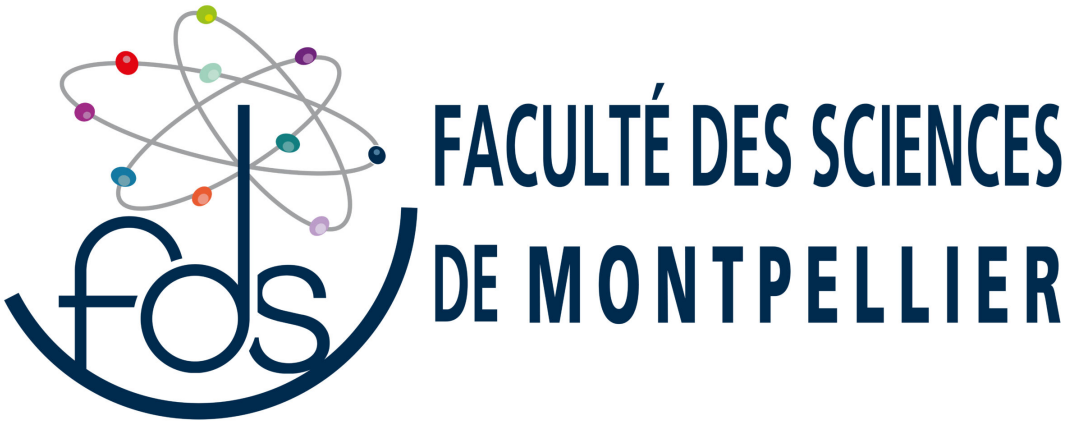
Université de Montpellier

M1 - IMAGINE - Programmation 3D
Compte Rendu - RayTracing phase 2

Etudiant :
Guillaume Bataille

Encadrant :
Noura Faraj
Marc HARTLEY

Année 2022-2023



Sommaire

1. Contexte et Objectif

Afin de pouvoir réaliser ce projet de rayTracing qui va nous permettre de rendre une image via des lancer de rayons, nous allons avoir 3 phases.
Dans cette premiere seconde phase, nous allons donner la notion de 3D a notre rendu via le modèle de phong ! On essayera aussi de créer des ombres dures et smooth

2. Visuels

Voila des visuels pour montrer ou j'en suis (fin phase 2 debut phase 3) mais je vous invite a regarder mes nombreux fails dans le repertoire de mon projet.

FIGURE 2.1 – Phong + Hard Shadow

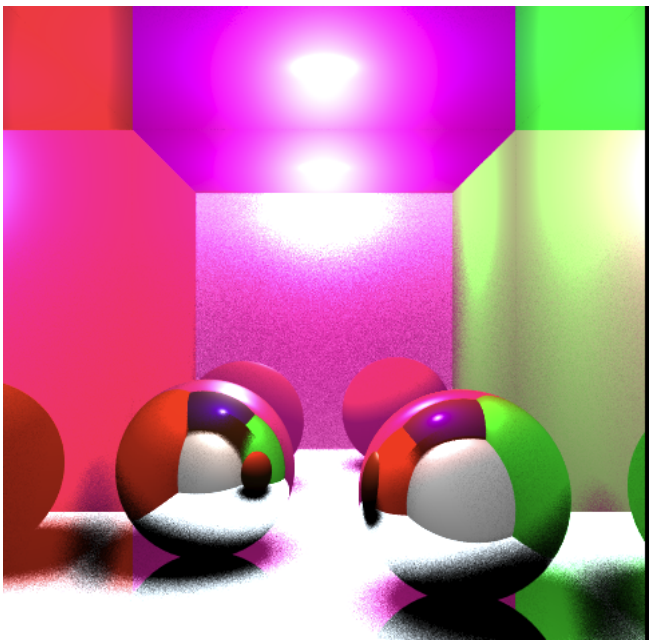


FIGURE 2.2 – Bounce number = 1

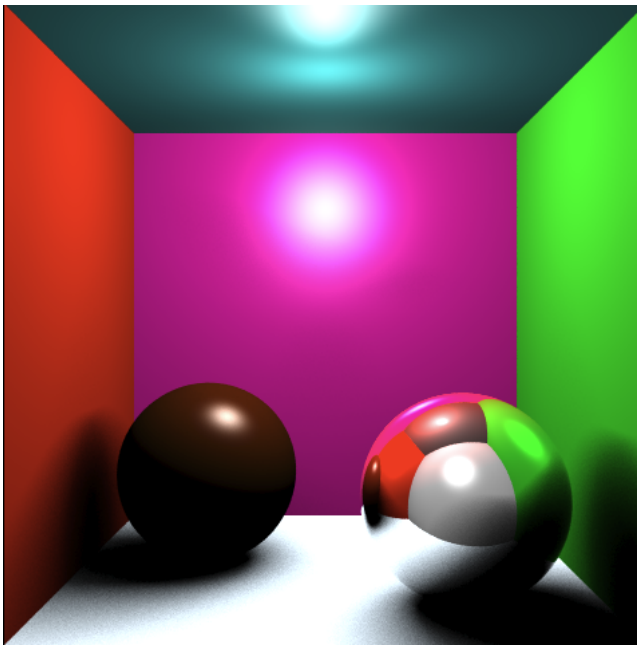


FIGURE 2.3 – Phong + softshadow + mirrorball

3. Phong

Je ne vais pas faire l'affront de résumer phong ici.

```
void  phong(RaySceneIntersection RSI, Vec3 &color, Material mat, Ray &ray){
   for (unsigned long int i = 0; i < lights.size(); i++)
       {
           Vec3 L = lights[i].pos - RSI.position; // Le vecteur depuis le point d'intersection vers la lumière
           L.normalize();
           //DIFFUSE
             double lightvalue = Vec3::dot(L, RSI.normal); // On le recup
               if (lightvalue <= 0) continue;
           Vec3 diffuse = Vec3::compProduct(lights[i].material, mat.diffuse_material) * lightvalue;
           //SPECULAR
           Vec3 R = 2 * (Vec3::dot(RSI.normal, L) * RSI.normal) - L;
           Vec3 V = ray.origin() - RSI.position;
           R.normalize();
           V.normalize();
           Vec3 specular = Vec3::compProduct(lights[i].material,mat.specular_material * pow(Vec3::dot(R,V),mat.
           color = diffuse + specular;
}color += mat.ambient_material;
}
```

4. Shadow

J'utilise ma fonction shadow pour calculer l'ombre dans ma scène. J'ai deux types d'ombres : la dure qui est appelée si le type de lumière est sphérique, et la douce si la lumière est de type quad. Le seul principe a retenir est qu'on va partir du point d'intersection et viser le point de lumière courant.Il suffira juste de verifier si on arrive ou non a atteindre la lumière via computeintersect et t.

```
void shadow(RaySceneIntersection RSI, Vec3 &color, Material mat)
 {
     for (unsigned long int i = 0; i < lights.size(); i++)
     {
         Vec3 L = lights[i].pos - RSI.position; // Le vecteur depuis le point d'intersection vers la lumière
         double range = L.length();             // La distance entre la position et la lumière
         L.normalize();
         double shadowvalue = 0;           // Valeur de l'ombre courante (0 si obstacle et 1 sinon)
         Ray light = Ray(RSI.position, L); // origine la position de l'intersect et direction la light

         // OMBRES DURES - cas avec type of light spherical / ponctuel
         if (lights[i].type == LightType_Spherical)
         {   RaySceneIntersection inter2 = computeIntersection(light, 0.0001);
             if (inter2.t >= range)
             {
                 shadowvalue = 1; // Le 0.001 sur le znear me sert a ne pas considérer comme obstacle un vert
                 // Pas besoin de shadowvalue car ici l'ombre est nette, soit 0 soit 1;
             }
         }
         else
         {
             // OMBRES DOUCES
             // création du square de lumière a partir du mesh ajouté dans l'init de la scène de cornell
             Vec3 bottom_left = lights[i].quad.vertices[0].position;
             Vec3 right = lights[i].quad.vertices[1].position - bottom_left;
             Vec3 up = lights[i].quad.vertices[3].position - bottom_left;


             for (unsigned long int j = 0; j < ECHANTILLONAGE_LIGHT; j++)
             {
             Vec3 randx = (double(rand()) / RAND_MAX * right);
             Vec3 randy = (double(rand()) / RAND_MAX * up);
             Vec3 randpoint = randx + randy + bottom_left;

             Vec3 vec_vers_randpoint = randpoint - RSI.position;
             double rangebis = vec_vers_randpoint.norm();
             vec_vers_randpoint.normalize();
             light = Ray(RSI.position, vec_vers_randpoint);
             RaySceneIntersection inter3 = computeIntersection(light,0.0001);
```

```
40                  if(!(inter3.intersectionExists && inter3.t < range && inter3.t > 0.0001)){
41                      shadowvalue ++;
42                  }
43                  }
44                  shadowvalue /= ECHANTILLONAGE_LIGHT; // Recupère le pourcentage de light qui est passé en divisa
45                   color *= shadowvalue/lights.size(); // On pondère l'intensité lumineuse (ambiante + diffuse + s
46              }
47          }
48
49      }
```

5. Equation de Bounce

Importante pour Raytracerecursive, je la note ici :Vect_reflechi = Vecteur_incident - 2 * normale *dot(Vecteur_incident,normale)

6. La fonction RayTraceRecursive

RayTraceRecursive agit comme raytrace mais donne la posibilité de gérer des rebonds de rayons pour des effets particuliers (mirroir ou autre) ! On utilise donc phong, shadow pour avoir la couleur lors de la première itération. Et en fonction du nombre de rebond restant, on réitère avec un nouveau rayon créer avec l'équation de bounce et la derniere position intersection.

```
1   Vec3 rayTraceRecursive(Ray ray, int NRemainingBounces, double znear)
2    {
3
4        // TODO appeler la fonction recursive
5        Vec3 color;
6        RaySceneIntersection RSI = computeIntersection(ray, znear);
7        if (!RSI.intersectionExists) // Si y'a pas d'intersection
8            return Vec3(0, 0, 0);
9
10       Material mat = getRayMaterial(RSI); // On recupère le material courant
11       color = mat.ambient_material; // Récup couleur ambiant
12       phong(RSI, color, mat, ray); // On appelle phong pour y ajouter la couleur de l'intersection (diffuse +
13       shadow(RSI,color,mat);
14       //computeSoftShadow(color, RSI, ECHANTILLONAGE_LIGHT);
15       if (NRemainingBounces == 0) { //Si y'a plus de rebond
16           return color;
17       }
18       // Parametrage du nouveau point de depart du rayon
19       Vec3 new_origin = RSI.position; //La position de l'intersection devient l'origine du rayon
20       Vec3 new_direction = RSI.bounce_direction ; // La direction bounce (reflexion via old direction et norma
21       Ray Ray_bounce = Ray(new_origin,new_direction);
22       //TODO RaySceneIntersection raySceneIntersection = computeIntersection(ray);
23       color += rayTraceRecursive(Ray_bounce,NRemainingBounces-1, 0.001); // On retire un rayon en décrémentant
24       // znear =0.001 c'est pour pas qu'il se cogne sur lui même ou sur son voisin
25       return color;
26   }
```

7. Remarques

Pour effectuer le tp nous avons :
- Perdu la tête à force d'avoir des erreurs INCOMPREHENSIBLE
- Ajouter des attributs dans les structures Intersect
- Ajouter des variables gloables define NBBOUNCES, NBSAMPLE, ZNEAR, NBECHANTILLONS ;
- Ajouter une fonction qui facilite de retrouver le material courant d'un RSI.

A retenir :
-Attention aux normalisations de vecteurs
-Tester regulièrement le code
-Faire attentions aux variables useless car ça comment a devenir long les rendus