Compte Rendu HAI719I_RayTracing_Phase 3,4

Xihao Wang 21923448

4 novembre 2025

1 Introduction

Ce projet ajoute des effets intéressants au ray tracing précédent, y compris l'intersection entre les rays et les modèles mesh. Pour les sphères, j'ai ajouté des effets comme les sphères réfléchissantes et transparentes avec réfraction. J'ai également ajouté une structure KdTree pour accélérer le rendu des images. Enfin, j'ai ajouté d'autres effets, comme les textures et des sphères métalliques avec un effet flou (fuzzy).

2 L'intersection avec Meshes

Pour calculer l'intersection entre un rayon et un mesh, nous devons déterminer les points d'intersection entre le rayon et chaque triangle du mesh. En utilisant une méthode similaire aux sphères et quats, nous calculons d'abord la valeur de **t**, puis déterminons le point d'intersection **P** à l'aide de la formule : $P = t * dirction + origin$. Pour un triangle, la formule pour calculer t est : $t = (plane\_normal \cdot (v0 - origin))/(plane\_normal \cdot direction)$

Listing 1 – 1

```
1  //calculer seulement le t
2  float getIntersection_T_WithSupportPlane( Line const & L ) const {
3          return Vec3::dot(m_normal, m_c[0] - L.origin()) /
4          Vec3::dot(m_normal, L.direction());
5      }
```

Mais ce n'est pas encore suffisant, le rayon n'intersecte pas toujours le triangle (l'intersection avec le plan de ce triangle, OUI). Pour vérifier l'intersection avec le triangle lui-même, on utilise les coordonnées barycentrics (u_0, u_1, u_2). L'intersection est valide seulement si $0 < u_0, u_1 < 1$

Listing 2 – 1

```
1  void computeBarycentricCoordinates( Vec3 const & p ,
2      float & u0 , float & u1 , 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];
6
7          float d00 = Vec3::dot(v0, v0);
8          float d01 = Vec3::dot(v0, v1);
9          float d11 = ...; d20 =...; d21 =...;
10
11         float denom = d00 * d11 - d01 * d01;
12         if (denom == 0) {
13             u0 = u1 = u2 = -1;
14             return;
15         }
16         u1 = (d11 * d20 - d01 * d21) / denom;
```

```
        u2 = (d00 * d21 - d01 * d20) / denom;
        u0 = 1.0f - u1 - u2;
    }
```

Finalement nous pouvons calculer l'intersection avec les triangles.

Listing 3 – 1

```
RayTriangleIntersection getIntersection( Ray const & ray ) const {
        RayTriangleIntersection result;
        result.intersectionExists = false;
        // 1) check that the ray is not parallel to the triangle:
        if(isParallelTo(ray)){
            return result;
        }
        // 2) check that the triangle is "in front of" the ray:
        float t = getIntersection_T_WithSupportPlane(ray);
        if(t<0){
            return result;
        }
        Vec3 P = ray.origin() + t * ray.direction();
        // 3) check that the intersection point is inside the triangle:
        float u0,u1,u2;
        computeBarycentricCoordinates(P,u0,u1,u2);
        if(u0<0||u1<0||u2<0||u0>1||u1>1||u2>1){
            return result;
        }
        // 4) Finally, if all conditions were met
        result.intersectionExists = true;
        result. ...;
        return result;
    }
```

Une fois l'intersection avec triangle finie, on peut passer au Mesh.h pour implementer la methode *getIntersection()* a chaque mesh.

Listing 4 – 1

```
RayTriangleIntersection intersect( Ray const & ray ) const {
        RayTriangleIntersection closestIntersection;
        ...
        for(auto& tri:triangles){
            Vec3 v0 = vertices[tri[0]].position;
            Vec3 v1 = ...,v2 = ...;
            Triangle triangle(v0,v1,v2);
            RayTriangleIntersection intersection =
                    triangle.getIntersection(ray);

            if (...) {
                closestIntersection = intersection;
            }
        }
        return closestIntersection;
    }
```

Et voila les images de l'intersection avec meshes.

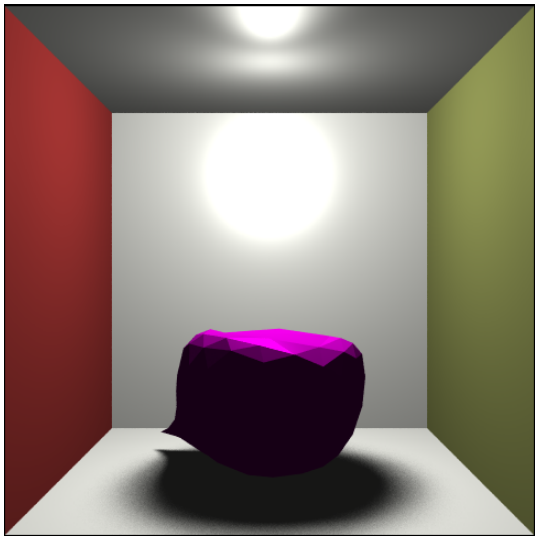

FIGURE 1 – blob-closed

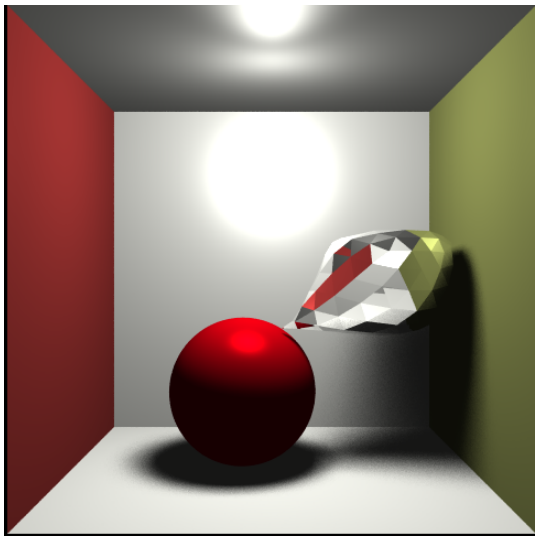

FIGURE 2 – blob & sphere

3 Sphère réfléchissante et transparente

3.1 La sphère réfléchissante

Le principe de réflexion est simple, le rayon se réfléchisse symétriquement par rapport à la normale au point d'intersection. La couleur obtenue par le rayon réfléchi correspond à la celle du point d'intersection.

Listing 5 – 2

```
1 if(reflection_material>0.f&&NRemainingBounces>0)
2 {
3     Vec3 reflectDir = ray.direction()
4             - 2*Vec3::dot(ray.direction(),normal)*normal;
5     reflectDir.normalize();
6     Ray rayReflection(intersectionPoint+epsilon*reflectDir,reflectDir);
7     reflecColor = rayTraceRecursive(rayReflection,NRemainingBounces-1);
8 }
```

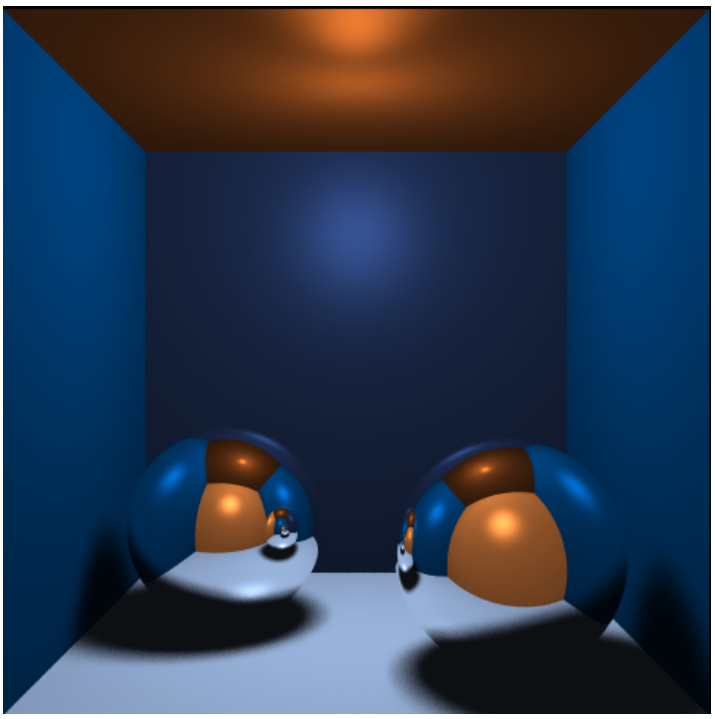

FIGURE 3 – two mirror sphere

3.2 La sphère transparente

La réfraction est un peu plus complexe : on calcule la direction du rayon initial r_0 d'abord, lorsqu'il entre dans la sphère, il donne un nouveau rayon r_1 (calculé à partir du ratio entre l'indice du milieu et celui de l'air). Ensuite, on calcule la direction de r_1 lorsqu'il sort de la sphere. On va obtenir un nouveau rayon r_2. La couleur obtenue par r_2 correspond à la couleur du point d'intersection de r_0 avec la sphère.

Listing 6 – 2

```
1  Vec3 computeRefraction(Vec3 I, Vec3 N, float index_medium){
2      Vec3 refracted_dir;
3      float cos1 = -Vec3::dot(I,N);
4      float sin2t = index_medium*index_medium*(1-cos1*cos1);
5      float cos2 = sqrt(1-sin2t);
6      refracted_dir = index_medium*I + (index_medium*cos1 - cos2)*N;
7      return refracted_dir; // return the direction of ray
8  }
9
10 // in rayTraceRecursive:
11 if (transparency) {
12     Vec3 inside_dir = computeRefraction(...);
13     if(inside_dir.length()>0){
14         Ray insideRay(...);
15         rayInside = computeIntersection(insideRay);
16         if(exist intersection with rayinside){
17             Vec3 insidePoint,insideNormal;
18             if(intersect to sphere) {insidePoint = ...;Normal = ...;}
19             outside_dir = computeRefraction();
20             outside_dir.normalize();
21             Ray ousideRay(...);
22             color = transparency*rayTraceRecursive(ousideRay,..);
```

```
23          }
24      }
25  }
```

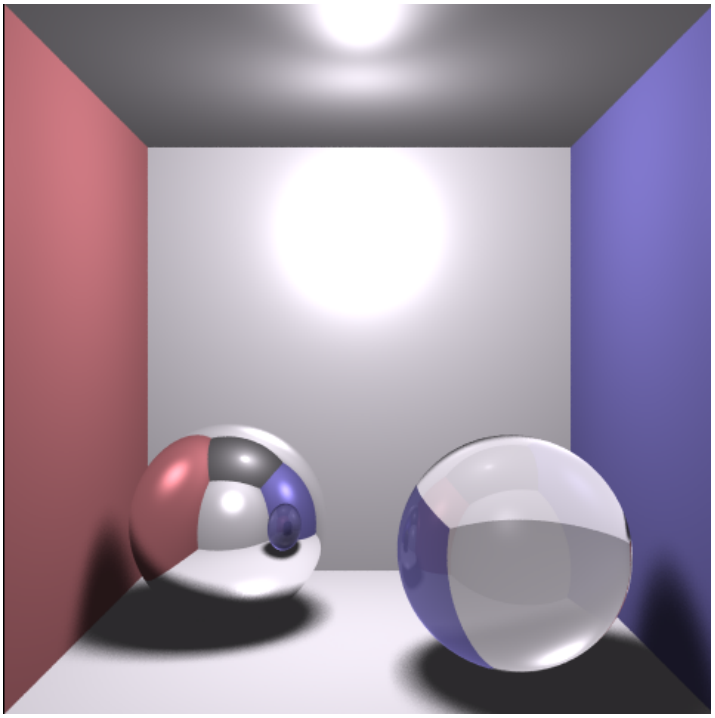

FIGURE 4 – transparent sphere

4 KD-TREE

Le **Kd-tree** peut accélérer le chargement des modèles meshes. Il divise le modèle en plusieurs nœuds (finalement nœuds de feuille) par la répartition spatiale des triangles. Un rayon n'a besoin de calculer que les points d'intersection avec les triangles dans les nœuds feuilles. Nous utilisons la méthode **SAH** pour calculer la position du split_plane et l'axe correspondant.

Listing 7 – 1

```
1  struct KdTreeNode{
2      // information of each node
3  };
4  struct Event{
5      // To determinate the left or right side
6  };
7  recBuild_SAH(){
8      if(achieve to leaf node condition){
9          enregistre the leaf node;
10     }
11     calculate the bounding-box coord;
12
13     for(each axis){
14         calculate the best postion of split plane;
15         // scanne each event, and calculate the cost
16         // cost = S_left/S_total*NBtri_l
```

```
                +S_right/S_total*NBtri_r
            pick the best split plane;
        }
        separate the left and right triangles by split plane;

        enregistre the information(axis,splitPosition,...);

        //recursive the children nodes
        leftLeaf = recBuild_SAH(left_index,depth+1,cpt);
        rightLeaf = recBuild_SAH(right_index,depth+1,cpt);
}
traverse () {
        if (isLeaf)
            return Triangle_intersect(...);
        t = (splitPos - origin[axis]) / direction[axis];
        if (t<= t_min)
            return traverse(with the farNode);
        else if (t >= t_max)
            return traverse (with the nearNode);
        else{
            t_hit = traverse (..., t_min, t);//nearnode
            if (t hit <= t) return t_hit;
            return traverse (...,t, t _max);//farnode
        }
}
```

Nous pouvons afficher le plan de split.

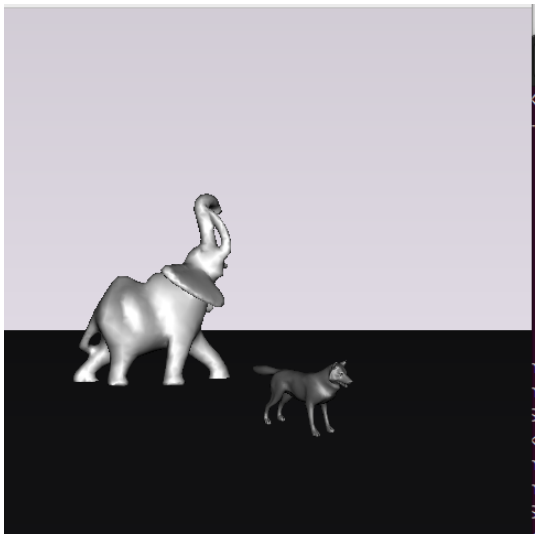

FIGURE 5 – without split plane

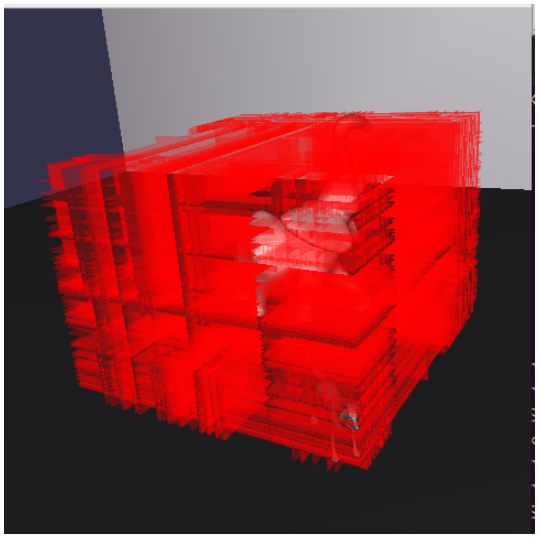

FIGURE 6 – show split planes

Comparer le temps d'exécution avec et sans l'utilisation de KdTree.

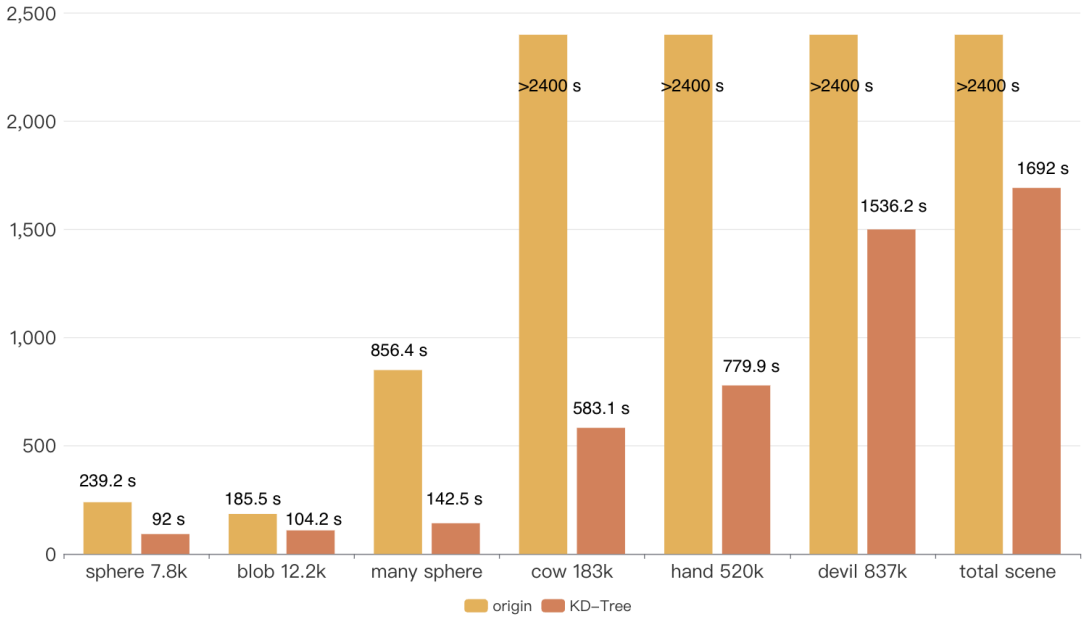

FIGURE 7 – statistic of speed

4.1 Problème restant

Même si la structure actuelle du KdTree permet d'accélérer le rendu des modèles mesh d'environ 2 fois, le temps nécessaire reste très long (plusieurs dizaines de minutes) pour des modèles complexes. Certaines triangles se trouvent sur le plan de split. Dans ce cas, il est nécessaire de les inclure à la fois dans le nœud gauche et le nœud droit. Le plus de noeuds generes, le plus de triangles augmente, ce qui rend les calculs d'intersection plus complexes.

* Une solution naïve est : comparer la position du centre des triangles avec le plan split. Les triangles ne seront pas dupliqués. Mais, le modèle du rendu n'est pas complet. Parce-que ces certains triangles ne sont pas dans les deux nœuds, ce qui genere des parties manquantes dans le rendu.

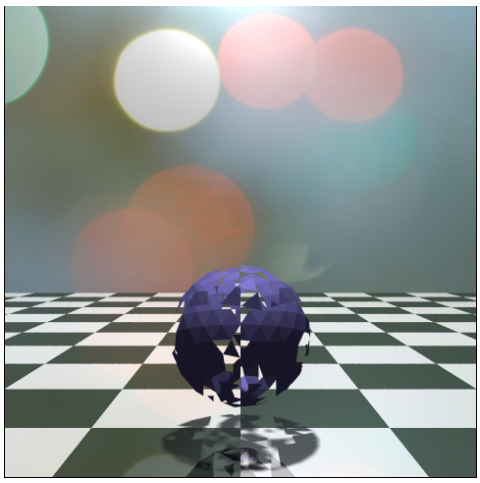

FIGURE 8 – "broken" sphere

* Une autre est : couper ces triangles en deux parties (gauche et droite). Mais il est possible de générer des quadrilatères, ce qui rendra le problème encore plus complexe !

5 Les autres effets

5.1 Texture

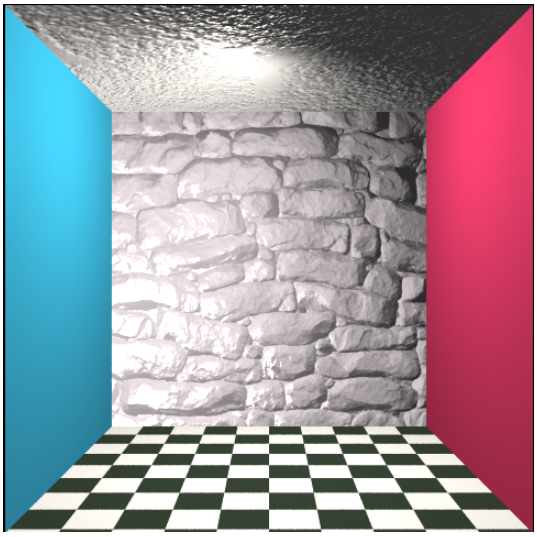

FIGURE 9 – plane texture

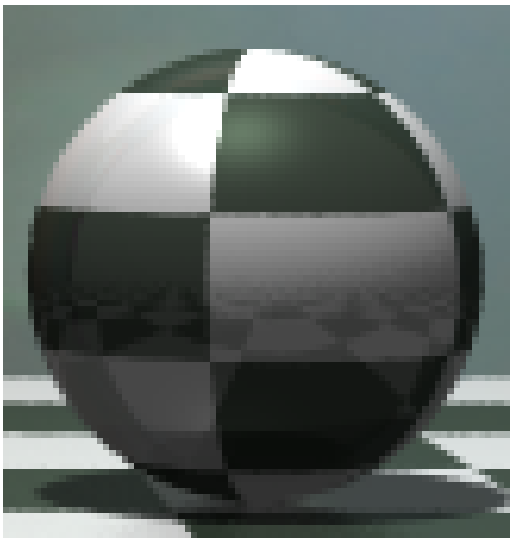

FIGURE 10 – sphere texture

5.2 Fuzzy metal

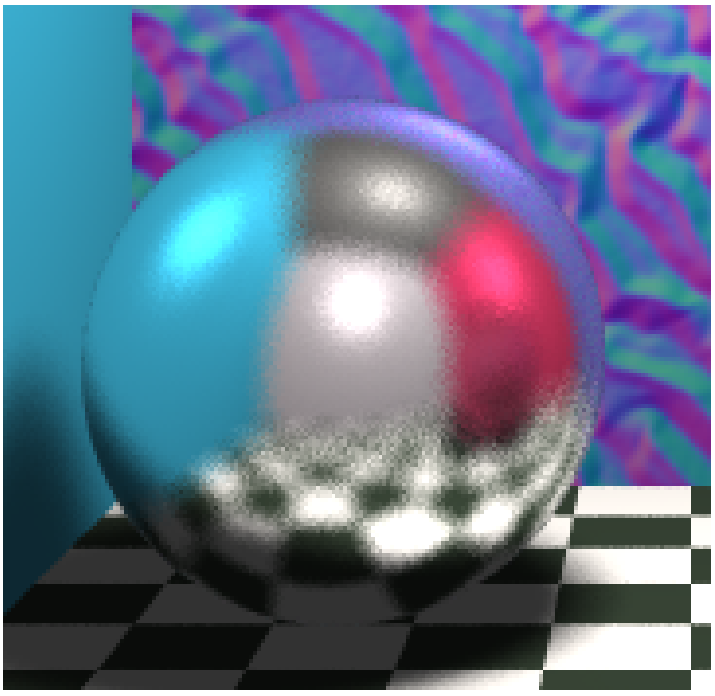

FIGURE 11 – fuzzy metal sphere

6 Final

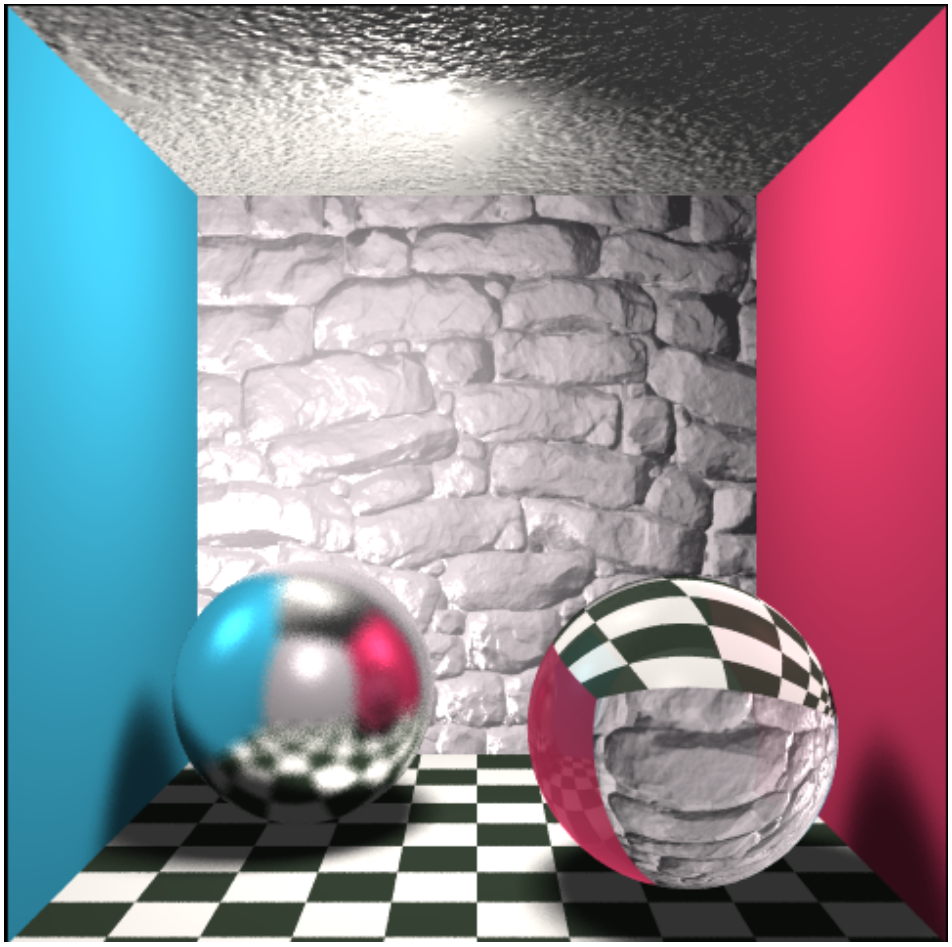

FIGURE 12 – final cornellbox

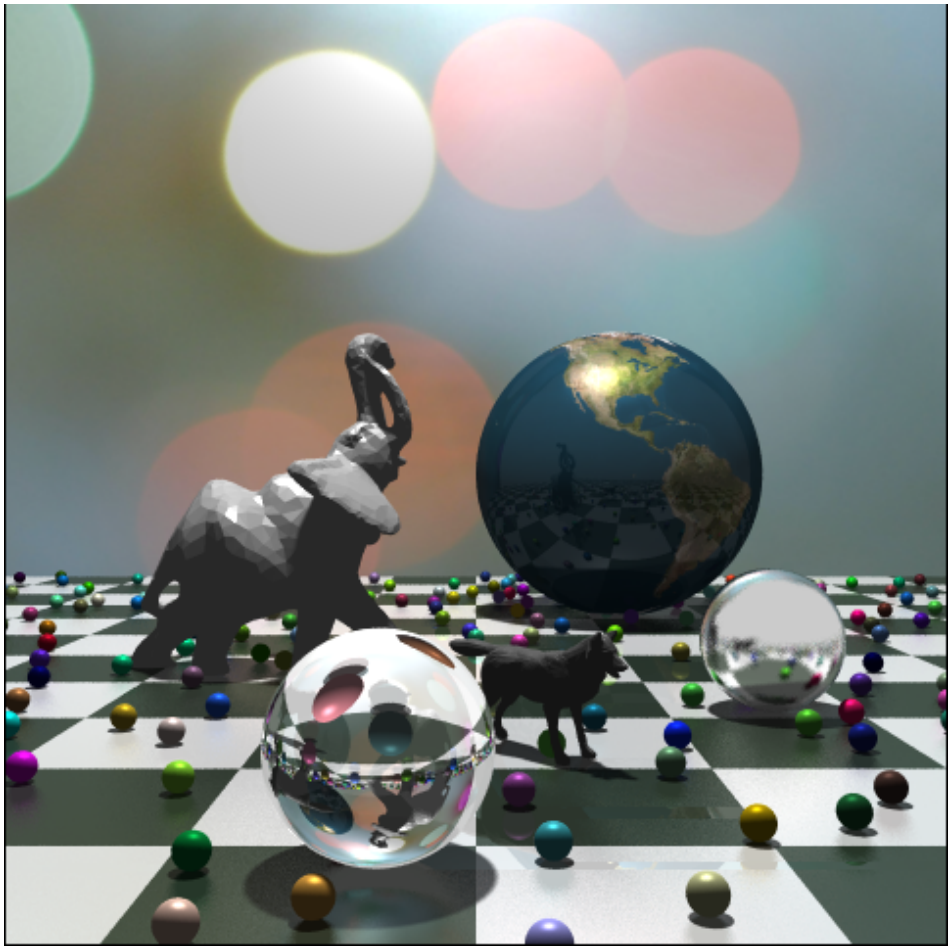

FIGURE 13 – final scene