Programmation 3D

Raytracer rendu terminal

Louis Jean
Master 1 IMAGINE
Université de Montpellier

7 janvier 2023

Table des matières

1 Introduction

2 Phases 1 et 2

2.1 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.

2.2 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.

2.2.1 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.

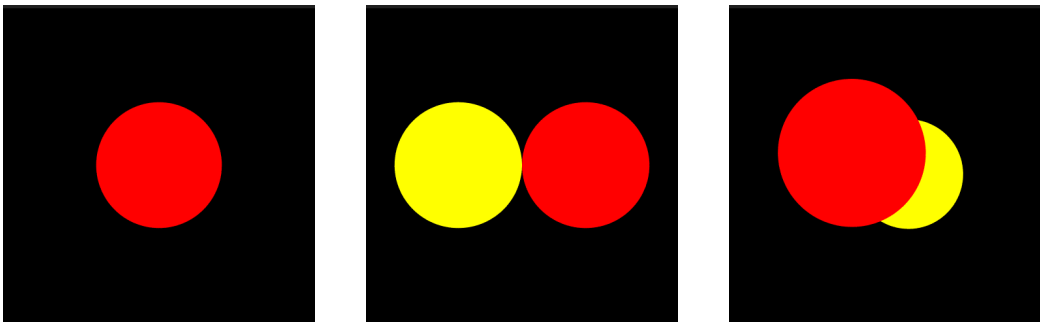
```
1  Vec3 rayTraceRecursive(Ray ray, int NRemainingBounces) {
2
3      RaySceneIntersection rayScene Intersection =
           ↪ computeIntersection(ray);
4      Vec3 intersectionPoint(0.0f,0.0f,0.0f);
5      Vec3 normalToIntersectionPoint(0.0f,0.0f,0.0f);
6      Vec3 reflectedDirection(0.0f,0.0f,0.0f);
7      Vec3 color(0.0f,0.0f,0.0f); // Fond noir
8
9      if(!rayScene Intersection.intersectionExists) {
10         return color;
11     }
12
13     switch(rayScene Intersection.typeOfIntersectedObject)
           ↪ {
14         case MeshType_Sphere:
15             intersectionPoint = rayScene Intersection.
                   ↪ raySphereIntersection.intersection;
16             normalToIntersectionPoint =
                   ↪ rayScene Intersection.
                   ↪ raySphereIntersection.normal;
17             color = spheres[rayScene Intersection.
                   ↪ objectIndex].material.diffuse_material;
18         break;
19
20          default:
21          break;
22     }
23
24     // Calcul de la direction du rayon réfléchi
25     reflectedDirection = ray.direction() - (2 * Vec3::dot
           ↪ (ray.direction(), normalToIntersectionPoint) *
           ↪ normalToIntersectionPoint);
26
27     if(NRemainingBounces > 0) {
28         rayTraceRecursive(Ray(intersectionPoint,
               ↪ reflectedDirection),NRemainingBounces - 1);
29     }
30
31     return color;
32 }
```

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

2.2.2 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.

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

```
    Vec3 getIntersectionPoint(Line const & L) const {

    Vec3 result = Vec3(-100.0f,-100.0f,-100.0f);

    if(!isParallelTo(L)) {
        float D = Vec3::dot(m_center,m_normal);
        float t = (D - Vec3::dot(L.origin(),m_normal)) /
            ↪ Vec3::dot(L.direction(),m_normal);
        if(t > 0) {
            result = L.origin() + t * L.direction();
        }
    }
    return result;
}
```

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

2.2.3 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**.

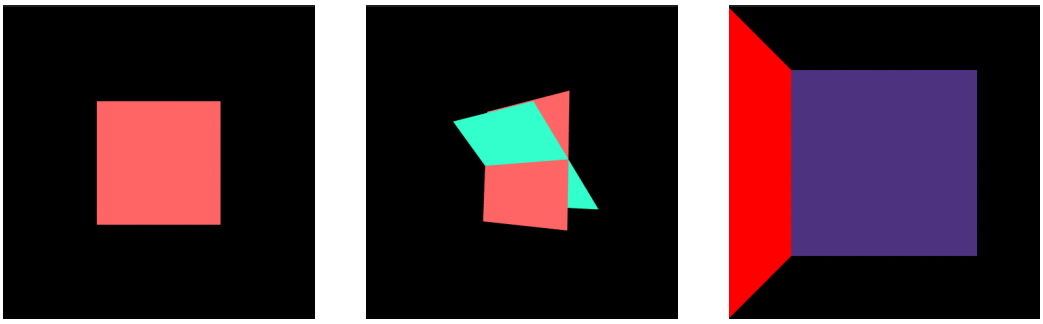
```
1  RaySquareIntersection intersect(const Ray &ray) const {
2      RaySquareIntersection intersection;
3      intersection.intersectionExists = false;
4      // Prise en compte des potentielles transformations
5      Vec3 bottomLeft = vertices[0].position;
6      Vec3 bottomRight = vertices[1].position;
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

2.2.4 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.

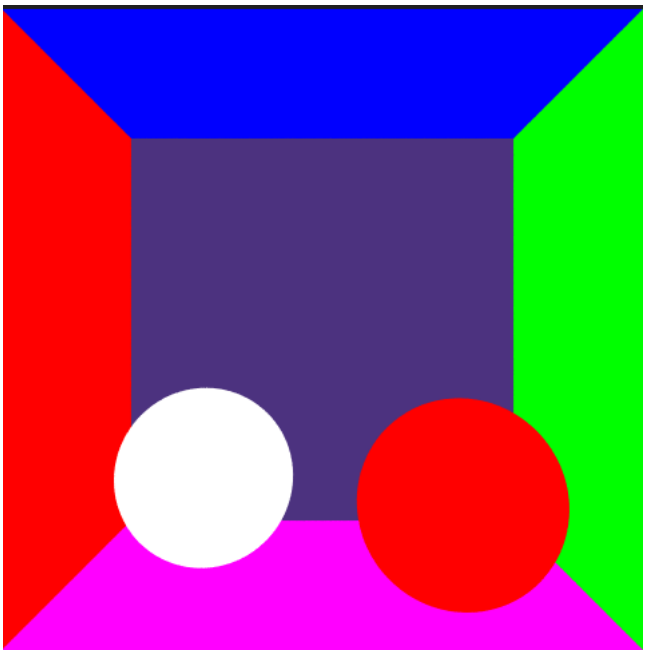


Figure 3: Rendu d'une boîte de Cornell

2.2.5 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.

2.3 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.

2.3.1 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.

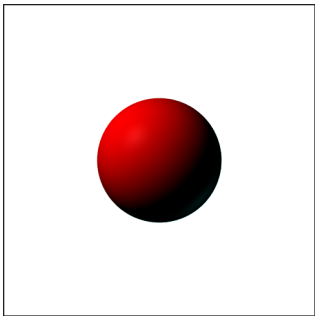
```
1  N = normalToIntersectionPoint;
2  L = lights[0].pos - intersectionPoint;
3  R = (2 * Vec3::dot(L,N) * N) - L;
4  V = ray.origin() - intersectionPoint;
5  N.normalize();
6  L.normalize();
7  R.normalize();
8  V.normalize();
9  ambient = currentSphere.material.ambient_material *
       ↪ lights[0].ambientPower;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

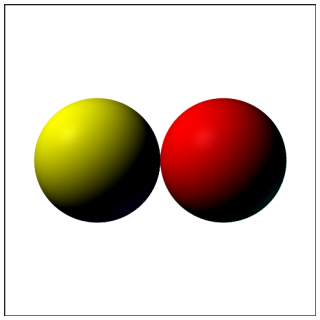
```
1  color = ambient + diffuse + specular;
```

Voici quelques rendus avec le modèle de Phong en action.

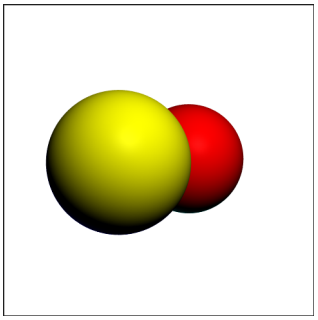
N.B. : J'ai choisi de changer la couleur d'arrière-plan par du blanc afin de mieux observer les phénomènes liés à l'éclairage.



(a) Sphère simple

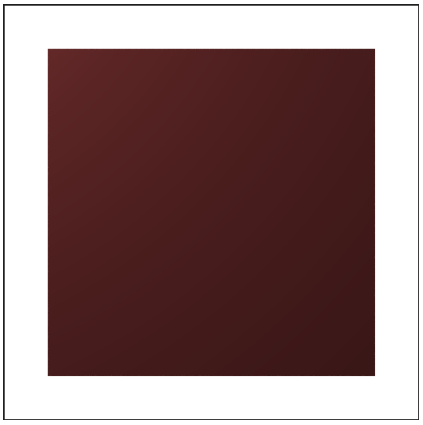


(b) Deux sphères

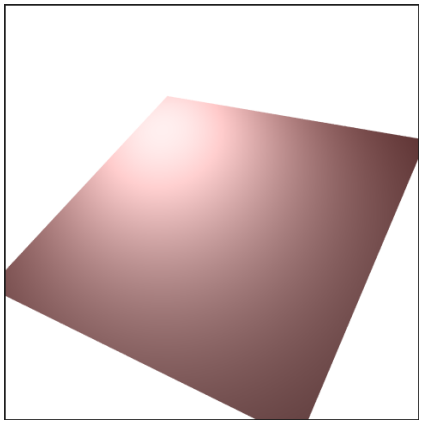


(c) Deux sphères après déplacement de la caméra

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

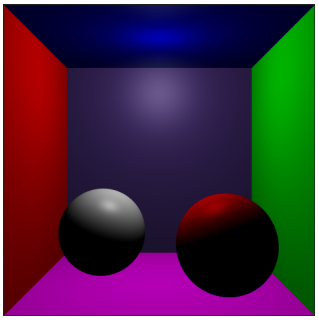


(a) Carré simple

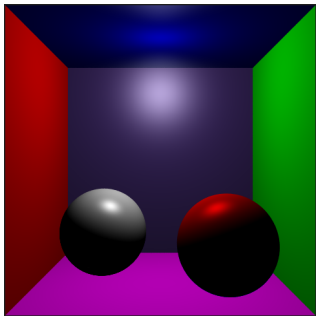


(b) Carré penché pour mettre en évidence la réflexion spéculaire

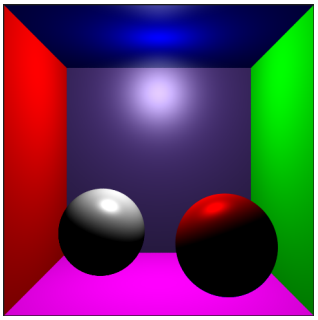
Figure 5: Rendus de carrés grâce au lancer de rayons et au modèle de Phong avec $I_{\text{sa}} = I_{\text{sd}} = 0.5$ et $I_{\text{ss}} = 1$



(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

2.3.2 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.

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

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

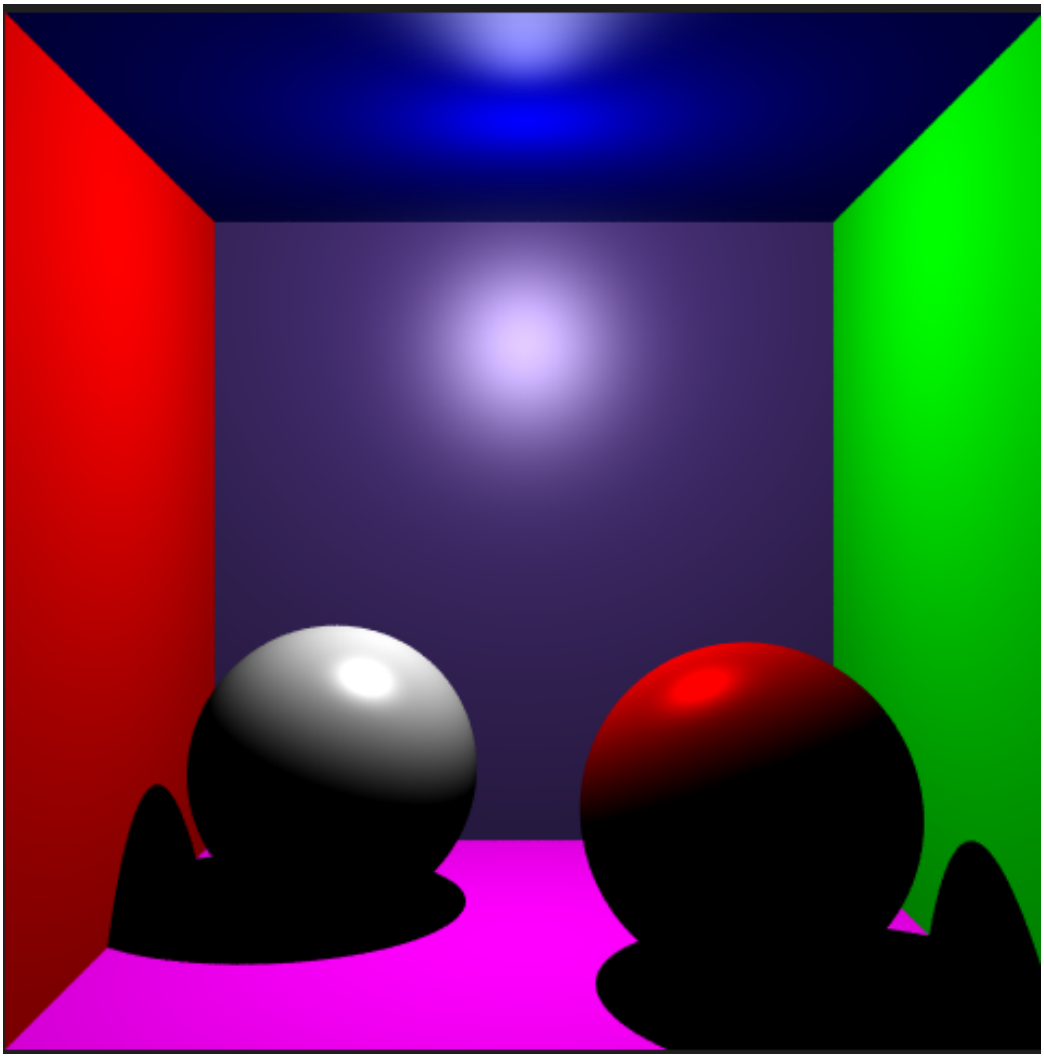


Figure 7: Rendu d'une boîte de Cornell avec ombres dures

2.3.3 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.

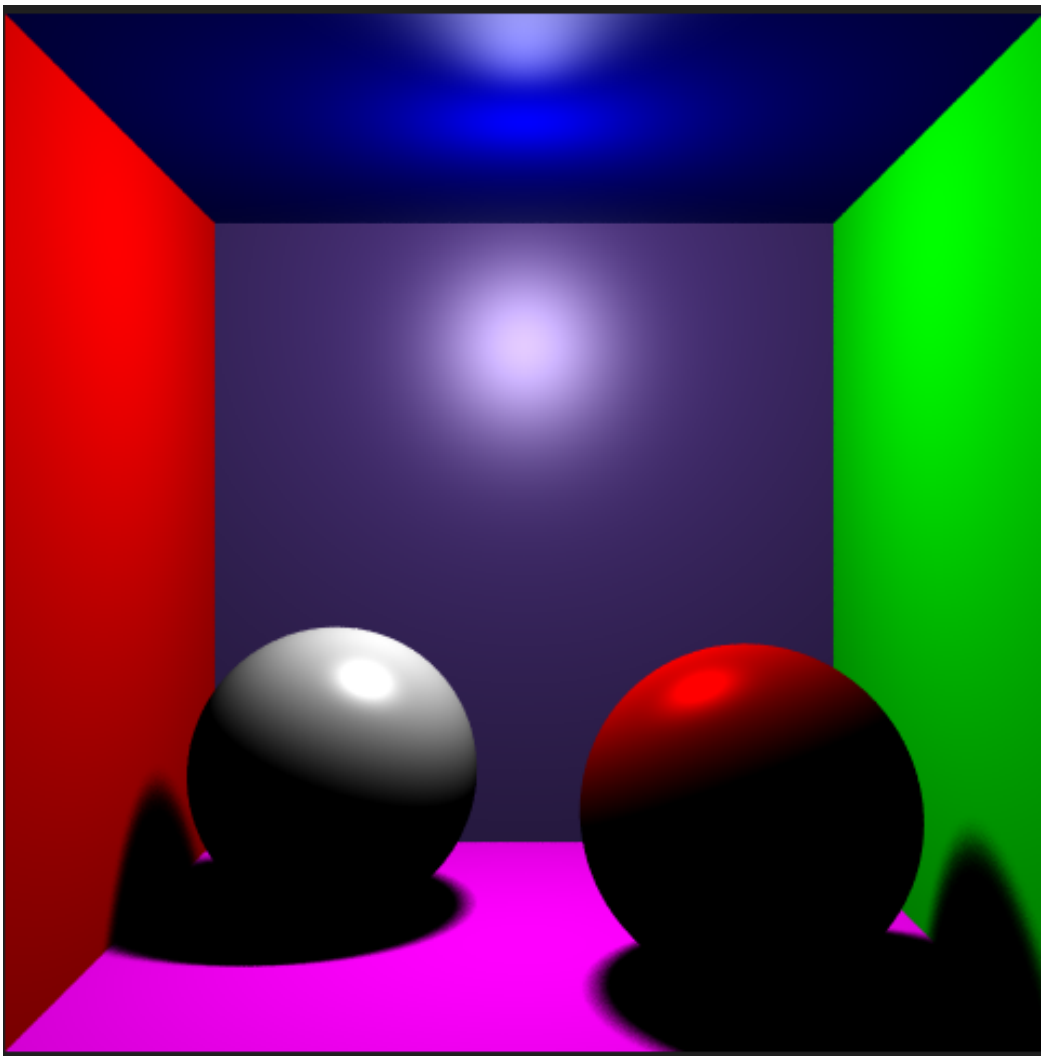


Figure 8: Rendu d'une boîte de Cornell avec ombres douces, avec 1 rebond par rayon parent et 10 points échantillonnés par rayon

2.4 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,...).

3 Phase 3

3.1 Introduction

Poursuivant l'aventure du raytracer, la troisième phase de ce projet a mis l'accent sur le peaufinage de la finesse des interactions lumineuses avec les surfaces complexes. Au programme : le calcul minutieux des intersections des rayons avec des maillages, l'interpolation des valeurs de sommets pour un rendu nuancé et la création d'une sphère réfléchissante. J'en ai aussi profité pour faire du nettoyage dans mon code.

3.2 Intersection rayon-triangle

Premièrement, afin de pouvoir charger un maillage 3D (au format **off**), il a fallu calculer l'intersection rayon-triangle. Après avoir calculé l'intersection du rayon avec le plan support du triangle, je suis ensuite passé aux coordonnées barycentriques. Pour cela, j'ai d'abord tenté d'implémenter la méthode vue dans le cours en utilisant la méthode **area** du fichier **Triangle.h**, en vain. J'ai alors choisi la méthode algébrique en m'inspirant de cet article : `https://codeplea.com/triangular-interpolation`.

```
1 Vec3 getIntersectionPointWithSupportPlane(Line const & L)
    ↪  const {
2         Vec3 result = Vec3(-100.0f,-100.0f,-100.0f);
3         if (isParallelTo(L)) return result;
4         Vec3 P0 = L.origin();
5         Vec3 P1 = m_c[0];
6         Vec3 D = L.direction();
7         float t;
8         t = Vec3::dot(P1 - P0,m_normal) / Vec3::dot(D,
              ↪ m_normal);
9         result = P0 + t * D;
10        return result;
11 }
```

Figure 9: Calcul de l'intersection entre le rayon et le plan support du triangle

```
void computeBarycentricCoordinates(Vec3 const &p, float &
    ↪ u0, float &u1, float &u2) const {
        Vec3 A = m_c[0];
        Vec3 B = m_c[1];
        Vec3 C = m_c[2];
        Vec3 v0 = B - A;
        Vec3 v1 = C - A;
        Vec3 v2 = p - A;
        float d00 = Vec3::dot(v0, v0);
        float d01 = Vec3::dot(v0, v1);
        float d11 = Vec3::dot(v1, v1);
        float d20 = Vec3::dot(v2, v0);
        float d21 = Vec3::dot(v2, v1);
        float denom = d00 * d11 - d01 * d01;
        u1 = (d11 * d20 - d01 * d21) / denom;
        u2 = (d00 * d21 - d01 * d20) / denom;
        u0 = 1.0f - u1 - u2;
}
```

Figure 10: Calcul des coordonnées barycentriques pour un point $\mathbf{p}$ donné

```
RayTriangleIntersection getIntersection(Ray const & ray)
    ↪ const {
     RayTriangleIntersection result;
     result.intersectionExists = false;
     Vec3 intersectionPoint =
        ↪ getIntersectionPointWithSupportPlane(ray);
     Vec3 originToIntersection = intersectionPoint - ray.
        ↪ origin();
     if(Vec3::dot(ray.direction(),originToIntersection) <
        ↪ 0) return result;
     float u0,u1,u2;
     computeBarycentricCoordinates(intersectionPoint, u0,
        ↪ u1, u2);
     if(u0 < 0 || u0 > 1 || u1 < 0 || u1 > 1 || u2 < 0 ||
        ↪ u2 > 1) return result;
     result.intersectionExists = true;
     result.intersection = intersectionPoint;
     result.w0 = u0;
     result.w1 = u1;
     result.w2 = u2;
     result.t = originToIntersection.norm();
     Vec3 normal = Vec3::cross(m_c[1] - m_c[0], m_c[2] -
        ↪ m_c[0]);
     normal.normalize();
     result.normal = normal;
     return result;
}
```

Figure 11: Calcul de l'intersection rayon-triangle

```
RayTriangleIntersection intersect( Ray const & ray )
    ↪ const {
     RayTriangleIntersection closestIntersection;
     closestIntersection.t = FLT_MAX;
     float triangleScaling = 1.000001;
     int nbTriangles = triangles.size();
     for(int i = 0; i < nbTriangles; i++) {
         unsigned int index0 = triangles_array[3 * i];
         unsigned int index1 = triangles_array[3 * i + 1];
         unsigned int index2 = triangles_array[3 * i + 2];
         MeshVertex vertex0 = vertices[index0];
         MeshVertex vertex1 = vertices[index1];
         MeshVertex vertex2 = vertices[index2];
         Triangle currentTriangle = Triangle(vertex0.
            ↪ position*triangleScaling,vertex1.position*
            ↪ triangleScaling,vertex2.position*
            ↪ triangleScaling);
         RayTriangleIntersection currentIntersection =
            ↪ currentTriangle.getIntersection(ray);
         if(currentIntersection.intersectionExists &&
            ↪ currentIntersection.t < closestIntersection
            ↪ .t) {
             closestIntersection = currentIntersection;
             currentIntersection.tIndex = i;
         }
     }
     return closestIntersection;
}
```

Figure 12: Gestion de l'intersection rayon-triangle dans **Mesh.h**

Après un simple parcours des triangles du maillage considéré, voici ce que j'ai obtenu.

(a) Triangle simple (b) Maillage d'un cylindre (c) Maillage d'un blob

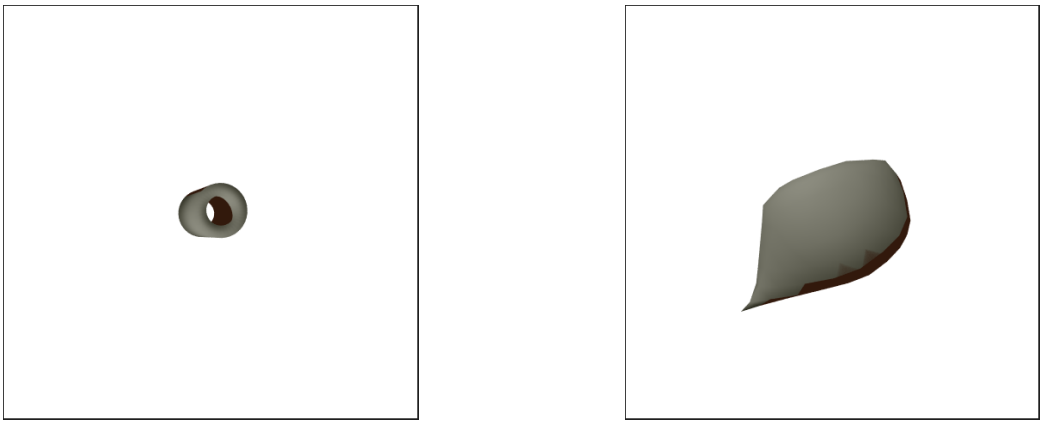
Figure 13: Rendus de maillages grâce au lancer de rayons

3.3 Interpolation des valeurs des sommets

Pour obtenir un rendu plus convaincant et réaliste, j'ai interpolé les valeurs des normales dans les maillages. Voici ce que j'ai rajouté dans la condition de la fonction **intersect**.

```
1 Vec3 interpolatedNormal = currentIntersection.w0 *
    ↪ vertex0.normal + currentIntersection.w1 * vertex1.
    ↪ normal + currentIntersection.w2 * vertex2.normal;
2 interpolatedNormal.normalize();
3 closestIntersection.normal = interpolatedNormal;
```

Figure 14: Interpolation des normales



(a) Maillage d'un cylindre

(b) Maillage d'un blob

Figure 15: Rendus de maillages grâce au lancer de rayons avec interpolation des normales

Je n'ai pas trouvé l'intêret d'interpoler les couleurs des sommets sachant que l'on définit un **Material** par maillage. Pour les coordonnées de textures (**uv**), j'implémenterai l'interpolation lorsque je commencerai à m'intéresser aux textures.

3.4 Surfaces réfléchissantes

Afin d'obtenir un objet totalement réfléchissant (donc un miroir), j'ai créé une condition sur le type du **Material** considéré dans **Scene.h**, en jouant avec l'appel récursif à **rayTraceRecursive**. Si le point tapé est de type miroir, alors il prend la couleur qu'ira taper son rayon réfléchi.

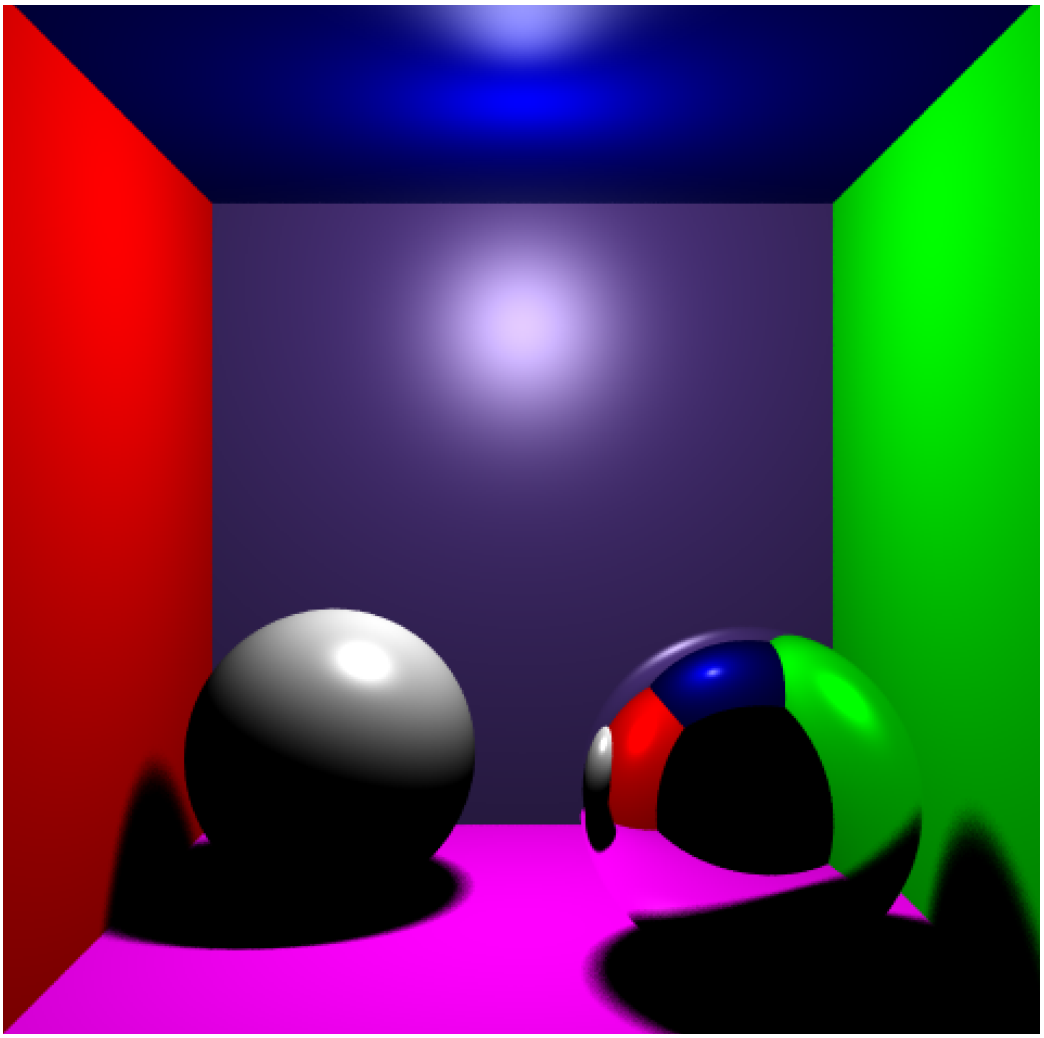


Figure 16: Sphère miroir

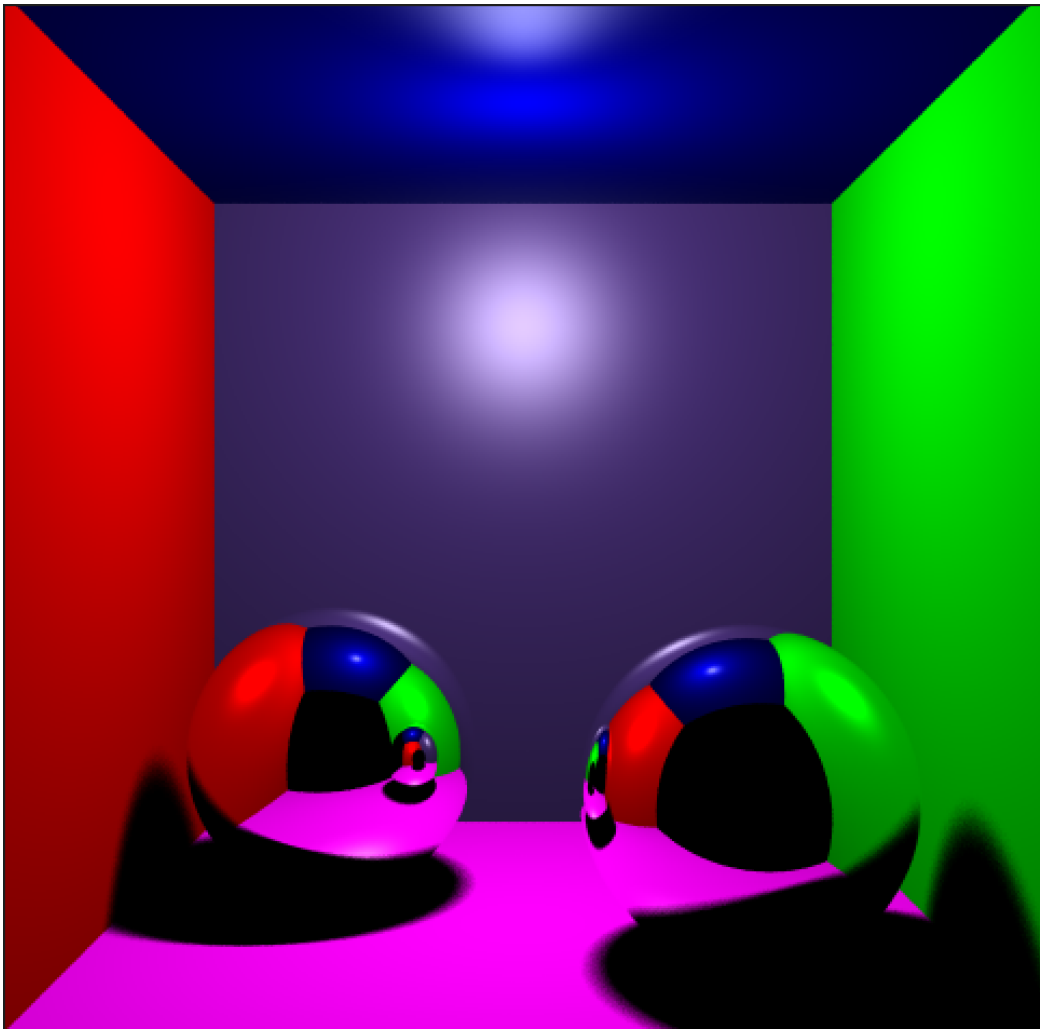


Figure 17: Deux sphères miroir

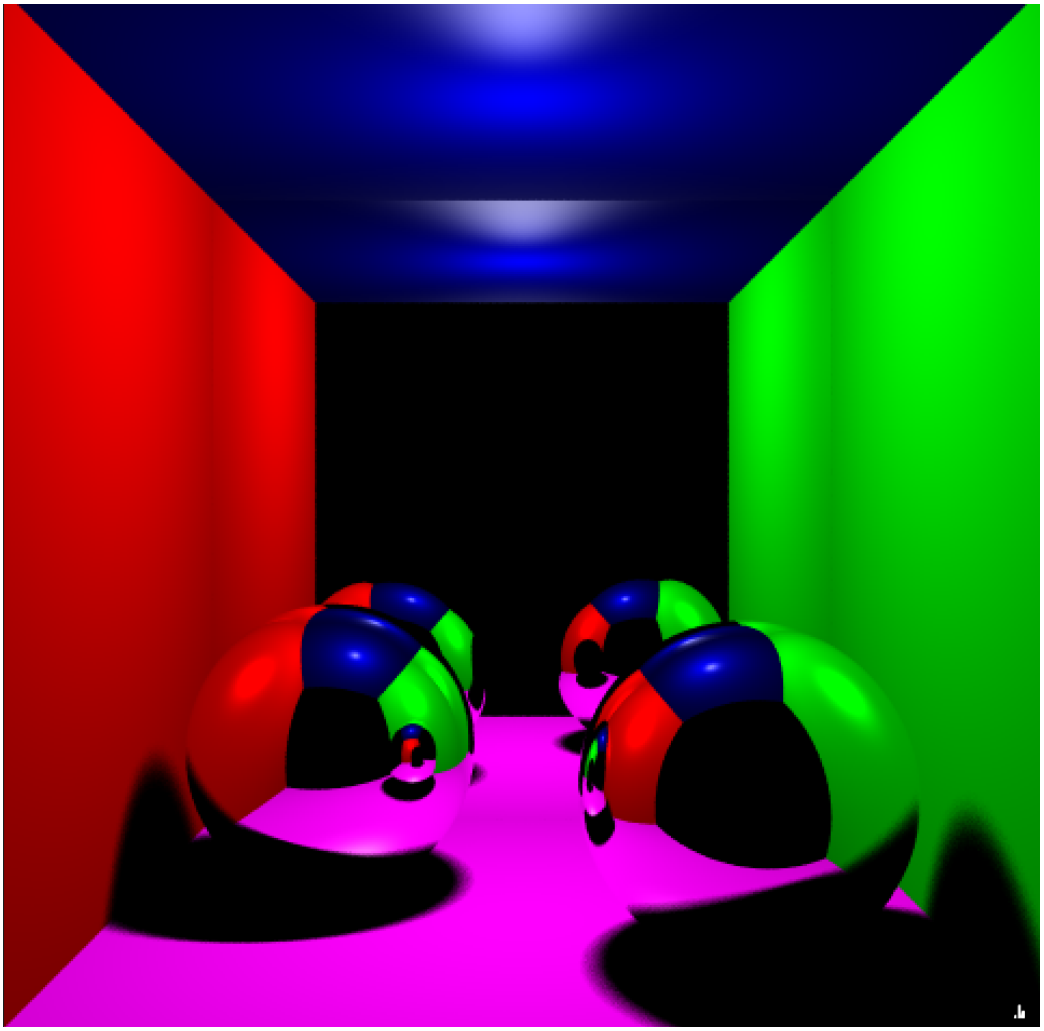


Figure 18: Deux sphères miroir et un mur miroir

3.5 Conclusion

La phase 3 du projet de ray tracing a marqué un progrès significatif avec l'intégration réussie d'intersections de maillages, d'interpolation de normales et de la création de sphères réfléchissantes. Ces avancées ont non seulement amélioré le réalisme visuel des rendus mais ont également posé les bases pour des développements plus poussés. L'expérience acquise et les défis relevés préparent le terrain pour l'étape suivante, qui consistera à améliorer les performances de l'application à l'aide d'une structure d'accélération, et qui mettra en place la réfraction des rayons.

4 Phase 4

4.1 Introduction

La phase 4 du projet s'est concentrée sur des éléments de complexité et de raffinement avancés avec l'ajout de la réfraction et des structures d'accélération comme le k-d tree. Cette étape a permis d'enrichir la représentation visuelle en intégrant des sphères transparentes qui ajoutent une couche supplémentaire de réalisme par le jeu complexe de la lumière. Elle a aussi marqué une amélioration significative des performances, essentielle pour gérer des scènes plus élaborées. J'en ai également profité pour refactoriser mon code afin de le rendre plus propre et lisible, notamment en séparant mon code en plusieurs fonctions, et ai aussi amélioré les performances en corrigeant des fuites de mémoire.

4.2 Surfaces transparentes et réfraction

Pour implémenter la réfraction de la lumière, je me suis basé sur la loi de Snell-Descartes, qui établit une relation entre les angles d'incidence et de réfraction lorsque la lumière passe d'un milieu à un autre. Cette loi stipule que le produit de l'indice de réfraction du premier milieu et le sinus de l'angle d'incidence est égal au produit de l'indice de réfraction du second milieu et le sinus de l'angle de réfraction.
Mathématiquement, cela s'exprime comme suit : si n_i, n_r sont les indices respectifs de réfraction des milieux incident et réfractant, et θ_i, θ_r les angles d'incidence et de réfraction, alors la relation est :

$$n_i \cdot \sin(\theta_i) = n_r \cdot \sin(\theta_r)$$

L'effet de Fresnel, que j'ai intégré, décrit comment la quantité de lumière qui est réfléchie ou réfractée change en fonction de l'angle d'incidence. Il révèle que les surfaces sont plus réfléchissantes sous des angles rasants par rapport à la normale. Pour modéliser cet effet, j'ai utilisé l'approximation de Schlick, une formule simplifiée qui calcule la réflectivité en fonction de l'angle d'incidence et des indices de réfraction des deux milieux. Cela permet de rendre les rendus encore plus réalistes en simulant comment les matériaux réagissent différemment à la lumière selon l'angle de vue.
Voici comment cela se traduit au niveau de la programmation.

```
1  // Calcule la direction réfractée du rayon considéré
2  Vec3 computeRefractedDirection(const Vec3 &incident,
       ↪ const Vec3 &normal, const float &
       ↪ airRefractionIndice, const float &ior) {
3      float cosi = std::clamp(-1.0f, 1.0f, Vec3::dot(
           ↪ incident, normal));
4      float etai = airRefractionIndice, etat = ior;
5      Vec3 n = normal;
6      if (cosi < 0) {cosi = -cosi;} else {std::swap(etai,
           ↪ etat); n= -1*normal;}
7      float eta = etai / etat;
8      float k = 1 - eta * eta * (1 - cosi * cosi);
9      return k < 0 ? Vec3(0.0f,0.0f,0.0f) : eta * incident
           ↪ + (eta * cosi - sqrtf(k)) * n;
10 }
```

Figure 19: Fonction calculant la direction du rayon réfracté

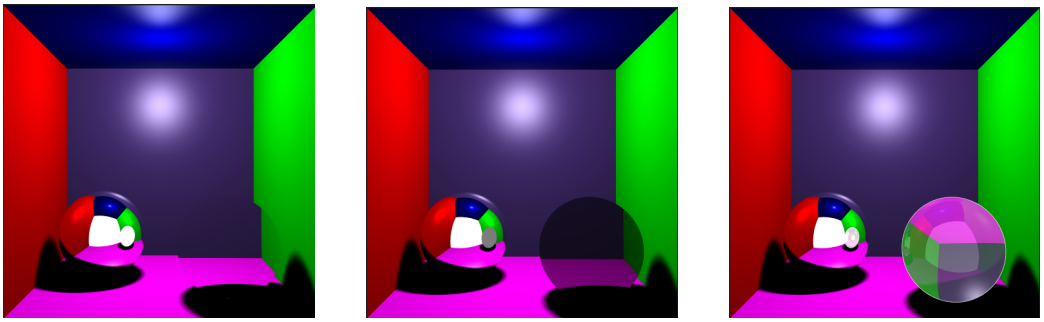
```
float computeFresnelEffect(const Vec3 &I, const Vec3 &N,
    ↪ const float &airRefractionIndice, const float &ior)
    ↪  {
    float cosi = std::clamp(-1.0f, 1.0f, Vec3::dot(I, N))
       ↪ ;
    float etai = airRefractionIndice, etat = ior;
    if (cosi > 0) { std::swap(etai, etat); }
    // Calcul de sin(theta_t)^2 via la loi de Snell-
       ↪ Descartes
    float sint = etai / etat * sqrtf(std::max(0.f, 1 -
       ↪ cosi * cosi));
    // Gestion totale de la réflexion interne
    if (sint >= 1) {
        return 1.0f;
    }
    else {
        float cost = sqrtf(std::max(0.f, 1 - sint * sint)
           ↪ );
        cosi = fabsf(cosi);
        float Rs = ((etat * cosi) - (etai * cost)) / ((
           ↪ etat * cosi) + (etai * cost));
        float Rp = ((etai * cosi) - (etat * cost)) / ((
           ↪ etai * cosi) + (etat * cost));
        return (Rs * Rs + Rp * Rp) / 2;
    }
}
```

Figure 20: Fonction calculant l'effet de Fresnel

```
if (m.type == Material_Glass && NRemainingBounces > 0) {
    float airRefractionIndice = 1.00029f;
    float materialRefractionIndice = m.index_medium;
    Vec3 refractedDirection = computeRefractedDirection(
        ↪ rayDirection, normalToIntersectionPoint,
        ↪ airRefractionIndice, materialRefractionIndice);
    refractedDirection.normalize();
    Vec3 refractedColor = Vec3(0.0f, 0.0f, 0.0f);
    Vec3 reflectedColor = Vec3(0.0f, 0.0f, 0.0f);
    float reflectance = computeFresnelEffect(rayDirection
        ↪ , normalToIntersectionPoint,
        ↪ airRefractionIndice, m.index_medium);
    bool totalInternalReflection = refractedDirection ==
        ↪ Vec3(0.0f, 0.0f, 0.0f);
    if(!totalInternalReflection) {
        Vec3 refractedRayOrigin = intersectionPoint;
        refractedColor = m.transparency *
            ↪ rayTraceRecursive(Ray(refractedRayOrigin,
            ↪ refractedDirection), NRemainingBounces - 1)
            ↪ ;
    }
    reflectedColor = rayTraceRecursive(Ray(
        ↪ intersectionPoint, reflectedDirection),
        ↪ NRemainingBounces - 1);
    color = reflectance * reflectedColor + (1 -
        ↪ reflectance) * refractedColor;
}
```

Figure 21: Traitement du cas où un rayon tape un objet transparent

Et voici quelques rendus montrant une sphère transparente, avec n_r l'indice de réfraction et t l'indice transparence de la sphère.



(a) $n_r = 1.05$ et $t = 1$ (b) $n_r = 1.05$ et $t = 0.7$ (c) $n_r = 2$ et $t = 1$

Figure 22: Rendus de sphères transparentes

On remarque que plus l'indice de transparence est petit, moins la sphère laisse passer de lumière. Sur la figure (c), j'ai volontairement choisi une valeur aberrante de n_r pour mettre en exergue l'effet de Fresnel.
N.B. : on observe que le reflet de la sphère transparente dans la sphère miroir n'est pas le bon. Je n'ai pas réussi à corriger ce problème.

Voici un petit rendu d'un maillage de cylindre imbriqué dans une sphère transparente. On remarque bien l'effet de réfraction.

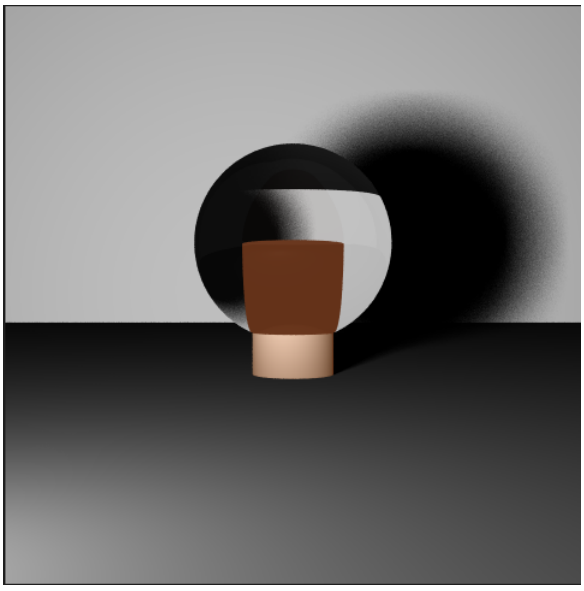


Figure 23: Cylindre imbriqué dans une sphère transparente

4.3 Structure d'accélération : le k-d tree

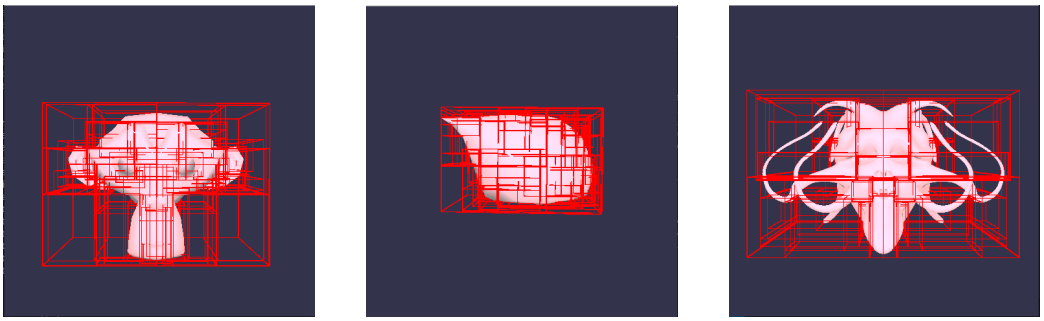
Passons au gros morceau de cette phase 4. Implémenter un k-d tree n'a pas été de tout repos, mais je m'en suis étonnamment assez bien sorti. Pour cette section, j'ai fait le choix de ne pas mettre de code dans le compte-rendu, car cela aurait été trop indigeste. J'ai choisi de créer trois classes : **AABB**, **KdNode** et **KdTree**. Une AABB (Axis-Aligned Bounding Box), représentée par **AABB**, est une boîte englobante alignée sur les axes x,y et z, qui se compose d'une position minimum et d'une position maximum, et qui permet de circonscrire l'espace de recherche d'intersections. Un nœud, structuré par **KdNode**, est composé d'une AABB, de deux pointeurs vers ses fils gauche et droit, d'un attribut pour savoir s'il est feuille, et d'une liste de triangles (un sous-ensemble des triangles du maillage considéré). Un arbre, défini par **KdTree**, est constitué d'un nœud racine, et de tout un tas de méthodes permettant au final la traversée de l'arbre.

Dans chaque scène, un k-d tree par maillage est construit. Je n'ai implémenté le k-d tree que pour les maillages chargés, et non pour les objets créés "à la main" (sphères et carrés). Le k-d tree est construit de cette manière :

- on calcule l'AABB globale du maillage
- ensuite, on divise récursivement cet espace en deux, en choisissant un axe (x, y ou z) et un point de division qui sépare les triangles du maillage de manière équilibrée (ici, j'ai choisi l'heuristique la plus simple, à savoir l'index médian de la liste des triangles).
- pour chaque division, on calcule les AABB des sous-ensembles de triangles.
- ce processus se poursuit jusqu'à ce qu'un des deux critères d'arrêt soit atteint (un nombre minimal de triangles dans un nœud ou une profondeur maximale de l'arbre).
- à la fin, chaque feuille de l'arbre contient une petite liste de triangles et leur AABB respective.

Lorsqu'un rayon est lancé, sans structure d'accélération comme le k-d tree, il faut tester l'intersection de ce rayon avec chaque objet de la scène, ce qui est coûteux en temps de calcul. Le k-d tree segmente l'espace de la scène en régions plus petites, chacune contenant un sous-ensemble d'objets. Ainsi, lors du test d'un rayon, seuls les objets situés dans les régions traversées par ce rayon sont testés pour les intersections. Cela réduit considérablement le nombre de tests nécessaires et accélère le rendu global.

J'ai implémenté la visualisation des AABBs pour bien voir comment mon k-d tree était construit, en appuyant sur la touche **k** lors de la prévisualisation, et voici le résultat. Les k-d tree sont ici générés avec une profondeur de 6 et un nombre minimal de 1 triangle par nœud. Pour changer les critères d'arrêt, il suffit de modifier les macros présentes en haut du fichier **KdTree.h**.



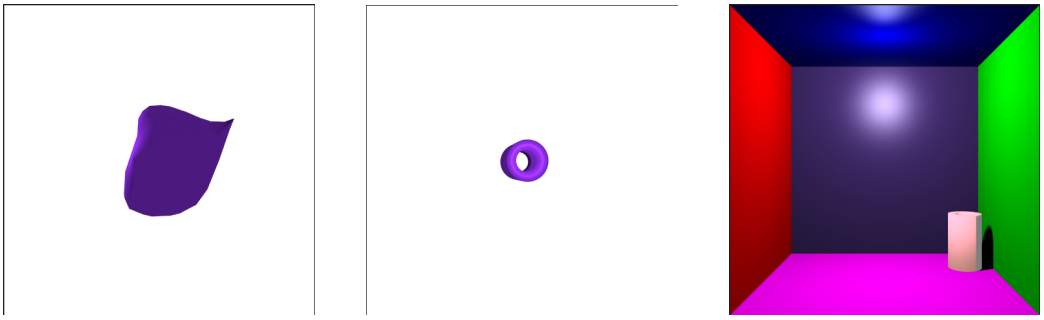
(a) K-d tree pour le maillage de Suzanne

(b) K-d tree pour le maillage de blob

(c) K-d tree pour le maillage de Sea Monster

Figure 24: Visualisation des AABBs

Au niveau des performances, on remarque une très nette amélioration. Voici quelques comparaisons de temps de rendu à scène égale et à rendu égal.



(a) 270 triangles
13s avec k-d tree
1m52s sans

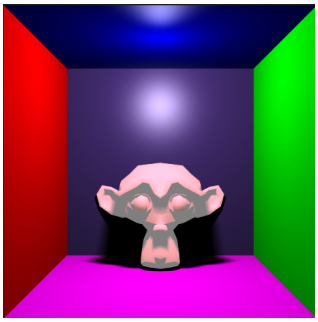
(b) 320 triangles
3s avec k-d tree
1m26s sans

(c) 320 triangles + boîte
11s avec k-d tree
7m50s sans

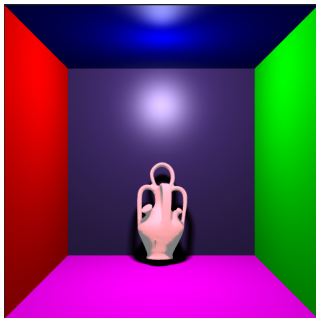
Figure 25: Rendus et comparaison de temps de rendu avec et sans k-d tree

On observe que selon la disposition des triangles, le temps de rendu varie assez significativement. Aussi, on remarque que dans la scène où le cylindre est dans la Cornell Box, le k-d tree augmente drastiquement les performances.

Grâce au k-d tree, j'ai aussi pu rendre des maillages avec beaucoup plus de triangles, en un temps raisonnable !



(a) 968 triangles + boîte
28 secondes



(b) 29734 triangles + boîte
Environ 3 minutes

(c) 78966 triangles
Environ 7 minutes

Figure 26: Rendus avec k-d tree

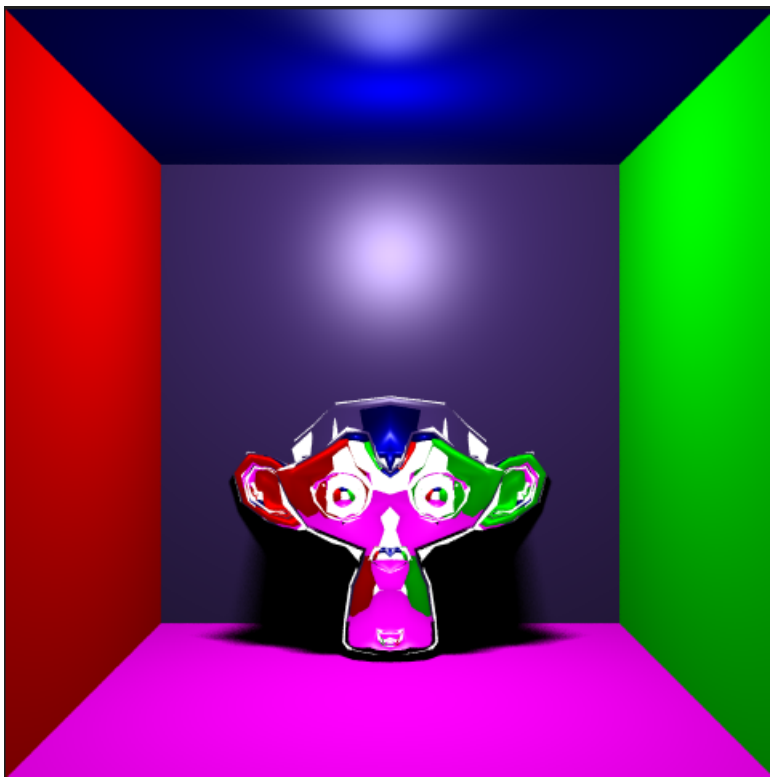


Figure 27: Rendu d'un maillage de Suzanne miroir

4.4 Effets et options

4.4.1 Profondeur de champ

Enfin, pour terminer ce projet, j'ai décidé d'implémenter la profondeur de champ. L'idée était de simuler une lentille, pour en approcher le comportement. Pour cela, j'ai rajouté deux attributs à la caméra : le rayon de la lentille, et la distance focale de cette même lentille. Ensuite, j'ai modifié la fonction qui s'occupe de lancer les rayons, pour échantillonner aléatoirement la position des rayons dans la lentille de la caméra. Puis, j'ai changé la direction du rayon, pour qu'il parte de sa nouvelle origine, vers la focale de la lentille (le centre). J'ai fait en sorte de pouvoir activer/désactiver l'effet de profondeur de champ avant de faire le rendu, en appuyant sur la touche **d**. Il faut ensuite régler les paramètres de la caméra dans la fonction **main** du fichier **main.cpp**. Actuellement, ces paramètres sont réglés pour la quatrième scène de mon programme.

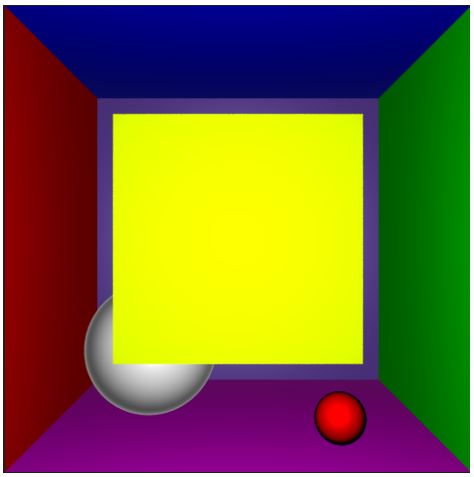
```
1 Vec3 Camera::random_in_unit_disk() {
2     Vec3 p;
3     do {
4         p = 2.0 * Vec3((float)rand() / RAND_MAX, (float)
              ↪ rand() / RAND_MAX, 0) - Vec3(1, 1, 0);
5     } while (p.squareLength() >= 1.0);
6     return p;
7 }
```

Figure 28: Simple fonction pour échantillonner des points dans un cercle unitaire

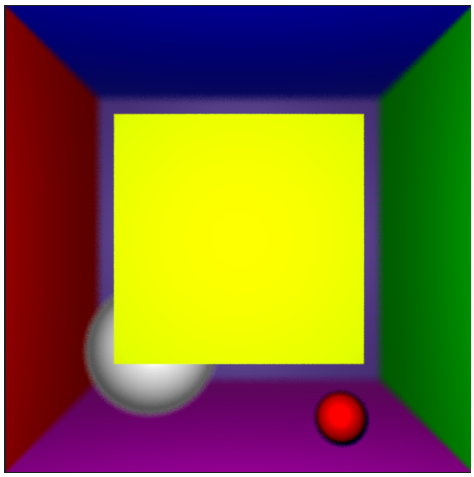
```
float u = ((float)(x) + (float)(rand())/(float)(RAND_MAX)
    ↪ ) / w;
float v = ((float)(y) + (float)(rand())/(float)(RAND_MAX)
    ↪ ) / h;
// this is a random uv that belongs to the pixel xy.
screen_space_to_world_space_ray(u,v,pos,dir);
if(depthOfField) {
    Vec3 randomPointOnLens = camera.getLensRadius() *
        ↪ Camera::random_in_unit_disk();
    Vec3 focalPoint = pos + dir * camera.getFocalDistance
        ↪ ();
    Vec3 newOrigin = pos + randomPointOnLens;
    Vec3 newDirection = focalPoint - newOrigin;
    newDirection.normalize();
    color = scenes[selected_scene].rayTrace(Ray(newOrigin
        ↪  , newDirection));
}
else {
    color = scenes[selected_scene].rayTrace(Ray(pos , dir
        ↪ ));
}
```

Figure 29: Modification dans la fonction **ray_trace_from_camera**

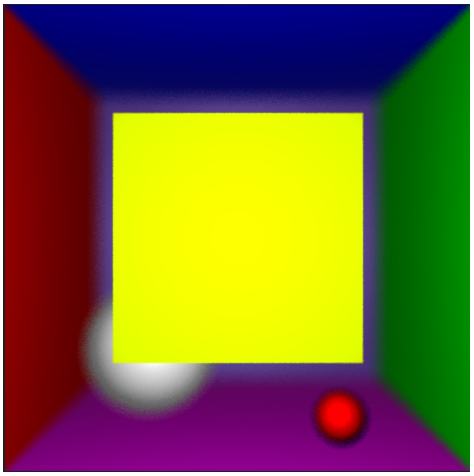
Voici les rendus obtenus, avec plusieurs lentilles différentes.



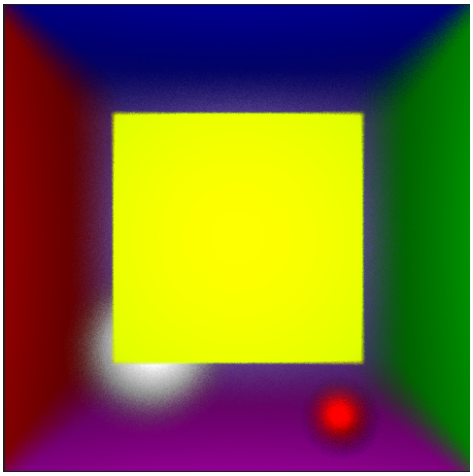
(a) Rendu témoin sans flou



(b) Rendu avec une lentille de taille 0.1



(c) Rendu avec une lentille de taille 0.2



(d) Rendu avec une lentille de taille 0.5

Figure 30: Rendus avec profondeur de champ

4.5 Conclusion

La phase 4 du projet de ray tracing représente une avancée majeure, marquée par l'implémentation réussie de la réfraction des rayons, l'introduction des kd-trees pour l'accélération des performances, et l'intégration de la profondeur de champ.

5 Conclusion du projet

J'ai pris beaucoup de plaisir à réaliser ce raytracer. Chaque phase a apporté son lot de défis techniques et d'opportunités d'apprentissage, aboutissant à un moteur de rendu capable de produire des images d'une qualité et d'un réalisme convaincants. Les compétences et connaissances acquises lors de ce projet pavent la voie à mes futures expérimentations dans le domaine de l'informatique graphique. J'aurai aimé ajouter plus d'options, notamment la gestion des textures, mais avec les révisions des examens, j'ai préféré m'arrêter là pour ne pas mettre tous mes efforts dans ce seul projet.

Merci pour le temps et l'attention que vous avez consacrés à la lecture de ce compte-rendu.