Programmation 3D

Raytracer phases 1 et 2

Louis Jean
Master 1 IMAGINE
Université de Montpellier

6 novembre 2023

Table des matières

1 Introduction

Dans le vaste univers de la visualisation graphique, le lancer de rayons est une technique incontournable et très en vogue pour générer des images de haute qualité avec un réalisme saisissant. Ce projet vise à concevoir et à mettre en œuvre une application graphique exploitant le potentiel du lancer de rayons. Grâce à une démarche structurée en deux phases distinctes et une base de code à compléter fournie, j'ai établi les fondations nécessaires pour explorer cette technique fascinante, que je peaufinerai par la suite, jusqu'au rendu final.

2 Phase 1

La phase 1 du projet de lancer de rayons constitue la pierre angulaire de l'application graphique. Elle est principalement axée sur la conceptualisation et l'implémentation des mécanismes fondamentaux permettant d'évaluer l'intersection des rayons lumineux avec des formes géométriques élémentaires. Plus précisément, je me suis penché sur deux formes de base : la sphère et le carré. Cette étape est élémentaire.

2.1 Intersection rayon-sphère

Premièrement, j'ai codé le calcul de l'intersection d'un rayon avec une sphère dans **Sphere.h**, en me basant sur le cours. Cette fonction **intersect** renvoie une structure contenant les informations de l'intersection avec la sphère.

```
RaySphereIntersection intersect(const Ray &ray) const {

    RaySphereIntersection intersection;
    float a = Vec3::dot(ray.direction(),ray.direction());
    float b = 2 * Vec3::dot(ray.direction(),(ray.origin()
       ↪  - m_center));
    float c = pow((ray.origin() - m_center).norm(),2) -
       ↪ pow(m_radius,2);
    float delta = pow(b,2) - (4 * a * c);

    if(delta < 0) {
        intersection.intersectionExists = false;
    }

    else {
        intersection.intersectionExists = true;
        float t1 = (-b - sqrt(delta)) / (2*a);
        float t2 = (-b + sqrt(delta)) / (2*a);
        float tmin = std::min(t1,t2);
        float tmax = std::max(t1,t2);
        intersection.t = tmin;
        intersection.intersection = ray.origin() + (tmin
           ↪ * ray.direction());
        intersection.secondintersection = ray.origin() +
           ↪ (tmax * ray.direction());
        Vec3 normal = (intersection.intersection -
           ↪ m_center);
        normal.normalize();
        intersection.normal = normal;
    }
    return intersection;
}
```

N.B. **:** J'ai créé une énumération **MeshType** pour mieux m'y retrouver dans les différents types d'objets intersectés.

```
enum MeshType {
    MeshType_Sphere,
    MeshType_Square
};
```

Ensuite, j'ai pu commencer à écrire la fonction **computeIntersection** dans **Scene.h**. Cette fonction s'intéresse au traitement des intersections calculées plus tôt. Elle sélectionne l'intersection la plus proche de la caméra, et la retourne.

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

    RaySceneIntersection result;
    result.intersectionExists = false;
    float closestIntersection = FLT_MAX;
    int nbSpheres = spheres.size();

    // Traitement des sphères
    for(int i = 0; i < nbSpheres; i++) {

        const Sphere sphere = spheres[i];

        RaySphereIntersection sphereIntersection = sphere.intersect(ray);

        if(sphereIntersection.intersectionExists && sphereIntersection.t < closestIntersection) {
            closestIntersection = sphereIntersection.t;
            result.intersectionExists = true;
            result.t = sphereIntersection.t;
            result.objectIndex = i;
            result.typeOfIntersectedObject = MeshType_Sphere;
            result.raySphereIntersection = sphereIntersection;
        }
    }

    return intersection;
}
```

En utilisant **computeIntersection**, j'ai complété **rayTraceRecursive**, toujours dans **Scene.h**, qui s'occupe de calculer la couleur du pixel correspondant au rayon lancé et la direction du rayon induit par la réflexion de son rayon parent. Cette fonction récursive est vouée à être appelée sur chaque rayon lancé et réfléchi.

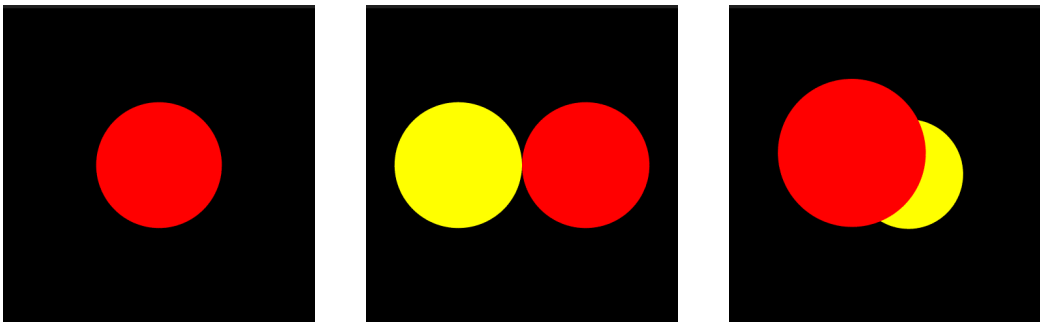
```
Vec3 rayTraceRecursive(Ray ray, int NRemainingBounces) {

    RaySceneIntersection rayScenIntersection = computeIntersection(ray);
    Vec3 intersectionPoint(0.0f,0.0f,0.0f);
    Vec3 normalToIntersectionPoint(0.0f,0.0f,0.0f);
    Vec3 reflectedDirection(0.0f,0.0f,0.0f);
    Vec3 color(0.0f,0.0f,0.0f); // Fond noir

    if(!rayScenIntersection.intersectionExists) {
        return color;
    }

    switch(rayScenIntersection.typeOfIntersectedObject) {
        case MeshType_Sphere:
            intersectionPoint = rayScenIntersection.raySphereIntersection.intersection;
            normalToIntersectionPoint = rayScenIntersection.raySphereIntersection.normal;
            color = spheres[rayScenIntersection.objectIndex].material.diffuse_material;
        break;

         default:
         break;
    }

    // Calcul de la direction du rayon réfléchi
    reflectedDirection = ray.direction() - (2 * Vec3::dot(ray.direction(), normalToIntersectionPoint) * normalToIntersectionPoint);

    if(NRemainingBounces > 0) {
        rayTraceRecursive(Ray(intersectionPoint, reflectedDirection),NRemainingBounces - 1);
    }

    return color;
}
```

Enfin, la fonction **rayTrace**, encore dans **Scene.h**, qui permet d'initier le lancer de rayons, en définissant le rayon de départ, et le nombre de rebonds pour chacun des rayons.

N.B. : Chacun des rendus inclus dans ce rapport a été réalisé avec 3 rebonds par rayon parent.

```
1 Vec3 rayTrace( Ray const & rayStart ) {
2     return rayTraceRecursive(rayStart,2);
3 }
```

À partir de là, j'ai pu observer mes premiers rendus.



(a) Sphère simple (b) Deux sphères (c) Deux sphères après déplacement de la caméra

Figure 1: Rendus de sphères grâce au lancer de rayons

2.2 Intersection rayon-plan

Ensuite, je me suis intéressé aux intersections rayon-plan, afin de pouvoir gérer les intersections rayon-carré ultérieurement.
Pour cela, j'ai écrit les fonctions (dans **Plane.h**) **isParallelTo**, qui vérifie si le rayon est parallèle au plan en question, et **getIntersectionPoint** qui calcule le point d'intersection (s'il existe) entre le rayon et le plan considéré.

```
1 bool isParallelTo(Line const & L) const {
2     return
3     fabs(Vec3::dot(L.direction(),m_normal)) < 1e-6;
4     // Pas 0 car imprécision sur les flottants en machine
5 }
```

```
1     Vec3 getIntersectionPoint(Line const & L) const {
2
3     Vec3 result = Vec3(-100.0f,-100.0f,-100.0f);
4
5     if(!isParallelTo(L)) {
6         float D = Vec3::dot(m_center,m_normal);
7         float t = (D - Vec3::dot(L.origin(),m_normal)) /
              ↪ Vec3::dot(L.direction(),m_normal);
8         if(t > 0) {
9             result = L.origin() + t * L.direction();
10        }
11    }
12    return result;
13 }
```

N.B. : Dans le cas où le rayon est parallèle au plan, j'ai fait le choix de renvoyer un vecteur arbitraire, le vecteur $(-100, -100, -100)$.

2.3 Intersection rayon-carré

Après avoir codé les intersections rayon-plan, il était plus facile de calculer les intersections rayon-carré. Le principe est le suivant : on calcule le plan du carré, on regarde si le rayon n'est pas parallèle, puis on projette le point d'intersection du rayon avec le plan du carré sur les vecteurs des côtés du carré. Si les deux scalaires obtenus sont compris entre 0 et 1, le rayon intersecte le carré. Pour mettre en œuvre cette méthode, j'ai écrit une fonction **intersect**, placée dans **Square.h**.

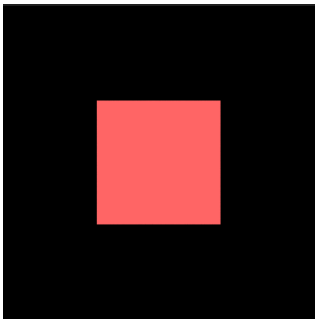
```
1  RaySquareIntersection intersect(const Ray &ray) const {
2      RaySquareIntersection intersection;
3      intersection.intersectionExists = false;
4      // Prise en compte des potentielles transformations
5      Vec3 bottomLeft = vertices[0].position;
6      Vec3 bottomRight = vertices[1].position;
7      // Un peu bourrin mais je n'ai pas trouvé mieux
8      Vec3 rightVector = bottomRight - bottomLeft;
9      Vec3 upVector = vertices[3].position - bottomLeft;
10     Vec3 normal = vertices[0].normal;
11     Plane squarePlane(bottomLeft,normal);
12     Vec3 squarePlaneIntersection = squarePlane.
           ↪ getIntersectionPoint(ray);
13     if(!squarePlane.isParallelTo(ray)) {
14         Vec3 localIntersectionVector =
               ↪ squarePlaneIntersection - bottomLeft;
15         float u_intersection = Vec3::dot(
               ↪ localIntersectionVector,rightVector) /
               ↪ rightVector.squareLength();
16         float v_intersection = Vec3::dot(
               ↪ localIntersectionVector,upVector) /
               ↪ upVector.squareLength();
17         if(u_intersection >= 0.0f && u_intersection <=
               ↪ 1.0f && v_intersection >= 0.0f &&
               ↪ v_intersection <= 1.0f) {
18             intersection.intersectionExists = true;
19             intersection.t = (squarePlaneIntersection -
                   ↪ ray.origin()).length();
20             intersection.u = u_intersection;
21             intersection.v = v_intersection;
22             intersection.intersection =
                   ↪ squarePlaneIntersection;
23             intersection.normal = normal;
24         }
25     }
26     return intersection;
27 }
```

Une fois ceci fait, j'ai complété les fonctions **computeIntersection** et **rayTracingRecursive**, afin de prendre en compte les intersections rayon-carré. Voici les bouts de code que j'ai respectivement ajoutés dans ces fonctions.

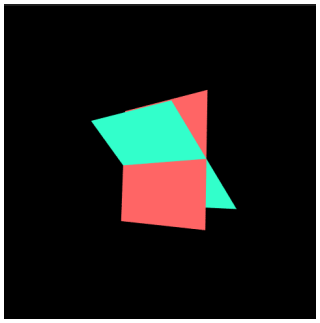
```
// Traitement des carrés
int nbSquares = squares.size();
for(int j = 0; j < nbSquares; j++) {
        const Square square = squares[j];
        RaySquareIntersection squareIntersection = square
            ↪ .intersect(ray);
        if(squareIntersection.intersectionExists &&
            ↪ squareIntersection.t < closestIntersection)
            ↪  {
            closestIntersection = squareIntersection.t;
            result.intersectionExists = true;
            result.t = squareIntersection.t;
            result.objectIndex = j;
            result.typeOfIntersectedObject =
                ↪ MeshType_Square;
            result.raySquareIntersection =
                ↪ squareIntersection;
        }
}
```

```
case MeshType_Square:
    intersectionPoint = raySceneIntersection.
        ↪ raySquareIntersection.intersection;
    normalToIntersectionPoint = raySceneIntersection.
        ↪ raySquareIntersection.normal;
    color = squares[raySceneIntersection.objectIndex].
        ↪ material.diffuse_material;
break;
```

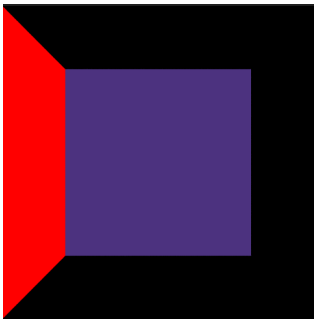
Passée cette étape, j'ai pu effectuer des rendus sur des carrés.



(a) Carré simple



(b) Croisement de deux carrés



(c) Deux carrés formant un angle droit

Figure 2: Rendus de carrés grâce au lancer de rayons

2.4 Boîte de Cornell

Pour la fin de la phase 1, le sujet proposait d'afficher une boîte de Cornell, afin de combiner les intersections rayon-sphère et rayon-carré, pour apprécier le travail accompli.

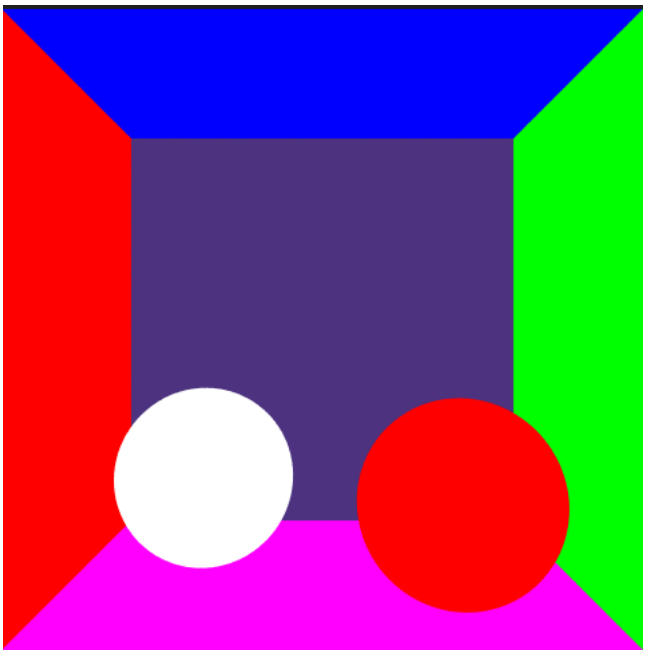


Figure 3: Rendu d'une boîte de Cornell

2.5 Complément

Il est important de noter que dans notre application, nous pouvons contrôler le nombre de rayons lancés par pixel, grâce à la fonction (déjà présente dans la base de code) **rayTraceFromCamera**, présente dans **main.cpp**.

```
1  void ray_trace_from_camera() {
2      int w = glutGet(GLUT_WINDOW_WIDTH), h = glutGet(
           ↪ GLUT_WINDOW_HEIGHT);
3      camera.apply();
4      Vec3 pos , dir;
5      unsigned int nsamples = 50;
6      std::vector<Vec3> image(w*h , Vec3(0,0,0));
7      for (int y=0; y<h; y++) {
8          for (int x=0; x<w; x++) {
9              for( unsigned int s = 0 ; s < nsamples ; ++s
                   ↪ ) {
10                 float u = ((float)(x) + (float)(rand())/(
                       ↪ float)(RAND_MAX)) / w;
11                 float v = ((float)(y) + (float)(rand())/(
                       ↪ float)(RAND_MAX)) / h;
12                 screen_space_to_world_space_ray(u,v,pos,
                       ↪ dir);
13                 Vec3 color = scenes[selected_scene].
                       ↪ rayTrace( Ray(pos , dir) );
14                 image[x + y*w] += color;
15             }
16             image[x + y*w] /= nsamples;
17         }
18     }
19     ...
20 }
```

En effet, la variable nsamples permet de contrôler cette valeur. Plus cette valeur est élevée, moins il y aura d'aliasing sur l'image finale, mais plus le rendu sera long à effectuer.

3 Phase 2

La phase 2 de ce projet de lancer de rayon introduit des éléments clés pour donner une apparence 3D aux objets. Elle s'intéresse au modèle de Phong pour calculer l'illumination des points d'intersection avec les objets. De plus, nous allons ajouter des ombres réalistes, en veillant à ce que les objets dans l'ombre ne soient pas éclairés. Cette phase aborde également la gestion des ombres douces. En résumé, elle permet d'approfondir et d'améliorer le rendu 3D obtenu par lancer de rayons.

3.1 Modèle de Phong

Le modèle de Phong se découpe en trois composantes : la réflexion ambiante, la réflexion diffuse et la réflexion spéculaire.

$$I_{\mathrm{a}} = I_{\mathrm{sa}} * K_{\mathrm{a}}$$
$$I_{\mathrm{d}} = I_{\mathrm{sd}} * K_{\mathrm{d}} * (\mathbf{N} \cdot \mathbf{L})$$
$$I_{\mathrm{s}} = I_{\mathrm{ss}} * K_{\mathrm{s}} * (\mathbf{R} \cdot \mathbf{V})^{n}$$

Combinées, ces trois données permettent une évaluation réaliste de notre scène 3D. Pour l'implémenter, il m'a d'abord fallu mettre à jour la structure **Light** présente dans **Scene.h** afin de rajouter les trois coefficients de lumière ambiante, diffuse et spéculaire.

```
1     float ambientPower;
2     float diffusePower;
3     float specularPower;
```

Ici, I_{sa} = ambientPower, I_{sd} = diffusePower et I_{ss} = specularPower.

Par la suite, j'ai pu mettre à jour la fonction **rayTraceRecursive** grâce aux équations de Phong, en ajoutant les lignes de code ci-dessous pour chacun des types d'objets dans le switch (ici, j'ai pris la sphère en exemple). N est la normale au point d'intersection, L est le vecteur qui part du point d'intersection et qui va en direction de la lumière, V est le vecteur de vue qui part du point d'intersection et qui va vers la caméra, R est la direction de la réflexion spéculaire.

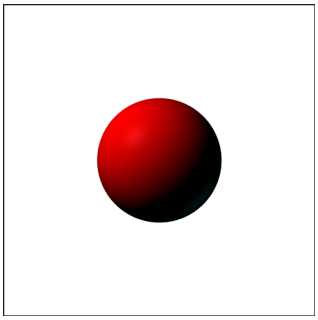
```
1  N = normalToIntersectionPoint;
2  L = lights[0].pos - intersectionPoint;
3  R = (2 * Vec3::dot(L,N) * N) - L;
4  V = ray.origin() - intersectionPoint;
5  N.normalize();
6  L.normalize();
7  R.normalize();
8  V.normalize();
9  ambient = currentSphere.material.ambient_material *
       ↪ lights[0].ambientPower;
10 diffuse = currentSphere.material.diffuse_material *
       ↪ lights[0].diffusePower * Vec3::dot(L,N);
11 specular = currentSphere.material.specular_material *
       ↪ lights[0].specularPower * pow(Vec3::dot(R,V),
       ↪ currentSphere.material.shininess);
```

Enfin, j'ai mis à jour la couleur à retourner. Comme indiqué dans le modèle de Phong, la couleur d'un point est la somme des réflexions ambiantes, diffuses et spéculaires en ce point.

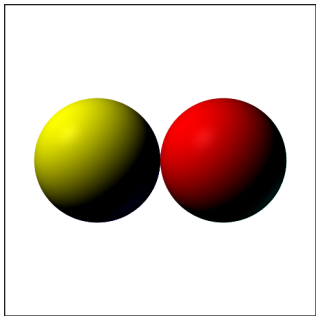
```
1  color = ambient + diffuse + specular;
```

Voici quelques rendus avec le modèle de Phong en action.

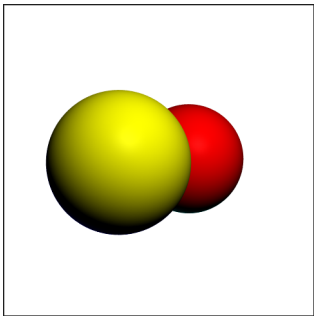
N.B. : J'ai choisi de changer la couleur d'arrière-plan par du blanc afin de mieux observer les phénomènes liés à l'éclairage.



(a) Sphère simple

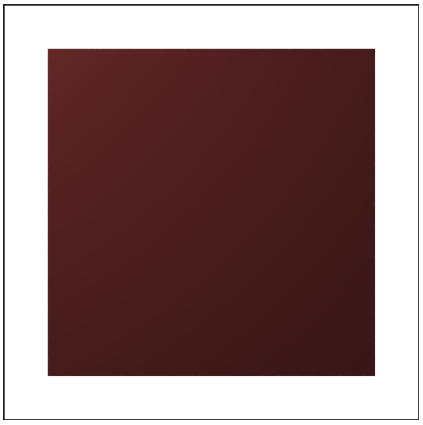


(b) Deux sphères

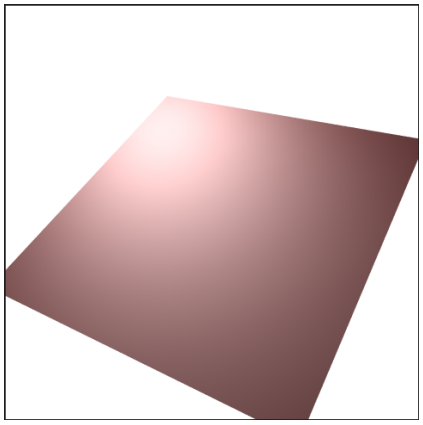


(c) Deux sphères après déplacement de la caméra

Figure 4: Rendus de sphères grâce au lancer de rayons et au modèle de Phong avec $I_{\text{sa}} = I_{\text{sd}} = 1$ et $I_{\text{ss}} = 0.4$

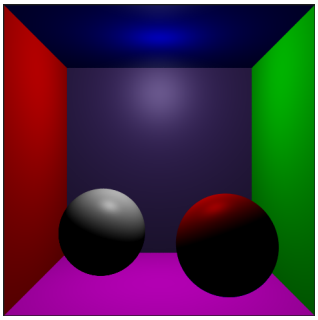


(a) Carré simple

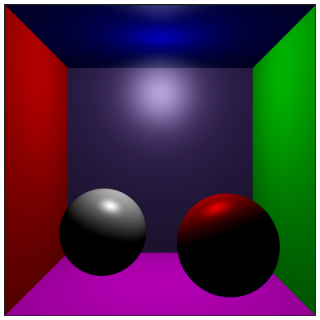


(b) Carré penché pour mettre en évidence la réflexion spéculaire

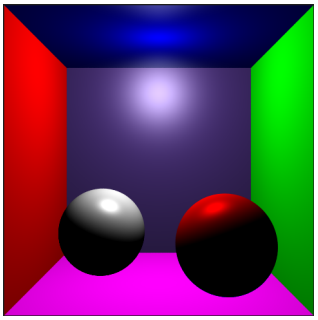
Figure 5: Rendus de carrés grâce au lancer de rayons et au modèle de Phong avec $I_{\text{sa}} = I_{\text{sd}} = 0.5$ et $I_{\text{ss}} = 1$



(a) $I_{sa} = I_{sd} = 0.7$ et $I_{ss} = 0.2$



(b) $I_{sa} = I_{sd} = 0.7$ et $I_{ss} = 0.5$



(c) $I_{sa} = I_{sd} = 1$ et $I_{ss} = 0.6$

Figure 6: Rendus de boîtes de Cornell grâce au lancer de rayons et au modèle de Phong

3.2 Ajout d'ombres dures

Pour implémenter les ombres dures, il faut vérifier si la lumière est bloquée par un autre objet avant qu'elle n'atteigne le point d'intersection. Pour faire ceci, on tire un rayon partant du point d'intersection en direction de la source lumineuse. Si ce rayon rencontre un objet en route, alors le point d'intersection originel est dans l'ombre et la couleur doit donc passer à noir. Voici ce que j'ai rajouté dans **rayTraceRecursive**, pour chacun des types d'objets.

```
1 float epsilon = 0.001f;
2 distanceToLight = L.length(); // Avant de normaliser L
3 harshShadowRay = Ray(intersectionPoint + epsilon * N, L);
4 shadowIntersection = computeIntersection(harshShadowRay);
5 if(shadowIntersection.intersectionExists &&
    ↪ shadowIntersection.t < distanceToLight) {
6     diffuse = Vec3(0.0f,0.0f,0.0f);
7     specular = Vec3(0.0f,0.0f,0.0f);
8 }
```

Après avoir complété mes fonctions d'intersection pour ignorer les petites valeurs de t, voici à quoi ressemblent les ombres dures.

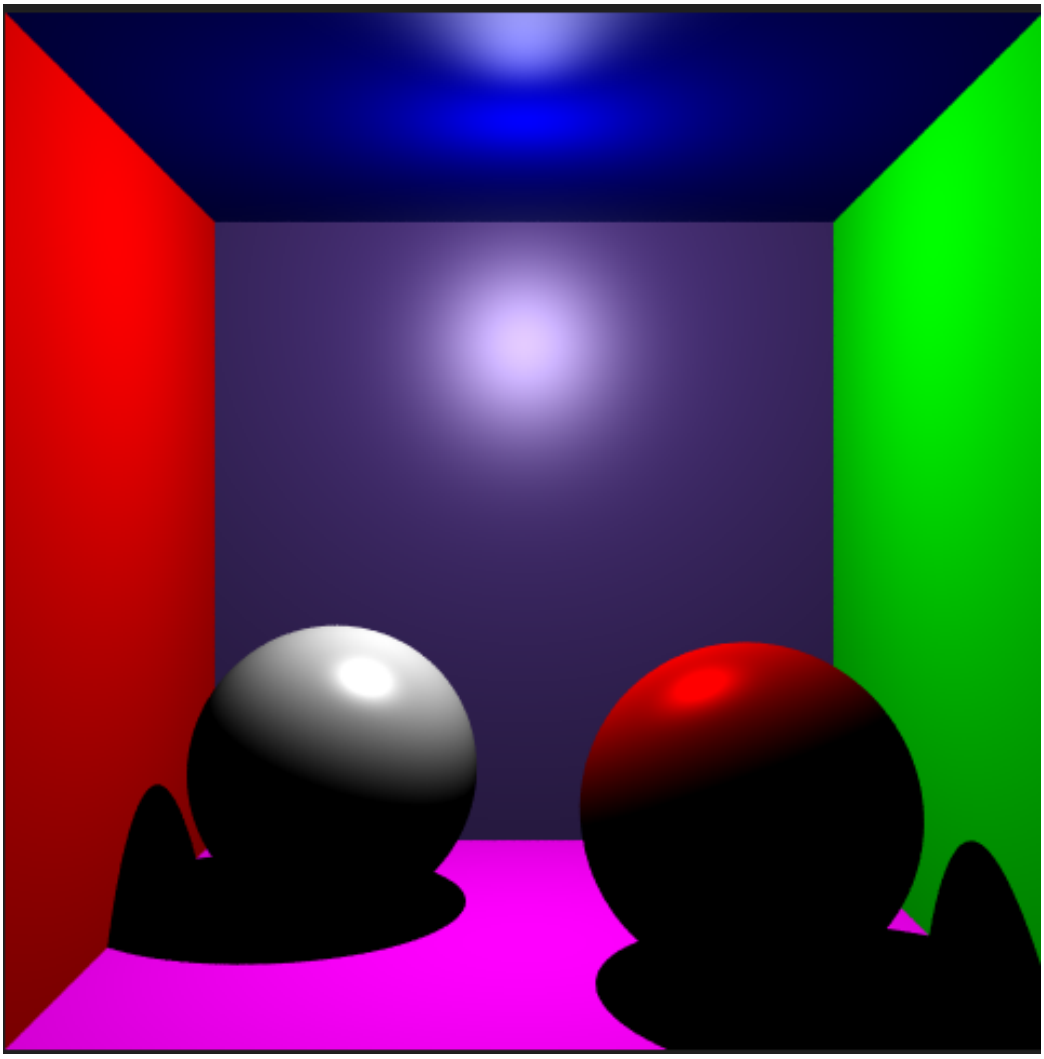


Figure 7: Rendu d'une boîte de Cornell avec ombres dures

3.3 Ajout d'ombres douces

Finalement, il était demandé d'ajouter des ombres douces, pour rendre les ombres plus réalistes. J'ai décidé de garder une lumière sphérique, et de simuler un carré imaginaire autour de celle-ci pour y échantillonner des points à l'intérieur. Pour cela, j'ai créé une fonction **sampleAreaLight** dans **Scene.h**.

```
Vec3 sampleAreaLight(const Light &light, float squareSide ) {

    Vec3 center = light.pos;
    Vec3 normal(0.0f,0.0f,-1.0f);
    Vec3 upVector(0.0f,1.0f,0.0f);
    Vec3 rightVector(1.0f,0.0f,0.0f);

    upVector = upVector * (squareSide / 2.0f);
    rightVector = rightVector * (squareSide / 2.0f);

    float u = (float)rand()/(float)(RAND_MAX) - 0.5f;
    float v = (float)rand()/(float)(RAND_MAX) - 0.5f;

    Vec3 sampledPoint = center + (u * rightVector) + (v * upVector);

    return sampledPoint;
}
```

Ensuite, j'ai remplacé la partie de calcul des ombres dures par le calcul des ombres douces dans **rayTraceRecursive**.

```
for(int i = 0; i < nbSamplesSmoothShadows; i++) {
    sampledPointPosition = sampleAreaLight(lights[0],
        ↪ squareSide);
    lightDirectionForSampledPoint = sampledPointPosition
        ↪ - intersectionPoint;
    distanceToLight = lightDirectionForSampledPoint.
        ↪ length();
    lightDirectionForSampledPoint.normalize();
    shadowRay = Ray(intersectionPoint + epsilon * N,
        ↪ lightDirectionForSampledPoint);
    shadowIntersection = computeIntersection(shadowRay);
    if(!(shadowIntersection.intersectionExists &&
        ↪ shadowIntersection.t < distanceToLight)) {
         shadowCounter += 1.0f;
    }
}
...
shadowCounter /= nbSamplesSmoothShadows;
diffuse *= shadowCounter;
specular *= shadowCounter;
```

Enfin, j'ai pu observer un rendu avec des ombres douces.

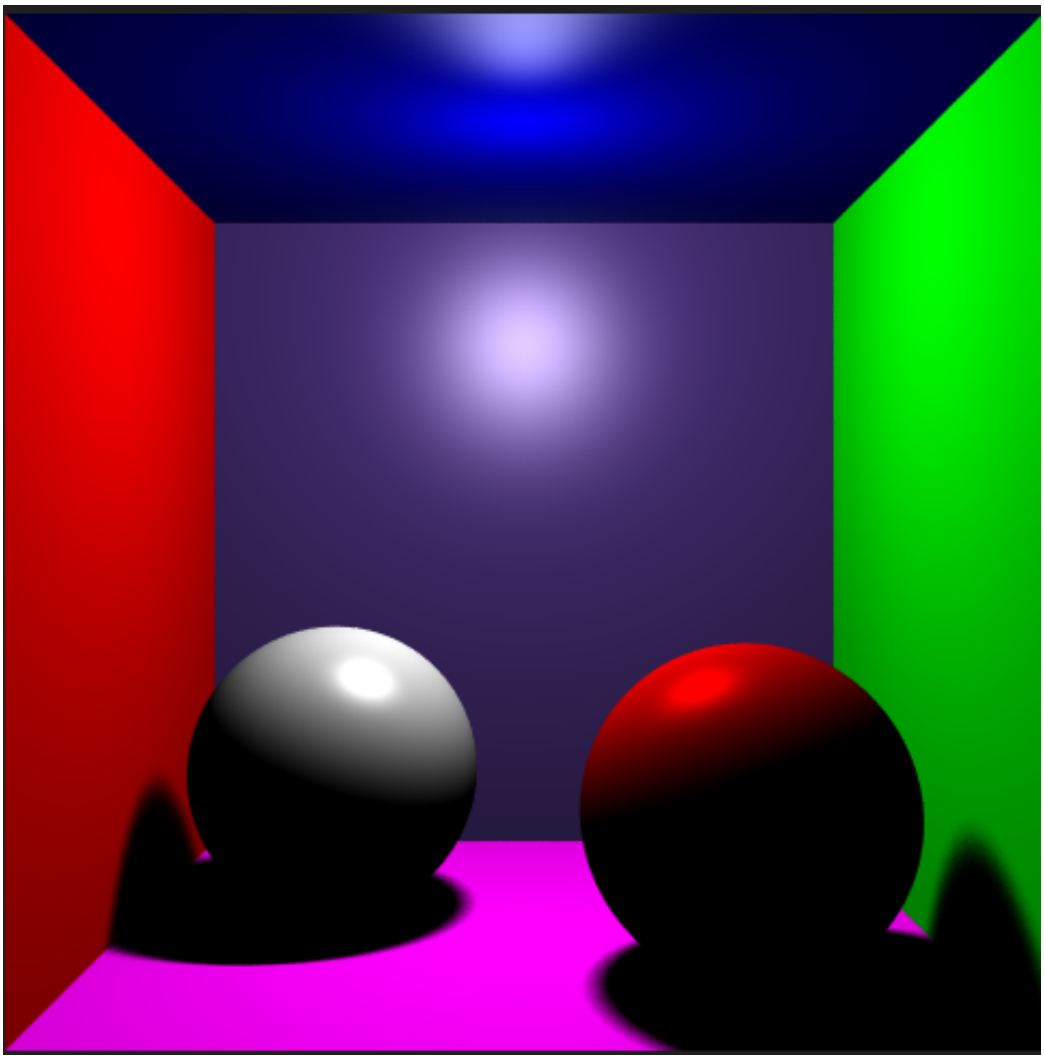


Figure 8: Rendu d'une boîte de Cornell avec ombres douces, avec 1 rebond par rayon parent et 10 points échantillonnés par rayon

4 Conclusion

Ces deux phases marquent le début de ce projet de raytracer. À partir de cette base solide, je vais pouvoir améliorer mon projet en implémentant diverses fonctionnalités (k-d tree, gestion de la transparence,...).
Merci pour le temps et l'attention que vous avez consacrés à la lecture de ce compte-rendu.