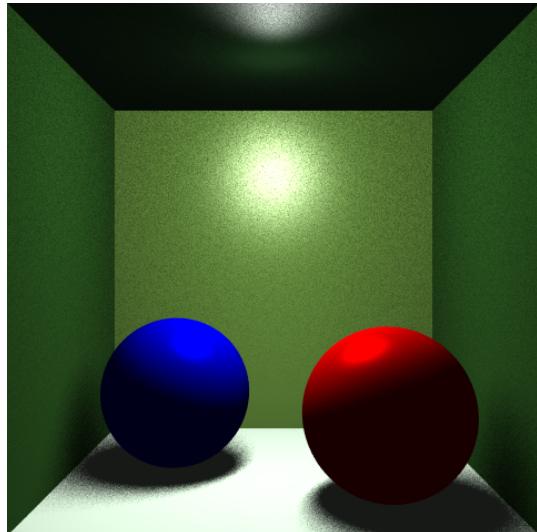


FIGURE 12 – Ombres douces avec 1 ray d'ombre

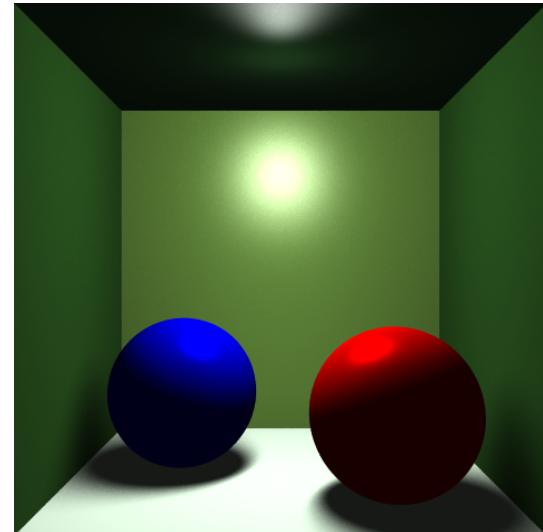


FIGURE 13 – Ombres douces avec 32 ray d'ombre

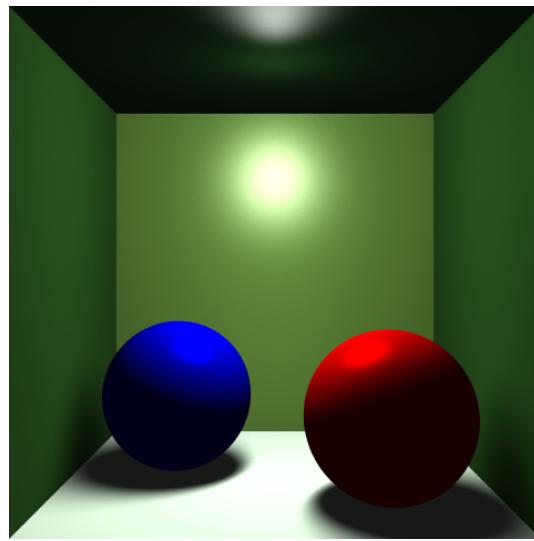


FIGURE 14 – Ombres douces avec 128 ray d'ombre

### 3 Troisième Phase

#### 3.1 3D Mesh

##### 3.1.1 Triangle intersection

Un maillage est composé de plusieurs triangles, pour trouver son intersection avec une droite, il faut donc savoir trouver l'intersection du triangle, mais celle-ci n'a pas été implémentée jusqu'ici. Pour ce faire, on suit les quatre étapes précisées dans la base de code. D'abord, on vérifie que le rayon n'est pas parallèle au plan du triangle (explication plus bas), en créant un plan avec la normale et les coordonnées du triangle. Ensuite, vérifie que le triangle est bien devant l'origine du rayon, en faisant le produit scalaire du point d'intersection avec le plan (explication plus bas) avec l'origine du rayon. Un résultat négatif implique qu'il est derrière l'origine. Puis, on utilise les coordonnées barycentriques (explication plus bas) pour s'assurer que le point d'intersection avec le plan est bien dans notre triangle, si les trois conditions sont validées, on a intersection.

```
1 RayTriangleIntersection getIntersection( Ray const & ray ) const {
2     RayTriangleIntersection result;
3     // 1) check that the ray is not parallel to the triangle:
4     Plane supportPlane(m_c[0],m_normal);
5     if( supportPlane.isParallelTo(ray) ) {
6         result.intersectionExists = false;
7         return result;
8     }
9     // 2) check that the triangle is "in front of" the ray:
10    Vec3 intersectionPoint = supportPlane.getIntersectionPoint(ray);
11    Vec3 originToIntersection = intersectionPoint - ray.origin();
12    float t = Vec3::dot(originToIntersection,ray.direction());
13    if( t < 0.f ) {
14        result.intersectionExists = false;
15        return result;
16    }
17    // 3) check that the intersection point is inside the triangle:
18    // CONVENTION: compute u,v such that p = w0*c0 + w1*c1 + w2*c2, check that
19    // 0 <= w0,w1,w2 <= 1
20
21    float w0,w1,w2;
22    computeBarycentricCoordinates(intersectionPoint,w0,w1,w2);
23    if( w0 < 0.f || w0 > 1.f || w1 < 0.f || w1 > 1.f || w2 < 0.f || w2 > 1.f )
24    {
25        result.intersectionExists = false;
26        return result;
27    }
28
29    // 4) Finally, if all conditions were met, then there is an intersection! :
30    result.intersectionExists = true;
31    result.t = t;
32    result.w0 = w0;
33    result.w1 = w1;
34    result.w2 = w2;
35    result.intersection = intersectionPoint;
36    result.normal = m_normal;
37
38    return result;
```

```
37 }  
38 };
```

Listing 12 – Intersection triangles (Triangle.h)

Une droite parallèle à un plan revient à dire que le vecteur directeur de la droite est orthogonal à la normale de la droite.

Source : <https://math.stackexchange.com/questions/1368461/how-do-i-verify-that-a-line-is-parallel-to-a-plane>

```
1 bool isParallelTo( Line const & L ) const {  
2     bool result;  
3     Vec3 directionVector = L.direction();  
4     result = Vec3::dot( directionVector , m_normal ) == 0.f;  
5     return result;  
6 }
```

Listing 13 – Parallèle à un plan (Plane.h)

Le point d'intersection sur un plan est trouvé à partir de l'équation de la droite et l'équation du plan.

Sources :

- <https://www.kristakingmath.com/blog/intersection-of-a-line-and-a-plane>
- [https://math.libretexts.org/Bookshelves/Calculus/Supplemental\\_Modules\\_\(Calculus\)/Multivariable\\_Calculus/1%3A\\_Vectors\\_in\\_Space/Intersection\\_of\\_a\\_Line\\_and\\_a\\_Plane](https://math.libretexts.org/Bookshelves/Calculus/Supplemental_Modules_(Calculus)/Multivariable_Calculus/1%3A_Vectors_in_Space/Intersection_of_a_Line_and_a_Plane)

```
1 Vec3 getIntersectionPoint( Line const & L ) const {  
2     Vec3 result;  
3     if (isParallelTo(L)) {  
4         throw std::runtime_error("The line is parallel to the plane and does  
5             not intersect.");  
6     }  
7  
7     Vec3 linePoint = L.origin();  
8     Vec3 lineDirection = L.direction();  
9     float t = Vec3::dot(m_center - linePoint , m_normal) / Vec3::dot(  
10        lineDirection , m_normal);  
11     result = linePoint + t * lineDirection;  
12  
13     return result;  
14 }
```

Listing 14 – Intersection plan (Plane.h)

Les coordonnées barycentriques d'un point sont un système de coordonnées pour représenter un point comme la combinaison des sommets du triangle. Ceci est utile dans notre cas afin de déterminer si le point est bien dans notre triangle ou pas. L'implémentation a été faite en adaptant le code donné dans le lien ci-dessus à un plan 3-Dimensionnel.

Source : <https://gamedev.stackexchange.com/questions/23743/whats-the-most-efficient-way-to-compute-barycentric-coordinates>

```
1 void computeBarycentricCoordinates( Vec3 const & p , float & u0 , float & u1 ,  
2                                     float & u2 ) const {  
3     Vec3 v0 = m_c[1] - m_c[0];  
4     Vec3 v1 = m_c[2] - m_c[0];  
5     Vec3 v2 = p - m_c[0];
```

```

5
6     float dot00 = Vec3::dot(v0, v0);
7     float dot01 = Vec3::dot(v0, v1);
8     float dot02 = Vec3::dot(v0, v2);
9     float dot11 = Vec3::dot(v1, v1);
10    float dot12 = Vec3::dot(v1, v2);
11
12    float denom = dot00 * dot11 - dot01 * dot01;
13
14    float invDenom = 1.0f / denom;
15    u1 = (dot11 * dot02 - dot01 * dot12) * invDenom;
16    u2 = (dot00 * dot12 - dot01 * dot02) * invDenom;
17    u0 = 1.0f - u1 - u2;
18}

```

Listing 15 – Coordonnées barycentriques (Triangle.h)

### 3.1.2 Mesh intersection

Le calcul d'intersection de mesh est assez simple, on crée un objet triangle (et non pas un MeshTriangle) pour chaque face du maillage, et on calcule l'intersection de ce triangle.

Actuellement, les rendus de maillages 3D sont très longs, avec 25 samples et 8 rayons d'ombres, un maillage de 968 triangles a pris un peu moins de 20 minutes (voir figure 15).

Source : <https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-polygon-mesh/ray-tracing-polygon-mesh-part-1.html>

```

1 RayTriangleIntersection intersect( Ray const & ray ) const {
2     RayTriangleIntersection closestIntersection;
3     closestIntersection.t = FLT_MAX;
4     for (const auto& triangle : triangles) {
5         Triangle tri(vertices[triangle[0]].position * 1.000001f,
6                     vertices[triangle[1]].position * 1.000001f,
7                     vertices[triangle[2]].position * 1.000001f);
8         RayTriangleIntersection intersection = tri.getIntersection(ray);
9         if (intersection.intersectionExists && intersection.t <
10             closestIntersection.t) {
11             closestIntersection = intersection;
12             closestIntersection.material = material;
13         }
14     }
15     return closestIntersection;
}

```

Listing 16 – Intersect mesh (Mesh.h)

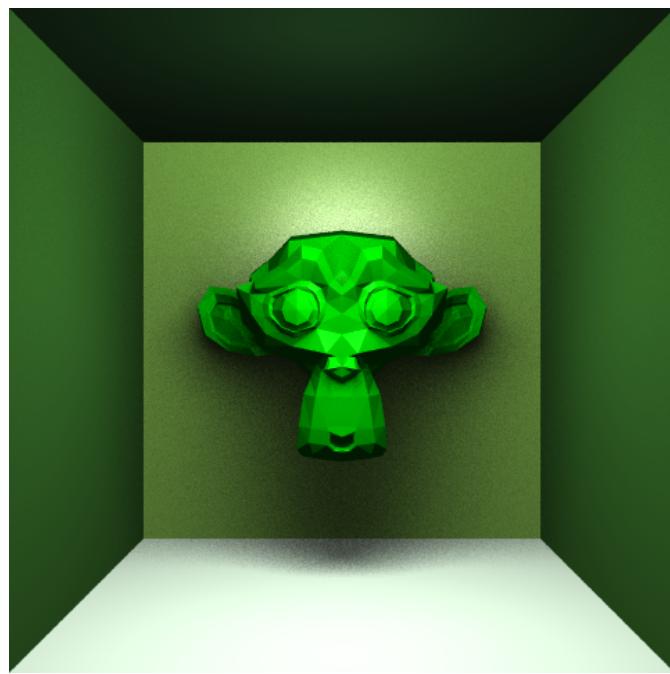


FIGURE 15 – Suzanne Box (25 samples, 8 rayons d'ombre, 968 Triangles , 18 minutes 38 secondes)

### 3.2 Interpolation de sommets (Incomplet)

L'interpolation des valeurs des sommets permet d'obtenir des surfaces plus lisses, que ce soit les couleurs, textures, normales... Dans mon cas, les sommets n'ont pas de textures, et les maillages contiennent une même couleur tout le long. Donc la seule interpolation possible serait celle des sommets. Grâce aux coordonnées barycentriques, on peut interpoler la normale en faisant la somme des coordonnées barycentriques des sommets multipliés par leurs normales. Mes résultats obtenus me semblent incorrects, et je pense que c'est du au fait que les sommets ne contiennent pas de valeur pour leurs normales, et normaliser leurs coordonnées ne suffit pas.

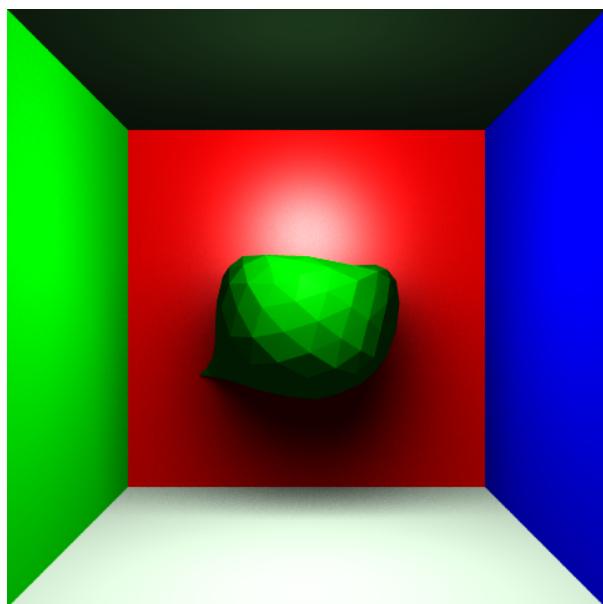


FIGURE 16 – Blob sans interpolation de normales

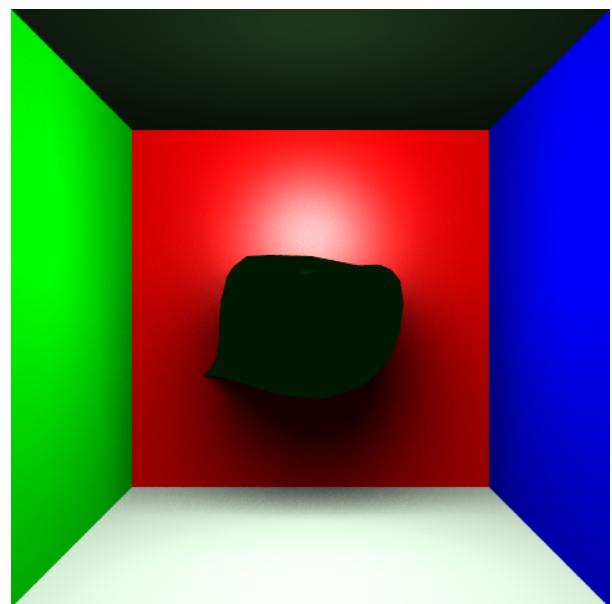


FIGURE 17 – Blob avec interpolation de normales

Source : [http://web.archive.org/web/20090609111431/http://www.crackthecode.us/barycentric/barycentric\\_coordinates.html](http://web.archive.org/web/20090609111431/http://www.crackthecode.us/barycentric/barycentric_coordinates.html)

```
1 RayTriangleIntersection getIntersection( Ray const & ray ) const {
2     ...
3     //INTERPOLATION
4     Vec3 n0 = m_c[0];
5     Vec3 n1 = m_c[1];
6     Vec3 n2 = m_c[2];
7     n1.normalize();n2.normalize();n0.normalize();
8
9     Vec3 interpolatedNormal = w0*n0 + w1*n2 + w2*n3 ;
10
11    interpolatedNormal.normalize();
12    //result.normal = m_normal; //without interpolation
13    result.normal = interpolatedNormal; //with interpolation
14    ...
15    return result;
16 }
```

Listing 17 – Interpolation des sommets (Triangle.h)

### 3.3 Réflexion

Le code pour la réflexion est exactement celui présent dans Ray Tracing in One Weekend de Peter Shirley

Source : Ray tracing in One Weekend - Peter Shirley

```
1 Vec3 rayTraceRecursive(Ray ray, int NRemainingBounces) {
2     ...
3     if(mat.type == Material_Mirror){
4         Vec3 reflectedDir = reflect(ray.direction(), N);
5         Ray reflectedRay(intersectionPoint + N * 0.001f, reflectedDir);
6         color += rayTraceRecursive(reflectedRay, NRemainingBounces - 1);
7         return color;
8     }
9     ...
10 }
```

Listing 18 – Application de réflexion dans RayTraceRecursive

```
1 Vec3 reflect(const Vec3 & v, const Vec3 & N) {
2     return v - 2 * Vec3::dot(v, N) * N;
3 }
```

Listing 19 – Méthode reflect

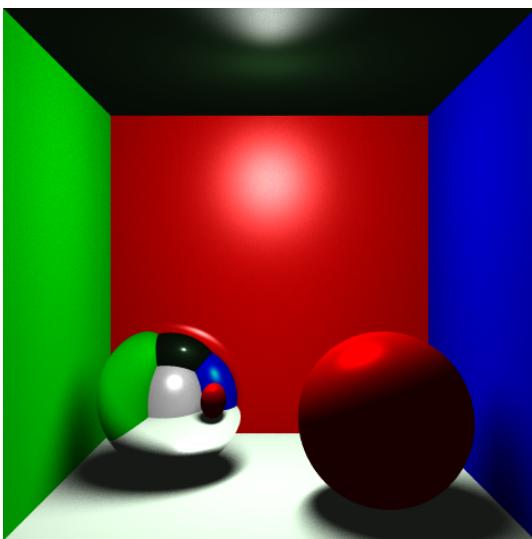


FIGURE 18 – Cornell box avec sphère en miroir

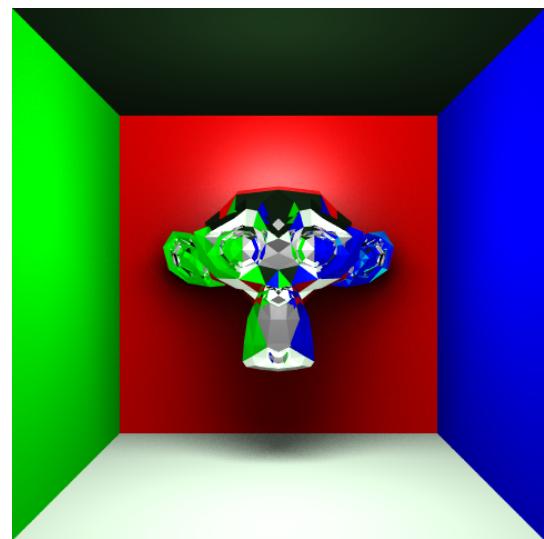


FIGURE 19 – Box avec Suzanne en miroir

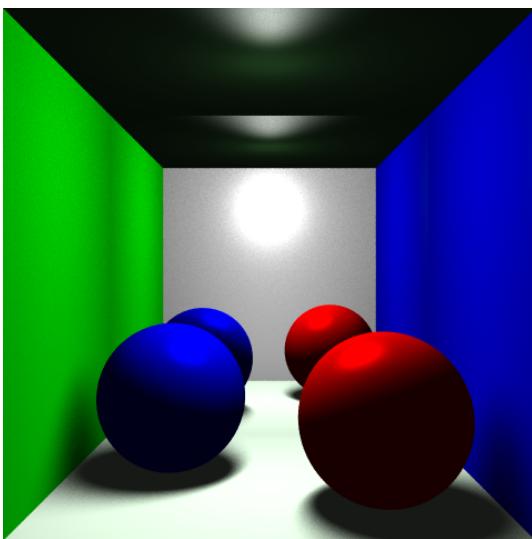


FIGURE 20 – Cornell box avec un mur en miroir

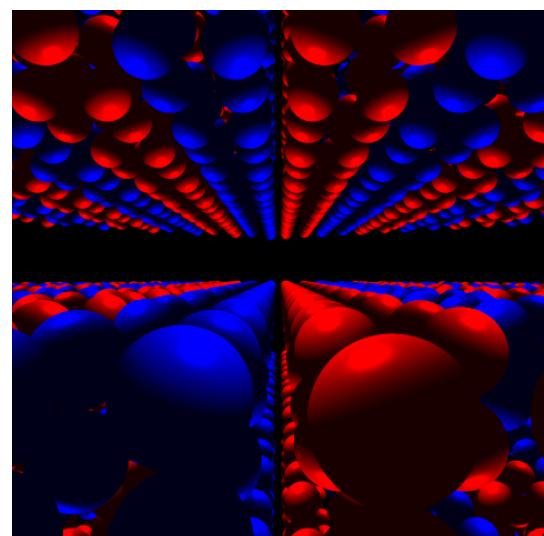


FIGURE 21 – Un peu trop de miroirs....

## 4 Quatrième Phase

### 4.1 Réfraction

L'objectif est de créer une sphère en verre, pour accomplir cette tâche il faut donc implémenter la réfraction. En utilisant la loi de Snell appliquée aux vecteurs, on obtient le vecteur directeur du rayon réfracté. On utilise l'index medium du verre (entre 1.4 et 1.9) et celui de l'air (ici arrondi à 1) ainsi que la normale pour le calculer. Mais, l'application directe de la formule ne suffit pas pour s'assurer du bon fonctionnement de la réfraction. Il faut vérifier si le rayon est entrant ou sortant de la sphère (ou de n'importe quel objet en verre), on fait ça en vérifiant le signe du produit scalaire de la direction et de la normale de la surface d'intersection, et on inverse ou pas les index

medium et normales dépendent de ceci.

Source : <https://physics.stackexchange.com/questions/435512/snells-law-in-vector-form>

```
1 Vec3 rayTraceRecursive(Ray ray, int NRemainingBounces) {
2     ...
3     if (mat.type == Material_Glass) {
4         mat.index_medium = 1.5f; // Glass
5
6         Vec3 n_inter;
7         float N1;
8         float N2;
9         //check if in or out of sphere
10        if(Vec3::dot(ray.direction(), N) < 0){
11            n_inter = N;
12            N1 = 1.0f;
13            N2 = mat.index_medium;
14        } else {
15            n_inter = -1*N;
16            N1 = mat.index_medium;
17            N2 = 1.0f;
18        }
19        //Vec3 n_inter = N;
20
21        Vec3 refractedDir = refract(ray.direction(), n_inter, N1, N2);
22        Ray refractedRay(intersectionPoint - n_inter * 0.001f, refractedDir
23                           );
24        color = rayTraceRecursive(refractedRay, NRemainingBounces - 1);
25        return color;
26    }
27}
```

Listing 20 – Application de réfraction dans RayTraceRecursive (Scene.h)

```
1 Vec3 refract(const Vec3 & i, const Vec3 & N, float N1, float N2) {
2     Vec3 n = N;
3     float ni = Vec3::dot(i, n);
4     float ni2 = ni * ni;
5     float n1 = N1;
6     float n2 = N2;
7     if(ni < 0){
8         ni = -ni;
9     }
10    else{
11        n = -1*n; n1 = N2; n2 = N1;
12    }
13    float mu = n1 / n2;
14    float mu2 = mu * mu;
15    Vec3 t = (mu*ni - sqrt(1 - mu2 * (1 - ni2))) * n + mu * (i);
16    //Vec3 t = (sqrt(1 - mu2 * (1 - ni2))) * n + mu * (i - ni * n); formule de
17    //base
18    return t;
}
```

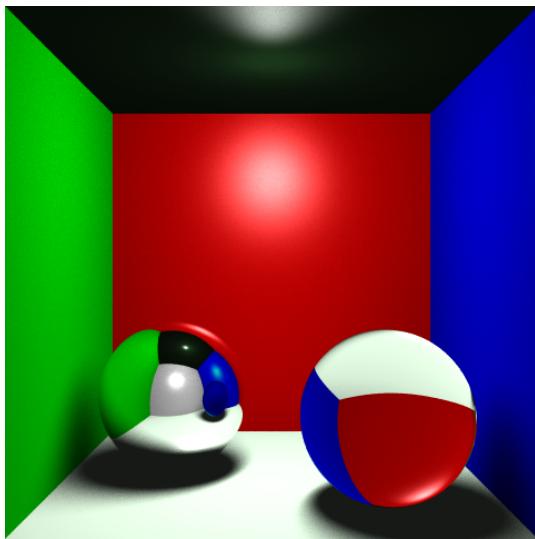


FIGURE 22 – Sphere en verre (index 1.5)

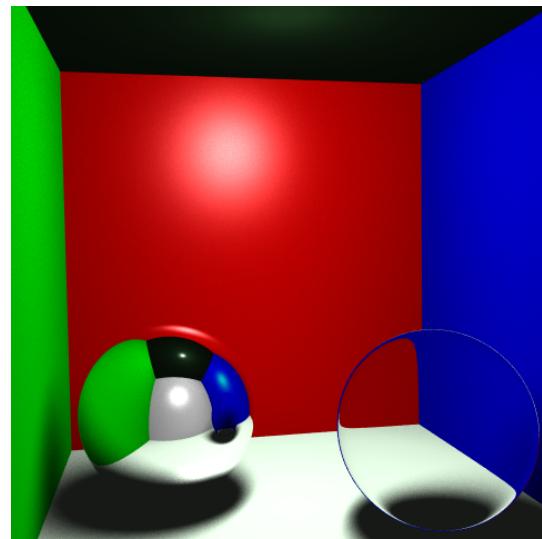


FIGURE 23 – Sphere en verre (index 1.05)



FIGURE 24 – Suzanne en verre (index 1.05)

## 4.2 Structure d'accélération : Kd-Tree

### 4.2.1 Bounding Box

La première étape pour définir un kd tree est de définir les boîtes englobantes (ou "Bounding Box"). C'est une boîte qui doit englober tous les triangles qu'on lui donne, donc agrandir à chaque fois que l'on ajoute un triangle qui n'est pas dans la boîte (Une logique de construction sera mise en place avec le Kd-Tree).

```
1 class BoundingBox{
2     public :
3         Vec3 min;
4         Vec3 max;
5 }
```

```

6     BoundingBox() : min(Vec3(FLT_MAX, FLT_MAX, FLT_MAX)), max(Vec3(-FLT_MAX
7         , -FLT_MAX, -FLT_MAX)) {}
8
9     void expand(const Vec3 &point){
10        min[0] = std::min(min[0], point[0]);
11        min[1] = std::min(min[1], point[1]);
12        min[2] = std::min(min[2], point[2]);
13        max[0] = std::max(max[0], point[0]);
14        max[1] = std::max(max[1], point[1]);
15        max[2] = std::max(max[2], point[2]);
16    }
17
18    void expand(const BoundingBox& box) {
19        expand(box.min);
20        expand(box.max);
21    }
22
23    Vec3 size() const {
24        return max - min;
25    }

```

Listing 21 – Classe BoundingBox (BoundingBox.h)

#### 4.2.2 Kd-Tree

Après avoir créé une BoundingBox, pour implémenter un Kd-Tree on a trois étapes à faire, la construction de l'arbre, la traversée, et l'intersection dans l'arbre.

Pour construire son arbre, on commence par agrandir la bounding box de la branche en y ajoutant tous les triangles présents. Ensuite, vu que l'arbre est créé de façon récursive, il faut gérer son cas d'arrêt. Ici, on s'arrête quand la branche sur laquelle on se trouve est une feuille, c'est à dire qu'elle n'a pas d'enfants, qu'on définit comme étant une branche avec un nombre de triangles inférieur à la valeur choisie. Afin de diviser les boîtes plus équitablement, j'ai décidé de faire avec le centre du triangle médian le long d'un axe qui alterne entre chacune des trois coordonnées. L'étape suivante est d'ajouter les triangles à leurs côtés de l'axe respectif en regardant si leurs centres sont à "gauche" ou à "droite" de la séparation. Puis les arbres gauche et droite sont construits récursivement avec ces triangles.

```

1 void KdTreeNode::build(const std::vector<Triangle> listOfTriangles , int depth)
2 {
3     //scene bounding box (big box)
4     for(const Triangle &t : listOfTriangles){
5         boundingBox.expand(t.get_mc(0));
6         boundingBox.expand(t.get_mc(1));
7         boundingBox.expand(t.get_mc(2));
8     }
9     //stopping condition (number of triangles)
10    if(listOfTriangles.size() <= maxTriangles || boundingBox.size().length() <
11        minSize){
12        isLeaf = true;
13        triangles = listOfTriangles;
14        return;
15    }

```

```

14 dimensionSplit = depth % 3;
15 //split par mediane
16 std::vector<Triangle> sortedTriangles = listOfTriangles;
17 std::sort(sortedTriangles.begin(), sortedTriangles.end(), [this](const
18     Triangle &a, const Triangle &b) {
19         return compareTriangles(a, b, dimensionSplit);
20     });
21 size_t medianIndex = sortedTriangles.size() / 2;
22 splitDistance = sortedTriangles[medianIndex].centroid()[dimensionSplit];
23 //add triangles to l or r
24 std::vector<Triangle> leftTriangles;
25 std::vector<Triangle> rightTriangles;
26 for (const Triangle& t : listOfTriangles) {
27     bool left = false, right = false;
28     if (t.centroid()[dimensionSplit] <= splitDistance) left = true;
29     else right = true;
30     if (left) leftTriangles.push_back(t);
31     if (right && !left) rightTriangles.push_back(t); //stop dupes
32 }
33 if (leftTriangles.size() == listOfTriangles.size() || rightTriangles.size()
34 == listOfTriangles.size() || leftTriangles.size() == 0 ||
35 rightTriangles.size() == 0) {
36     isLeaf = true;
37     triangles = listOfTriangles;
38     return;
39 }
40 //build left and right nodes
41 left = new KdTreeNode();
42 right = new KdTreeNode();
43 //update boxes fire emoji
44 left->boundingBox = boundingBox;
45 right->boundingBox = boundingBox;
46 left->boundingBox.max[dimensionSplit] = splitDistance;
47 right->boundingBox.min[dimensionSplit] = splitDistance;
48 left->build(leftTriangles, depth + 1);
49 right->build(rightTriangles, depth + 1);
50 isBuilt = true;
51 }
```

Listing 22 – Construction de l’arbre (KdTree.cpp)

Quand on traverse l’arbre, il faut s’arrêter que quand on est sur une feuille, sinon on regarde si l’on intersecte avec les deux bounding box gauche ou droite, et on traverse récursivement les boîtes où l’on intersecte.

```

1 RayTriangleIntersection KdTreeNode::traverse(const Ray& ray) const {
2     Vec3 rayDir = ray.direction();
3     Vec3 origin = ray.origin();
4     //stop condition
5     if (isLeaf) {
6         return intersect(triangles, ray);
7     }
8     bool leftHit = intersectAABB(left->boundingBox, origin, rayDir);
9     bool rightHit = intersectAABB(right->boundingBox, origin, rayDir);
10    if (leftHit && rightHit) {
11        RayTriangleIntersection leftHitResult = left->traverse(ray);
12        RayTriangleIntersection rightHitResult = right->traverse(ray);
13    }
14 }
```

```

13     if (leftHitResult.intersectionExists && leftHitResult.t <
14         rightHitResult.t) {
15         return leftHitResult;
16     }
17     return rightHitResult;
18 } else if (leftHit) {
19     return left->traverse(ray);
20 } else if (rightHit) {
21     return right->traverse(ray);
22 }
23 RayTriangleIntersection noHit;
24 noHit.intersectionExists = false;
25 return noHit;
}

```

Listing 23 – Traverse du Kd-Tree (KdTree.h)

On veut vérifier si un rayon intersecte une boîte englobante alignée sur les axes AABB. On calcule les positions des coins de l'AABB par rapport à l'origine du rayon et utilise l'inverse des composantes de la direction pour déterminer les temps d'intersection sur chaque axe. Une intersection est confirmée si le temps max est inférieur au temps min.

Source : [https://tavianator.com/2011/ray\\_box.html](https://tavianator.com/2011/ray_box.html) (Merci à Tom Zinck pour la source et son aide pour cette partie.)

```

1 bool intersectAABB(const BoundingBox& boundingBox , const Vec3& rayOrigin ,
2     const Vec3& rayDir) {
3     Vec3 aa1 = boundingBox.min - rayOrigin;
4     Vec3 aa2 = boundingBox.max - rayOrigin;
5     Vec3 facs(1.0 / rayDir[0], 1.0 / rayDir[1], 1.0 / rayDir[2]);
6
7     float tx1 = aa1[0] * facs[0];
8     float tx2 = aa2[0] * facs[0];
9     float tmin = std::min(tx1, tx2);
10    float tmax = std::max(tx1, tx2);
11
12    float ty1 = aa1[1] * facs[1];
13    float ty2 = aa2[1] * facs[1];
14    tmin = std::max(tmin, std::min(ty1, ty2));
15    tmax = std::min(tmax, std::max(ty1, ty2));
16
17    float tz1 = aa1[2] * facs[2];
18    float tz2 = aa2[2] * facs[2];
19    tmin = std::max(tmin, std::min(tz1, tz2));
20    tmax = std::min(tmax, std::max(tz1, tz2));
21
22    return tmax >= tmin;
}

```

Listing 24 – Détection d'intersection avec la boîte

L'intersection consiste simplement à itérer sur chaque triangle de la feuille, calculer son intersection (avec la méthode interséction du triangle détaillée plus haut) et prendre l'intersection la plus proche.

```

1 RayTriangleIntersection KdTreeNode::intersect(const std::vector<Triangle>&
2     listOfTriangles , const Ray& ray) const {

```

```

2     RayTriangleIntersection result;
3     result.intersectionExists = false;
4     result.t = FLT_MAX;
5     for(const Triangle &t : listOfTriangles){
6         RayTriangleIntersection hit = t.getIntersection(ray);
7         if(hit.intersectionExists && hit.t < result.t){
8             result = hit;
9         }
10    }
11 }

```

Listing 25 – Intersection triangle Kd-tree (KdTree.h)

#### 4.2.3 Results

La structure d'accélération n'a que été appliquée aux maillages car les calculs pour les sphères semblaient déjà être efficaces. On peut voir que cette structure accélère beaucoup les calculs (18 minutes 38 secondes sans Kd-Tree contre 31 secondes avec l'arbre) ce qui permet de tracer des formes plus complexes.

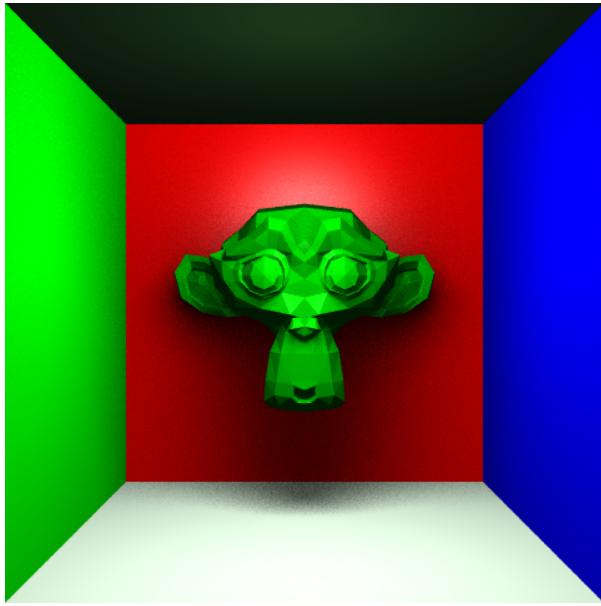


FIGURE 25 – Suzanne Box Kd (25 samples , 8 rayons d'ombre , 968 triangles, 31 secondes)

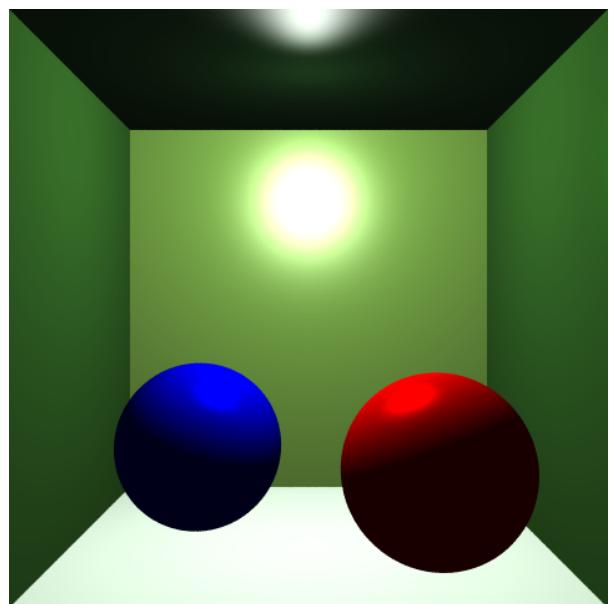


FIGURE 26 – Cornell box avec shading de Phong (25 samples , 8 rayons d'ombre , 968 triangles, 31 secondes)

#### Sources :

- Cours : Lancer de rayon et Structures d'accélération
- <https://www.youtube.com/watch?v=TrqK-atFfWY&t=2541s>

## 5 Phase Finale

### 5.1 Motion Blur

Le flou cinétique (ou "Motion Blur") est l'effet obtenu lorsque l'on a un mouvement pendant que l'obturateur de la caméra est ouvert. L'application de cet effet à notre ray tracer a deux soucis,

la gestion de temps et le mouvement.

La première étape logique serait d'ajouter le concept de temps au raytracer. Pour ce faire, on va simplement ajouter un attribut temps à chaque rayon afin de stocker leur temps exact.

```
1 class Ray : public Line {
2     private:
3         double tm;
4     public:
5         Ray() : Line() {}
6         Ray( Vec3 const & o , Vec3 const & d ) : Line(o,d) {}
7         Ray(Vec3 const & o , Vec3 const & d , double tm) : Line(o,d) , tm(
8             tm) {}
9
10        double time() const { return tm; }
11    };
```

Listing 26 – temps ajouté au rayon (Ray.h)

L'étape qui suit serait d'introduire le passage du temps pour imiter l'ouverture de l'obturateur. Mais la gestion des vrais paramètres de temps est plus complexe, avec le nombre de frames par secondes... Pour simplifier cette tâche, on va uniquement générer un temps aléatoire entre 0 et 1 pour chaque rayon, on calcule donc cela sur une seule frame.

```
1 void ray_trace_from_camera() {
2     ...
3     for (int y=0; y<h; y++) {
4         for (int x=0; x<w; x++) {
5             ...
6                 float time = (float)(rand()) / float(RAND_MAX); //normally
7                     between 0 and 1
8                 Ray ray(pos,dir,time);
9                 Vec3 color = scenes[selected_scene].rayTrace(ray); //blur
10                ...
11            }
12        ...
13    }
```

Listing 27 – Ajout de temps (main.cpp)

La dernière étape avant l'application du floutage, est de créer le mouvement de la sphère. On ajoute un attribut centre début, et un attribut centre de fin pour les sphères en mouvement. On utilisera ces points pour faire une interpolation linéaire d'un point à un autre pour obtenir les bons points d'intersection créant un effet de mouvement dans le temps.

```
1 RaySphereIntersection intersect2(const Ray &ray) const {
2     ...
3     //get current time and position
4     float t = ray.time();
5     Vec3 current_center = start_center + t * (end_center-start_center);
6     ...
7 }
```

Listing 28 – Intersection en mouvement (Sphere.h)

```

1 Vec3 m_center;
2 float m_radius;
3 Vec3 start_center;
4 Vec3 end_center;
5 bool inMotion = false;
6
7 Sphere() : Mesh() {}
8 Sphere(Vec3 c , float r) : Mesh() , m_center(c) , m_radius(r) {}
9 Sphere(Vec3 c1 , Vec3 c2 , float r) : Mesh() , m_center(c1), start_center(c1) ,
10   end_center(c2) , m_radius(r) {
11     inMotion = true;
12 }
```

Listing 29 – Paramètres et constructeur Sphère (Sphere.h)

Source : *Ray Tracing : The Next Week - Peter Shirley*

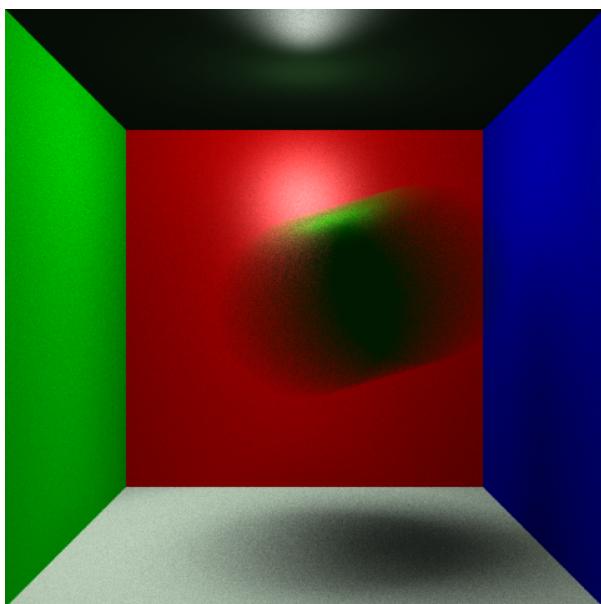


FIGURE 27 – Exemple de sphère avec flou cinétique

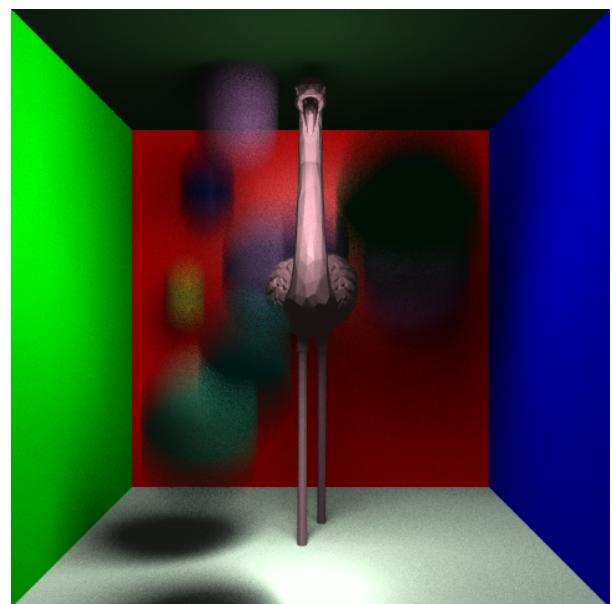


FIGURE 28 – "Meteor Shower" (Flamant rose avec 10 sphères en mouvement)

## 5.2 Rendus en plus

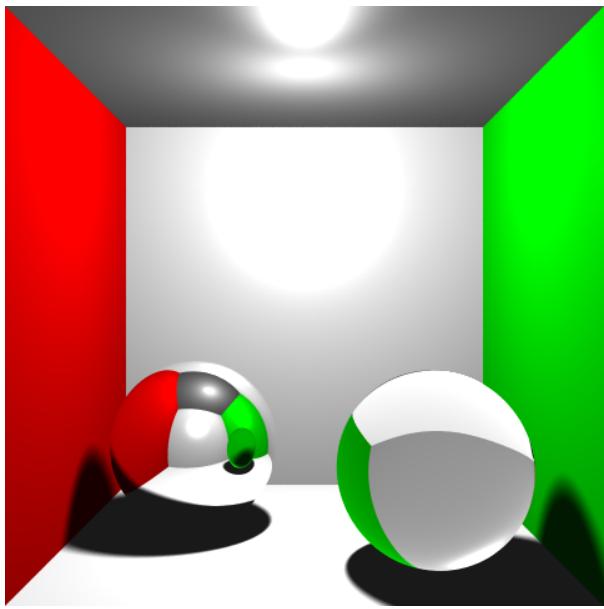


FIGURE 29 – Cornell box (100 samples, 32 rayons d'ombre)

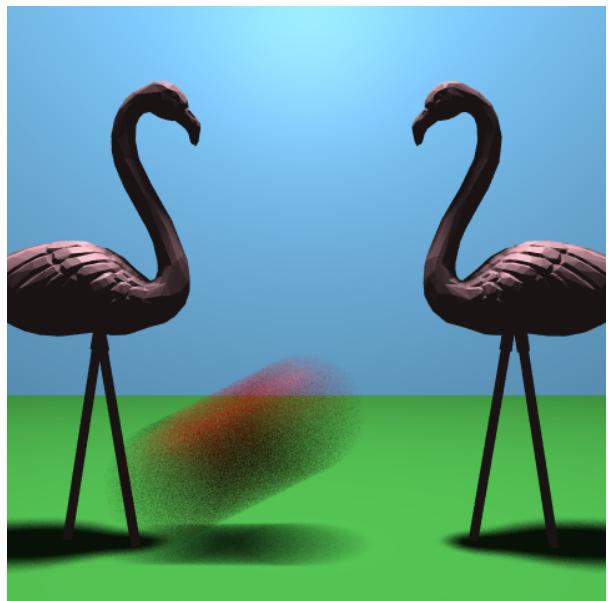


FIGURE 30 – "Petit match de foot"