TP L3 IGTAI - Lancer de rayons

Valentin Roussellet, Mathias Paulin, David Vanderhaeghe

6 février 2020

Introduction

Objectif

L'objectif de ce projet est d'implémenter un rendu par lancer de rayon (*ray-tracing*).

La génération de l'image de synthèse se fait en calculant l'intersection de rayons avec la surface des objets de la scène 3D, décrits par leur géométrie et leur matériau. À chaque point d'intersection, on fait une simulation d'éclairage dont le résultat est une couleur qui part dans la direction du rayon.

```
// Les codes d'exemple sont mis en forme comme ceci.
```

Le travail attendu est mis en valeur comme cela.

Votre projet est strictement personnel ! Vous ne devez pas vous transmettre de code et toute copie sera sévèrement sanctionnée. Vous pouvez *discuter* entre vous des différentes approches, mais PAS DE TRANSFERT DE CODE. Nous utilisons des outils avancés de détection de plagiat.

Si votre programme ne compile pas, la note sera de 0. Barème :

Version basique 12pts
- rapport **2pts**
- intersection sphère et plan **4pts**
- intersection scène **1pt**
- matériaux basiques **3pts**
- ombres **2pts**

Version avancée 16pts
- matériaux avancés **4pts**

Version étendue 20pts, en choisissant les extensions que vous souhaitez
- kd-tree **3pts**
- anti-aliasing **1pt**
- cylindre, triangle, cône ... **1pt par type d'objet**
- texture **2pts**
- faites nous rêver **2pts**

Vous devez compter une vingtaine d'heures de travail pour ce projet, en dehors des deux séances de TP.

Vous devez remettre un rapport avec votre projet, détaillant ce que vous avez implémenté, les problèmes rencontrés, et montrant les images que vous avez produites. Le dépôt du projet se fait sur Moodle, sous la forme d'un fichier .tar.gz ou .zip contenant votre code source, les images, et votre rapport. **Les fichiers objets et exécutables ne sont pas à remettre !**

Code

Il vous est fourni une base de code en C et C++ qu'il vous faudra compléter pour obtenir un rendu. Vous devez coder en C (ou en C++ si vous le souhaitez). Un Makefile est fourni pour la compilation, si vous n'utilisez pas `gcc` vous devrez sans doute enlever l'option `-fopenmp` des options de compilation. Le Makefile (testé sous linux) gère automatiquement les dépendances entre fichiers, les fichiers .d générés sont propres à un poste de travail (et ne doivent donc pas être copiés sur un autre poste). Libre à vous d'utiliser un autre système d'exploitation/compilation, mais assurez-vous que votre projet compile sur les machines de TP.

Une configuration cmake https://cmake.org/ est aussi fournie. cmake facilite la gestion des dépendances.

Pour compiler votre projet : soit make dans le répertoire de votre projet, soit

```
mkdir build
cd build
cmake ../
make
```

Deux bibliothèques (fournies dans l'archive) sont utilisées pour ce projet :

GLM est une bibliothèque de calcul vectoriel et matriciel. Les points et les vecteurs de l'espace (de dimension 3) sont représentés par des instances de la structure **`vec3`** dont les membres x,y et z correspondent aux trois coordonnées. GLM fournit toutes les opérations sur vecteurs : addition, soustraction, normalisation, produit scalaire et vectoriel, etc. Les points sont représentés par la structure **`point3`** dont les membres x, y et z correspondent également aux trois coordonnées du point dans l'espace. On utilise également un vecteur de dimension 3 pour représenter la couleur par ses 3 composantes r,g et b avec la structure **`color3`**.

Les opérateurs de bases comme $+$ et $-$ s'utilisent directement entre des variables de type **`vec3`**, c'est l'un des apports du C++. Voici quelques exemples de fonctions GLM :

```
vec3 a;
vec3 b;
a.x = 1.f;
a.yz = vec2(0.f, 1.f);
b = 2.f * a;
float d = dot(a, b);
vec3 c = cross(a, b);
vec3 r = reflect(a, b);
```

GLM suit la spécification de GLSL que vous trouverez ici : https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.1.20.pdf

lodepng est une bibliothèque qui permet d'exporter des images au format PNG, ce qui nous permettra de visualiser les résultats de notre lancer de rayon.

Structures de base

Voici un aperçu des fichiers qui vous sont fournis :

image définit la structure image et les fonctions de sauvegarde. Rien à modifier ici.

scene_types définit les principaux types d'objets dans une scène (caméra, éclairage, sphères et plans). Rien à modifier ici.

scene définit une scène et comment y ajouter des objets. Rien à modifier ici.

ray définit la structure représentant un rayon et les fonctions associées. Rien à modifier ici.

raytracer définit la structure représentant une intersection et les fonctions pour calculer les intersections. C'est là que vous devez implémenter les fonctionnalités demandées.

main le programme principal, avec la définition des scènes. Vous pouvez ajouter de nouvelles scènes dans ce fichier.

Ray Un rayon est constitué d'une origine et d'une direction. De plus, un rayon stocke les paramètres t_{min} et t_{max} qui sont les paramètres minimum et maximum que l'on considère sur le rayon. t_{max} sert lorsque l'on cherche l'intersection la plus proche, ou lorsque l'on veut restreindre la recherche d'une intersection à un segment.

Un rayon est donc défini par

$$r(t) = \mathbf{O} + t\,\overrightarrow{\mathbf{d}} \quad t \in [0, +\infty]$$

La structure fournie en C est la suivante :

```
typedef struct ray_s{
  point3 orig;
  vec3 dir;
  float tmax;
  float tmin;
  int depth;
} Ray;
```

L'entier `int depth` sert à stocker le nombre de "rebonds" fait par un rayon, pour arrêter la récursion si besoin.

Geometry Cette structure représente la géométrie d'un objet de la scène. Ici on utilise une `union` pour pouvoir représenter des sphères et des plans.

```
typedef struct geometry_s {
Etype type;
   union {
      struct {
         vec3 center;
         float radius;
      } sphere;
      struct {
         vec3 normal;
         float dist;
      } plane;
   };
} Geometry;
```

Le membre `type` nous permet de savoir s'il s'agit d'une sphère ou d'un plan, grâce aux constantes SPHERE et PLANE définies dans `scene.h` ce qui permettra d'accéder aux bons membres de l'union, selon le type de l'objet.

Intersection Une intersection contient le point d'intersection et la normale à la surface en ce point, ainsi qu'un pointeur vers le matériau de l'objet intersecté. Ce sont toutes les informations dont on a besoin pour calculer la couleur émise par ce rayon.

```
typedef struct intersection_s {
  vec3   normal;
  point3 position;
  Material *mat;
} Intersection;
```

Object Cette structure représente un objet de la scène, c'est-à-dire une géométrie et un matériau.

```
typedef struct object_s {
   Geometry geom;
   Material mat;
} Object;
```

Scene La scène contient une caméra, une couleur de ciel, une liste d'objets, une liste de lumières (il s'agit de `std::vector<>` C++). Le parcours des éléments d'une liste s'effectue comme sur un tableau, avec l'opérateur `[]`.

```
typedef struct scene_s {
  Lights lights; //! the scene have several lights
  Objects objects; //! the scene have several objects
  Camera cam; //! the scene have one camera
  color3 skyColor; //! the sky color, could be extended to a sky
     function ;)
} Scene;


// exemple d'acces a la liste objects :
  size_t objectCount = scene->objects.size();
  for(size_t i=0; i<objectCount; i++) {
     scene->objects[i]; // ne fait rien en dehors d'acceder a l'objet
        ;)
```

Chapitre 1

Intersections et éclairage simplifié

Dans cette partie, nous allons mettre en place la structure de base du lancer de rayons avec un modèle simple d'éclairage.

1.1 Calcul d'intersection

Nos scènes 3D seront composées de deux types d'objets : des plans et des sphères. Il nous faut donc écrire les fonctions capables de déterminer s'il y a intersection ou non, et si c'est le cas, calculer la position du point d'intersection.

Il vous faut donc implémenter les fonctions suivantes, définies dans le fichier `raytracer.cpp` :

```
bool intersectPlane (Ray *ray, Intersection *intersection, Object *
   plane);
bool intersectSphere(Ray *ray, Intersection *intersection, Object *
   sphere);
```

Ces fonctions doivent suivre la même spécification :

- Calculer l'intersection entre le rayon et l'objet.
- S'il n'y a pas d'intersection, renvoyer **`false`**.
- S'il y a une intersection, calculer la valeur du paramètre t correspondant au point d'intersection.
- Vérifier si t est entre t_{min} et t_{max} pour le rayon
- Si ce n'est pas le cas, on renvoie **`false`**.
- Sinon, mettre à jour les paramètres de la structure `Intersection` et de `Ray` et renvoyer **`true`**.

1.1.1 Intersection rayon sphère

Un rayon est caractérisé par son origine $\mathbf{O}$ et sa direction $\overrightarrow{\mathbf{d}}$ (correspondant aux membres `ray.origin` et `ray.dir` de la structure `Ray`). Les points appartenant au rayon s'écrivent donc tous comme $\mathbf{P} = \mathbf{O} + t\overrightarrow{\mathbf{d}}$.

Une sphère est caractérisée par son centre $\mathbf{C}$ et son rayon R (correspondant aux membres `sphere.center` et `sphere.radius` de la structure `Geometry`). Les points appartenant à la sphère sont ceux qui sont à une distance R du centre, soit $\|\mathbf{P} - \mathbf{C}\| = R$.

Il y a intersection s'il y a au moins un point du rayon qui se trouve sur la sphère, c'est-à-dire

qui satisfait les deux équations précédentes :

$$\begin{cases} \mathbf{P} = \mathbf{O} + t\,\overrightarrow{\mathbf{d}} \\ \|\mathbf{P} - \mathbf{C}\| = R \end{cases}$$

On remplace $\mathbf{P}$ dans la deuxième équation ce qui nous donne une seule équation dont l'inconnue est t.

$$\|(\mathbf{O} + t\,\overrightarrow{\mathbf{d}}) - \mathbf{C}\| = R$$

Pour simplifier cette équation, on élève chaque côté au carré, et on utilise le fait que pour tout vecteur, $\|\overrightarrow{\mathbf{v}}\|^2 = \overrightarrow{\mathbf{v}} \cdot \overrightarrow{\mathbf{v}}$.

$$((\mathbf{O} + t\,\overrightarrow{\mathbf{d}}) - \mathbf{C}) \cdot ((\mathbf{O} + t\,\overrightarrow{\mathbf{d}}) - \mathbf{C}) = R^2$$

En développant, on trouve une équation du second degré en t :

$$t^2 + 2t(\overrightarrow{\mathbf{d}} \cdot (\mathbf{O} - \mathbf{C})) + ((\mathbf{O} - \mathbf{C}) \cdot (\mathbf{O} - \mathbf{C}) - R^2) = 0$$

Cette équation peut avoir 0 solution (pas d'intersection), une solution (dans le cas où le rayon est tangent à la sphère) ou deux solutions. Dans le cas où il y a deux solutions on gardera le plus petit t compris entre t_{min} et t_{max}, qui permettra d'obtenir le point d'intersection le plus près de l'origine du rayon.

La normale à une sphère en un point $\mathbf{P}$ est toujours dirigée selon le vecteur $\overrightarrow{\mathbf{CP}}$ reliant le centre de la sphère et le point $\mathbf{P}$. N'oubliez pas de normaliser le résultat.

1.1.2 Intersection rayon plan

Le plan est défini par une normale $\overrightarrow{\mathbf{n}}$ et une distance à l'origine D. Un point $\mathbf{P}$ appartenant au plan vérifie l'équation $\mathbf{P} \cdot \overrightarrow{\mathbf{n}} + D = 0$. Ici on utilise un raccourcis de notation puisqu'on écrit le produit scalaire $\mathbf{P} \cdot \overrightarrow{\mathbf{n}}$ entre un point et un vecteur. En effet, le point $\mathbf{P}$ de coordonnées P_x, P_y, P_z peut être naturellement identifié au vecteur $\overrightarrow{\mathbf{P}} = (P_x, P_y, P_z)^T$. On a donc de nouveau un système dans lequel on substitue $\mathbf{P}$ pour trouver une équation en t :

$$(\mathbf{O} + t\,\overrightarrow{\mathbf{d}}) \cdot \overrightarrow{\mathbf{n}} + D = 0$$

Qui a pour solution :

$$t = -\frac{\mathbf{O} \cdot \overrightarrow{\mathbf{n}} + D}{\overrightarrow{\mathbf{d}} \cdot \overrightarrow{\mathbf{n}}}$$

sauf si $\overrightarrow{\mathbf{d}} \cdot \overrightarrow{\mathbf{n}} = 0$, dans ce cas il n'y a pas de solution (le rayon étant parallèle au plan).

La normale à un plan est constante et partout égale à $\overrightarrow{\mathbf{n}}$.

1.1.3 Tests unitaires

Les fonctions sont validées par des tests unitaires. Ces tests sont définis dans le fichier `unit-test.cpp`.

Pour vérifier votre implémentation, compilez les tests avec

`make unit-test`

Si votre implémentation est correcte, vous devriez obtenir l'affichage suivant en exécutant `./unit-test`

```
r0 to sphere1 : [OK]
r0 to sphere2 : [OK]
r0 to plane1 : [OK]
r0 to plane2 : [OK]
r1 to sphere1 : [OK]
r1 to sphere2 : [OK]
r1 to plane1 : [OK]
r1 to plane2 : [OK]
r2 to sphere1 : [OK]
r2 to sphere2 : [OK]
r2 to plane1 : [OK]
r2 to plane2 : [OK]
```

Codez les fonctions : `intersectPlane` et `intersectSphere`, vérifiez votre implémentation avec `./unit-test`

1.1.4 Test du lancer de rayons

Pour obtenir un premier rendu, il faut implémenter la dernière fonction manquante dans `raytracer.cpp`

```
bool intersectScene(const Scene *scene, Ray *ray, Intersection *
    intersection );
```

Cette fonction est appelée pour chaque rayon : son but est de vérifier s'il y a une intersection entre le rayon et chaque objet de la scène. La fonction doit renvoyer **`true`** s'il y a eu au moins une intersection et **`false`** sinon.

Pour l'instant les objets n'ont pas de couleur : la couleur de l'objet est calculée directement à partir de sa normale, ce qui vous permet de vérifier visuellement si votre calcul de normale est correct.

Enfin, vous devez implémenter la fonction `trace_ray` de `raytracer.cpp` qui doit renvoyer la couleur du pixel correspondant au rayon donné en argument (pour l'instant on ignore le paramètre `tree`).

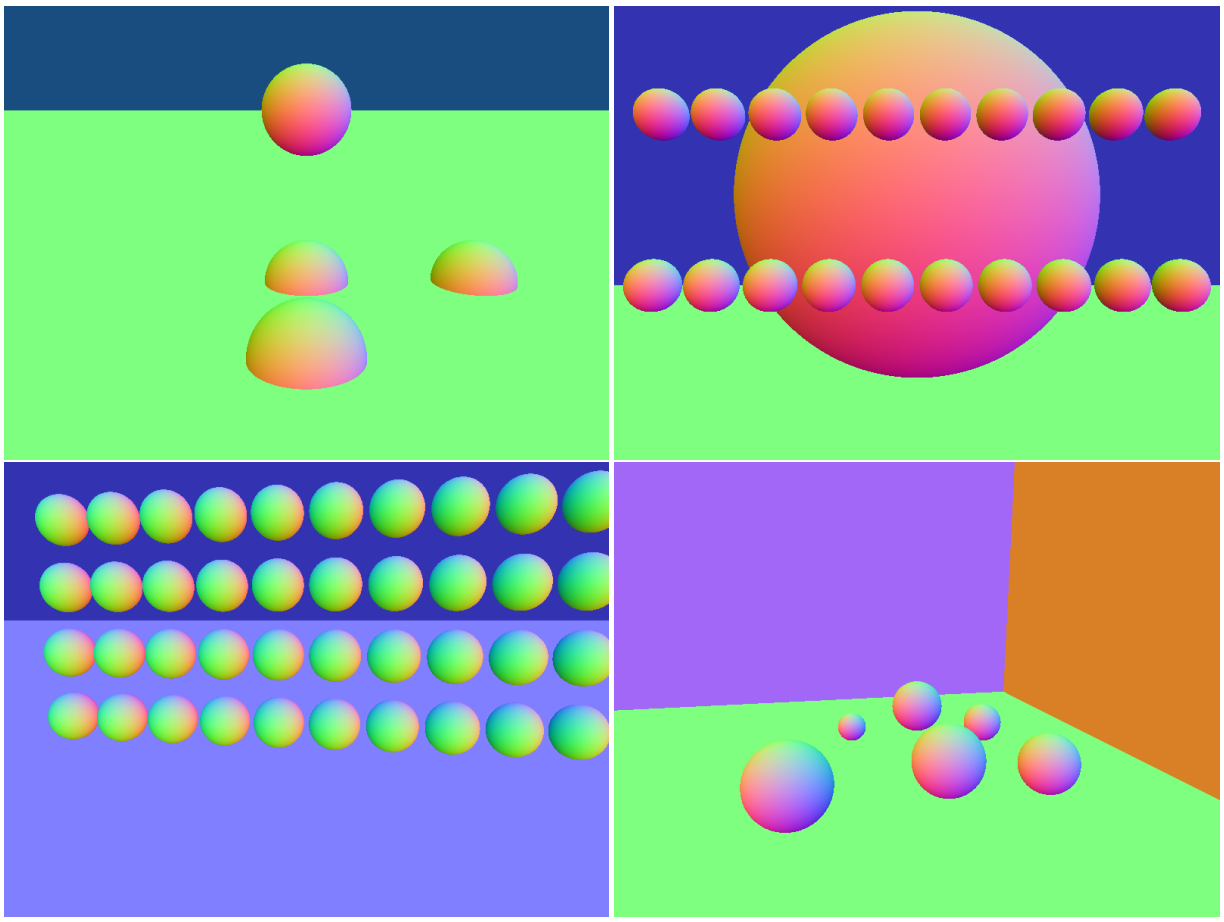
Cette fonction devra implémenter la logique suivante :

- Appeler `intersectScene()` sur le rayon considéré.
- S'il n'y a aucune intersection, on renvoie la couleur du "ciel" accessible dans `scene->skyColor`.
- S'il y a intersection, on renvoie une couleur calculée à partir des coordonnées de la normale de l'intersection.

$$c = 0.5\vec{\mathbf{n}} + 0.5$$

En compilant le raytracer avec `make mrt`, vous pouvez tester le résultat sur les différentes scènes prédéfinies.

Voici les résultats que vous devriez obtenir :



Codez la fonction `intersectScene`

1.2 Couleur et *shading*

Dans cette section, nous allons ajouter de la couleur aux objets dessinés. La couleur d'un objet en un point dépend du matériau de l'objet, de la normale à la surface (notée $\overrightarrow{\mathbf{n}}$), de la direction d'éclairage (notée $\overrightarrow{\mathbf{l}}$), de la direction d'observation (notée $\overrightarrow{\mathbf{v}}$) et de la couleur de la lumière lc. Ce calcul est représenté par la fonction `shade` de `raytracer.cpp`

Dans un premier temps, nous allons mettre en place un modèle simplifié de réflexion de la lumière où l'on ne prendra en compte que la composante *diffuse*, qui ne dépendra pas de $\overrightarrow{\mathbf{v}}$.

Pour cela, il faudra d'une part changer la fonction `trace_ray()` pour qu'elle appelle `shade()` avec les bons paramètres, et d'autre part implémenter le shading dans cette dernière.

À la place du calcul de couleur basé sur la normale, la fonction `trace_ray` devra appeler la fonction `shade` pour chaque source lumineuse. On obtiendra la couleur finale du point en additionnant les contributions de chacune de ces sources. On peut accéder aux sources lumineuses dans le tableau `scene->lights`.

Pour calculer les paramètres de `shade`, on rappelle que :

- La direction de vue $\overrightarrow{\mathbf{v}}$ est l'opposée de la direction du rayon $\overrightarrow{\mathbf{d}}$.
- La direction d'éclairage $\overrightarrow{\mathbf{l}}$ du point $\mathbf{P}$ éclairé par une lumière située en $\mathbf{L}$ est définie par $\overrightarrow{\mathbf{l}} = (L - P)/\|L - P\|$ (vecteur orienté du point vers la lumière).

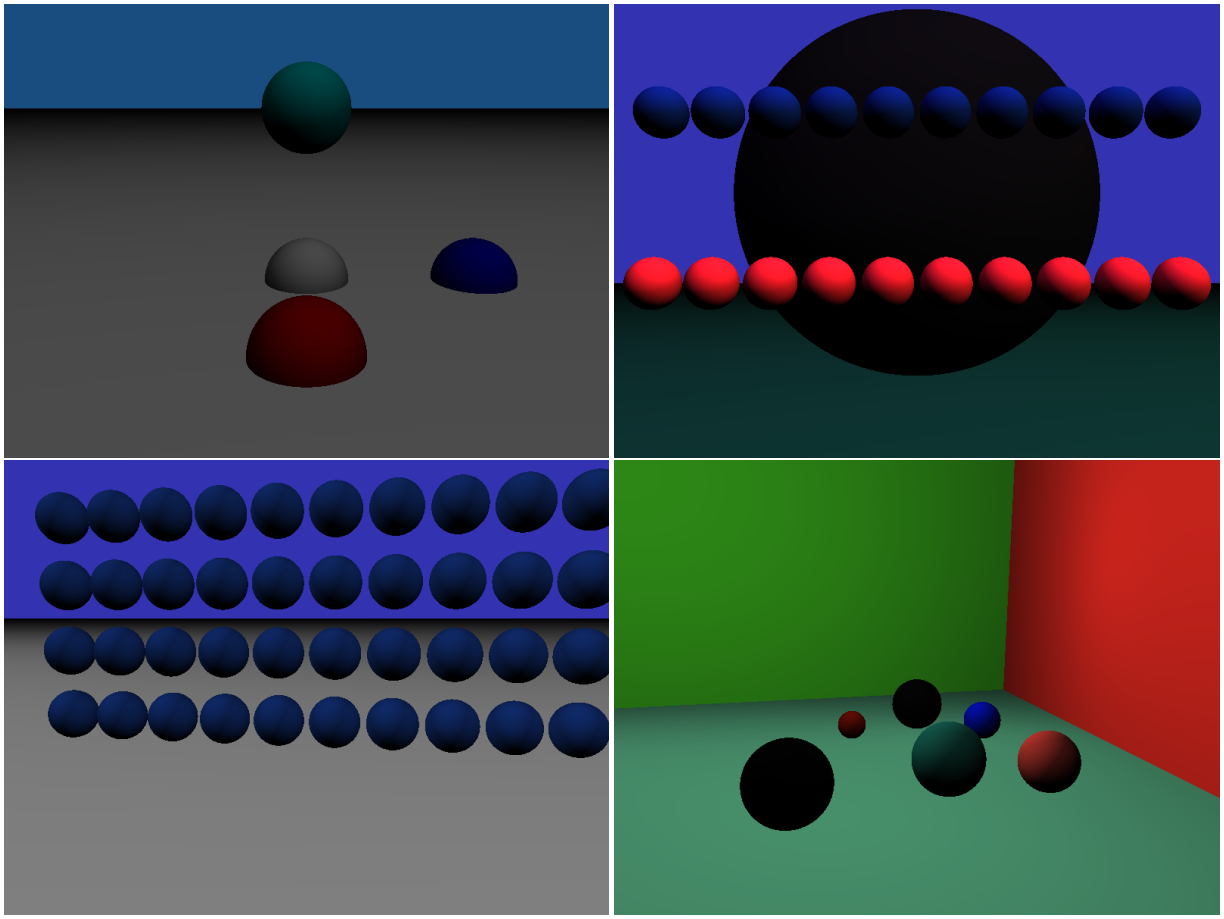
La fonction de shading elle-même est très simple :

$$c = \frac{K_d}{\pi} \times \overrightarrow{\mathbf{l}} \cdot \overrightarrow{\mathbf{n}} \times lc$$

où K_d est la couleur diffuse du matériau `m->diffuseColor`.

La couleur dépend du cosinus de l'angle θ entre $\overrightarrow{\mathbf{l}}$ et $\overrightarrow{\mathbf{n}}$, calculé au moyen du produit scalaire $\cos\theta = \overrightarrow{\mathbf{l}} \cdot \overrightarrow{\mathbf{n}}$. Cependant, le cosinus peut être négatif si $\overrightarrow{\mathbf{l}}$ et $\overrightarrow{\mathbf{n}}$ sont opposés. Cela correspond au cas où la lumière se trouve derrière l'objet. Dans ce cas, on renvoie simplement du noir (et pas une couleur négative !).

Vous devriez obtenir le résultat suivant :



Codez la fonction `shade` et appelez cette fonction pour le calcul de l'éclairage dans `trace_ray`

1.3 Ombres

Dans cette section, nous allons rajouter la prise en compte de l'ombre portée, lorsqu'un objet cache la lumière et projette une ombre sur un autre objet.

Pour tenir compte des ombres, on teste l'éclairage d'un point en envoyant un rayon partant du point considéré vers la source lumineuse. S'il n'y a pas d'intersection entre le point et la lumière, alors le point est éclairé, sinon il est dans l'ombre.

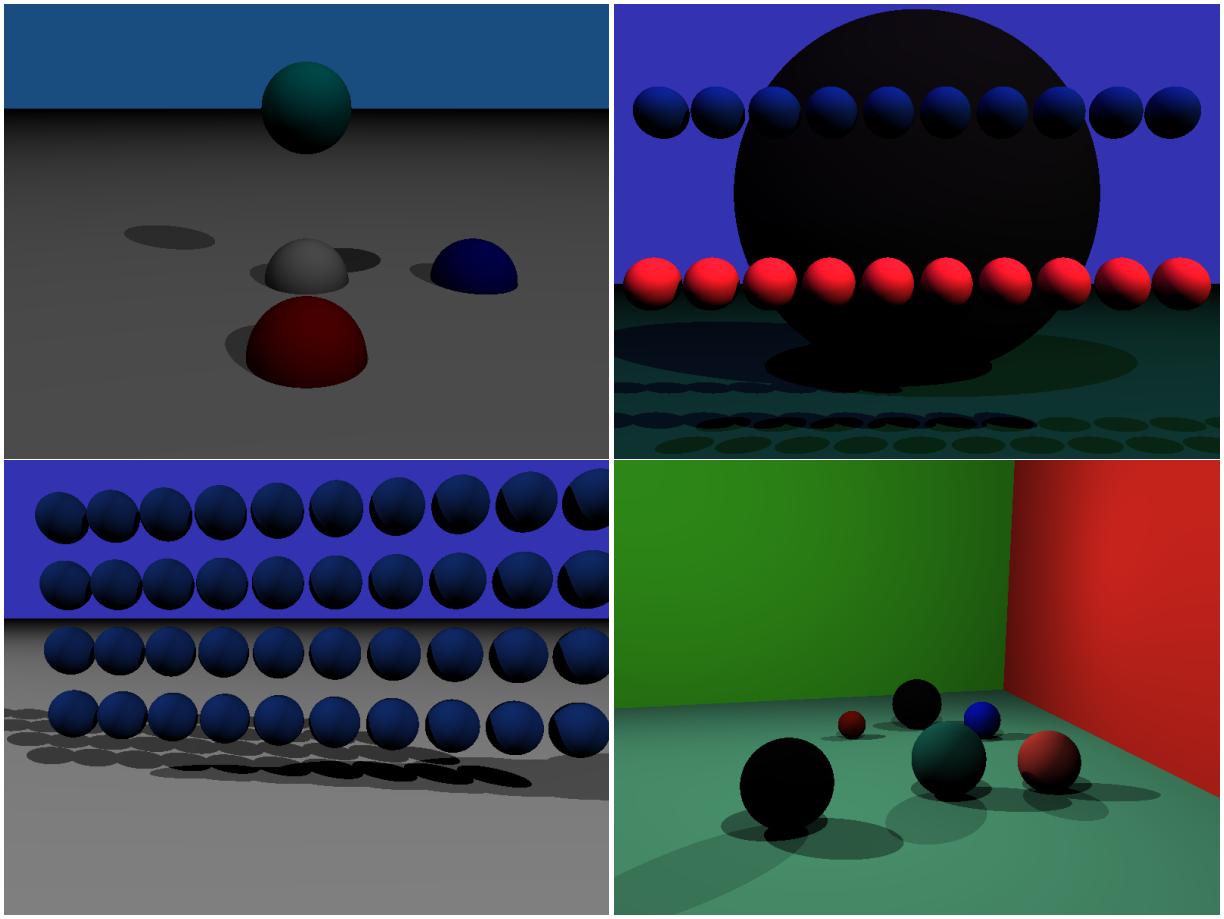
Si on ne prend pas de précautions pour initialiser le point de départ du rayon, on risque d'obtenir de l'*acné* à cause des imprécisions de calcul numérique. Pour éviter ce problème, il faut

décaler le point de départ dans la direction de la lumière pour que la surface de départ ne soit pas en intersection avec le rayon lancé en direction de la lumière.

On va modifier la fonction `trace_ray()` pour rajouter l'ombre portée. La nouvelle fonction devra implémenter l'algorithme suivant :

- Appeler `intersectScene()` sur le rayon considéré.
- S'il n'y a aucune intersection, on renvoie la couleur du "ciel" accessible dans `scene->skyColor`.
- S'il y a intersection, pour chaque source lumineuse :
  - Créer un rayon d'ombre ayant pour point d'origine $\mathbf{P} + \epsilon \overrightarrow{\mathbf{l}}$ et pour direction $\overrightarrow{\mathbf{l}}$, où $\mathbf{P}$ est le point d'intersection et ϵ est une petite constante définie dans le code par `acne_eps`.
  - On appelle la fonction `shade()` seulement si le rayon d'ombre n'intersecte aucun objet dans la scène entre $\mathbf{P}$ et $\mathbf{L}$. Sinon, la contribution de cette source est 0 (noir).
- On somme toutes les contributions de chaque source lumineuse.

Une fois les ombres mises en place, vous devriez obtenir ce résultat :



Modifiez `trace_ray` pour tenir compte des rayons d'ombre

Chapitre 2

Vers un modèle d'éclairage plus réaliste

Dans cette partie, nous allons remplacer le shading basique de la partie précédente par un modèle plus réaliste. Pour cela, nous allons utiliser un modèle décrivant la surface des objets à micro-échelle et permettant de caractériser les effets complexes de réflexion de la lumière. Un tel modèle est appelé modèle de BSDF (pour Bidirectionnal Scattering Distribution Function) et contient un terme diffus similaire au modèle simple développé dans la partie précédente, et un terme représentant la brillance d'un objet. Pour évaluer ce terme de brillance, le modèle que nous utilisons est fondé sur la théorie des micro-facettes. Dans cette théorie, une surface rugueuse est définie comme une distribution aléatoire de micro-facettes parfaitement lisses, sur lesquelles la réflexion de la lumière dépend, comme expliqué en cours, des indices de réfraction des deux milieux séparés par la surface. Pour développer un tel modèle, il faut donc programmer :

- la fonction de distribution des micro-facettes,
- la fonction évaluant le terme de Fresnel,
- une fonction caractérisant les phénomènes d'ombrage et de masquage pouvant survenir sur une surface rugueuse
- et une fonction évaluant le modèle complet de BSDF.

2.1 Distribution de Beckmann et terme de Fresnel

Dans un premier temps, nous allons implémenter les fonctions de distribution des micro-facettes et de calcul du terme de Fresnel.

La figure 2.1 met en évidence les vecteurs (directions) utilisés pour la définition des différentes fonctions.

- $\overrightarrow{l}$: direction d'incidence. Orientée vers la source de lumière.
- $\overrightarrow{v}$: direction de sortie. Direction de vue, opposée de la direction du rayon incident.
- $\overrightarrow{n}$: normale à la macro-surface.
- $\overrightarrow{h}$: bissectrice entre $\overrightarrow{l}$ et $\overrightarrow{v}$ (*half-vector*) : normale à la micro-facette sur laquelle $\overrightarrow{l}$ et $\overrightarrow{v}$ correspondent à des directions de réflexion miroir.

2.1.1 Distribution de Beckmann

Cette fonction représente la distribution des micro-facettes sur un matériau rugueux.

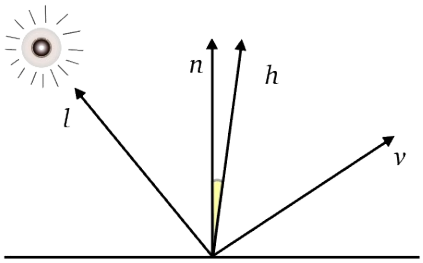


FIGURE 2.1 – Vecteurs pour le calcul de la BSDF.

Elle est définie par la formule

$$D(\theta_h, \alpha) = \chi^+ (\cos(\theta_h)) \frac{\exp\left(\frac{-\tan^2 \theta_h}{\alpha^2}\right)}{\pi \alpha^2 \cos^4(\theta_h)}$$

Avec

$$\chi^+(x) = \begin{cases} 1 & \text{si } x > 0 \\ 0 & \text{sinon} \end{cases}$$

Dans le code cela correspond à la fonction

```
float RDM_Beckmann(float NdotH, float alpha);
```

Ici, `NdotH` est le produit scalaire entre la normale $\overrightarrow{\mathbf{n}}$ et la bissectrice $\overrightarrow{\mathbf{h}}$.

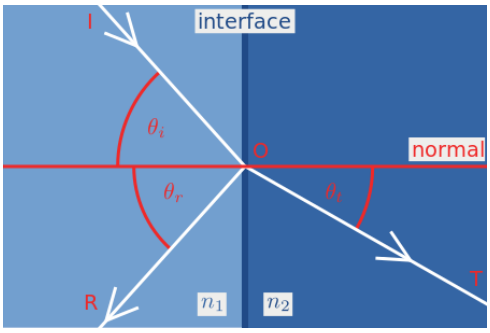
Ces deux vecteurs étant normalisés, on a directement $\cos \theta_h = \overrightarrow{\mathbf{n}} \cdot \overrightarrow{\mathbf{h}}$. Pour le calcul de $\tan^2 \theta_h$ on pourra utiliser la formule de trigonométrie suivante :

$$\tan^2 x = \frac{\sin^2 x}{\cos^2 x} = \frac{1 - \cos^2 x}{\cos^2 x}$$

La rugosité de la surface est définie par le paramètre α représentant l'erreur quadratique moyenne de l'orientation des micro-facettes. En pratique, ce paramètre varie de 0.001 pour une surface très lisse à 0.7 pour une surface très rugueuse.

2.1.2 Terme de Fresnel

Le terme de Fresnel F représente la quantité de lumière réfléchie et dépend des indices de réfraction du matériau extérieur η_1 et intérieur η_2 et des angles entre les rayons incident, réfracté et réfléchi.



On a

$$F = \frac{1}{2}\left(R_s + R_p\right)$$

$$R_s = \frac{\left(\eta_1 \cos\theta_i - \eta_2 \cos\theta_t\right)^2}{\left(\eta_1 \cos\theta_i + \eta_2 \cos\theta_t\right)^2}$$

$$R_p = \frac{\left(\eta_1 \cos\theta_t - \eta_2 \cos\theta_i\right)^2}{\left(\eta_1 \cos\theta_t + \eta_2 \cos\theta_i\right)^2}$$

La fonction correspondante dans le code est

```
float RDM_Fresnel(float LdotH, float extIOR, float intIOR)
```

Où `LdotH` représente le produit scalaire $\overrightarrow{\mathbf{l}} \cdot \overrightarrow{\mathbf{h}}$ entre la direction d'incidence $\overrightarrow{\mathbf{l}}$ et la normale à la micro-surface $\overrightarrow{\mathbf{h}}$, `extIOR` l'indice de réfraction extérieur η_1 (1.0 pour le vide) et `intIOR` l'indice de réfraction intérieur η_2.

Les cosinus des angles se calculent à partir de $\overrightarrow{\mathbf{l}} \cdot \overrightarrow{\mathbf{h}}$. On a directement $\cos\theta_i = \overrightarrow{\mathbf{l}} \cdot \overrightarrow{\mathbf{h}}$

L'angle θ_t se calcule avec la loi de Snell-Descartes pour la réflexion :

$$\eta_1 \sin\theta_i = \eta_2 \sin\theta_t$$

Avec quelques formules de trigonométrie, on a

$$\sin^2\theta_t = \left(\frac{\eta_1}{\eta_2}\right)^2 \left(1 - \cos^2\theta_i\right)$$

Selon la valeur des indices η, on peut potentiellement avoir $\sin^2\theta_t > 1$. Ce cas signifie en fait qu'il n'y a pas de réfraction et que toute la lumière est réfléchie. Dans ce cas la fonction doit renvoyer 1.

Sinon, on utilise encore un peu de formules trigonométriques pour avoir

$$\cos\theta_t = \sqrt{1 - \sin^2\theta_t}$$

et calculer R_s, R_p et finalement F.

Implémentez les fonctions `RDM_Beckmann` et `RDM_Fresnel` et vérifiez que les tests unitaires correspondants passent avec `./unit-test`

2.1.3 Atténuation

Sur une surface rugueuse, les différentes rugosités génèrent des phénomènes d'occultation partielle de la surface pour certaines directions. Ces occultations se traduisent par de l'ombrage et du masquage de la surface et dépendent de la fonction de distribution des micro-facettes. Dans notre cas (distribution de Beckmann), La fonction d'ombrage et de masquage de Smith permet de modéliser ces phénomènes. Elle est le le dernier terme manquant pour le calcul de la BSDF complète.

Cette fonction est définie par

$$G(\overrightarrow{\mathbf{v}}, \overrightarrow{\mathbf{l}}, \overrightarrow{\mathbf{h}}, \overrightarrow{\mathbf{n}}, \alpha) = G_1(\overrightarrow{\mathbf{l}}, \overrightarrow{\mathbf{h}}, \overrightarrow{\mathbf{n}}, \alpha).G_1(\overrightarrow{\mathbf{v}}, \overrightarrow{\mathbf{h}}, \overrightarrow{\mathbf{n}}, \alpha)$$

Pour calculer $G_1(\vec{\mathbf{x}}, \vec{\mathbf{h}}, \vec{\mathbf{n}}, \alpha)$ (pour $\vec{\mathbf{x}} = \vec{\mathbf{l}}$ ou $\vec{\mathbf{v}}$), il faut d'abord calculer

$$b = \frac{1}{\alpha \tan \theta_x} \qquad k = \frac{\vec{\mathbf{x}} \cdot \vec{\mathbf{h}}}{\vec{\mathbf{x}} \cdot \vec{\mathbf{n}}}$$

où θ_x est l'angle entre $\vec{\mathbf{n}}$ et $\vec{\mathbf{x}}$.

Si $b < 1.6$:

$$G_1 = \chi^+(k) \frac{3.535b + 2.181b^2}{1 + 2.276b + 2.577b^2}$$

Sinon

$$G_1 = \chi^+(k)$$

On se rappellera qu'on peut calculer $\tan \theta_x$ à partir de $\cos \theta_x = \vec{\mathbf{n}} \cdot \vec{\mathbf{x}}$ avec la formule suivante

$$\tan \theta_x = \frac{\sqrt{1 - \cos^2 \theta_x}}{\cos \theta_x}$$

2.2 Implémentation de la BSDF

Il ne reste plus qu'à mettre toutes ces fonctions dans le bon sens pour finir notre calcul d'éclairage dans la fonction `shade()`.

- Établir les vecteurs $\vec{\mathbf{l}}$, $\vec{\mathbf{v}}$ et $\vec{\mathbf{n}}$ à partir des informations décrivant l'intersection et le rayon traité.
- Calculer le *half-vector* $\vec{\mathbf{h}}$

$$\vec{\mathbf{h}} = \frac{\vec{\mathbf{v}} + \vec{\mathbf{l}}}{\| \vec{\mathbf{v}} + \vec{\mathbf{l}} \|}$$

- Calculer les produits scalaires $(\vec{\mathbf{l}} \cdot \vec{\mathbf{h}})$, $(\vec{\mathbf{n}} \cdot \vec{\mathbf{h}})$, $(\vec{\mathbf{v}} \cdot \vec{\mathbf{h}})$, $(\vec{\mathbf{l}} \cdot \vec{\mathbf{n}})$, et $(\vec{\mathbf{v}} \cdot \vec{\mathbf{n}})$.
- Passer tous ces paramètres à la fonction BSDF et retourner `ret = lc * RDM_bsdf(...)* LdotN`

La BSDF est donc définie par la fonction

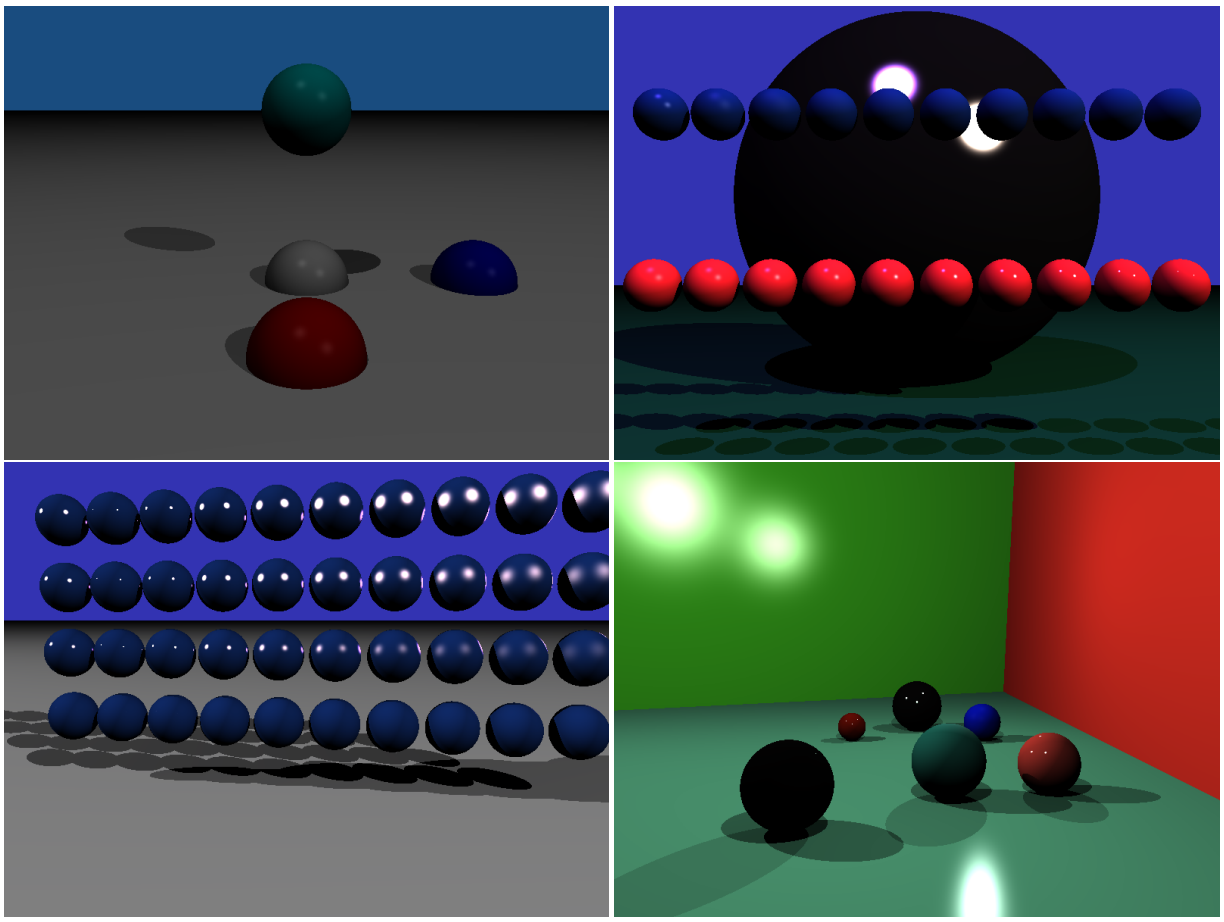
```
color3 RDM_bsdf(float LdotH, float NdotH, float VdotH, float LdotN,
   float VdotN, Material *m);
```

C'est elle qui se charge du calcul final du coefficient de la couleur. On commence par calculer D, F et G d'après les définitions précédentes. (Le paramètre α correspond au champ `m->roughness` de la structure `Material`).

$$BSDF = \frac{K_d}{\pi} + K_s \frac{D.F.G}{4(\vec{\mathbf{l}} \cdot \vec{\mathbf{n}})(\vec{\mathbf{v}} \cdot \vec{\mathbf{n}})}$$

K_d et K_s sont respectivement les paramètres `m->diffuseColor` et `m->specularColor`.

Implémentez les fonctions `RDM_G1`, `RDM_Smith` puis `RDM_bdsf_s` combinant les différents termes de la partie spéculaire de la BSDF. Implémentez ensuite `RDM_bsdf_d` puis `RDM_bsdf`. Enfin modifiez `shade` pour appeler `RDM_bsdf`.



2.3 Réflexions

L'ajout des réflexions ressemble au calcul des ombres : quand on calcule la couleur au point d'intersection, il faut ajouter la contribution provenant du rayon réfléchi en ce point. Donc, après avoir calculé l'éclairage dû aux sources de lumière, il faut déterminer le rayon réfléchi, puis calculer la contribution du rayon réfléchi c_r en appelant `trace_ray` et mélanger cette contribution avec la couleur obtenue par l'éclairage *direct* c_d. Vous pouvez utiliser la fonction `vec3 reflect(vec3, vec3)` fournie par GLM pour calculer la direction du rayon réfléchi.

Attention, pour ne pas avoir une infinité de rebonds, vous utiliserez le compteur `depth` du rayon. Ce compteur, initialisé à 0, doit être incrémenté à chaque nouveau rebond. La fonction de tracé de rayon s'arrête directement si un rayon a un compteur dépassant la limite autorisée (10 par exemple), dans ce cas elle renvoie une couleur spécifique (du noir).

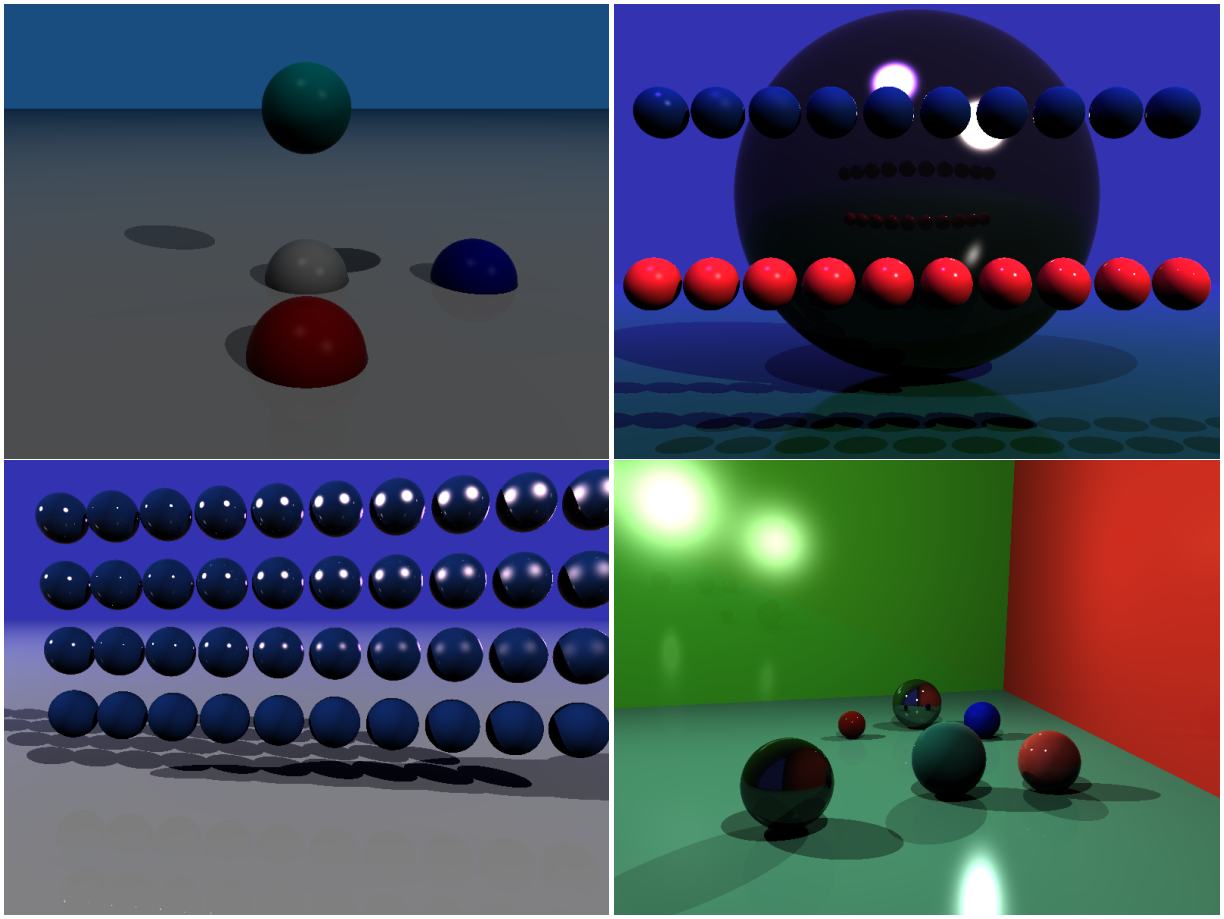
Vous pouvez améliorer ce critère d'arrêt en tenant compte de la contribution maximale que peut avoir le rayon réfléchi sur le pixel.

La contribution du reflet est ensuite additionnée à la couleur due à l'éclairage direct, en modulant par la couleur spéculaire c_s du matériau et par le terme de Fresnel :

$$c = c_d + F * c_r * c_s$$

Le terme de Fresnel est évalué en prenant comme direction de "lumière" la direction du vecteur réfléchi.

Modifiez `trace_ray`, en ajoutant l'envoi du rayon réfléchi, et en combinant le résultat à la couleur calculée.

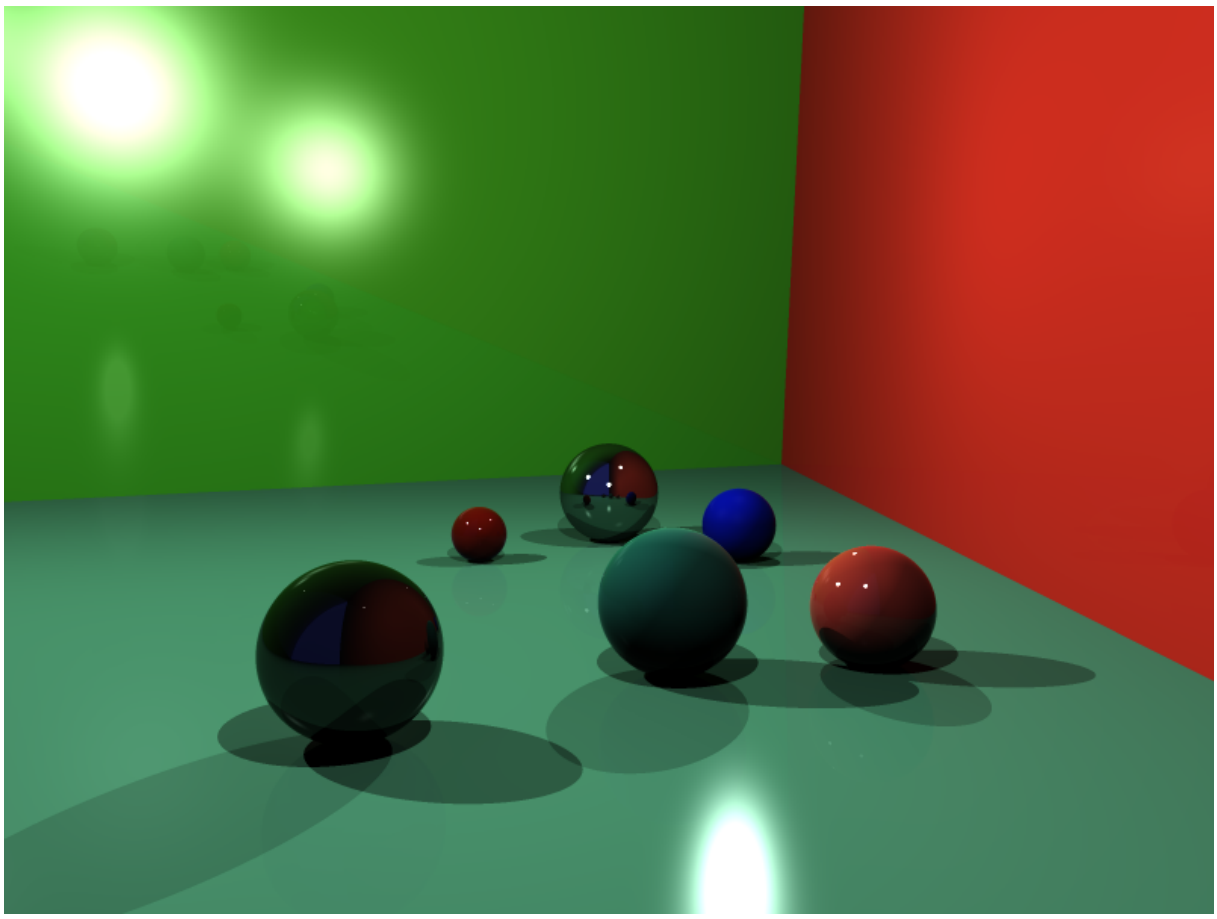


Chapitre 3

Fonctionnalités avancées

3.1 Antialiasing

L'aliasing est un problème d'échantillonnage. Prendre pour la couleur d'un pixel uniquement la couleur du rayon passant par le centre du pixel n'est qu'une simple approximation. Dans l'idéal, il faudrait intégrer la couleur reçue pour tous les rayons pouvant passer par le pixel. Dans la pratique, cette intégrale (non calculable) est approchée en faisant la moyenne de la couleur obtenue pour un "certain nombre" de rayons. Ajouter la gestion de l'anti-aliasing en vous inspirant par exemple de cette référence, section supersampling.



3.2 kd-tree

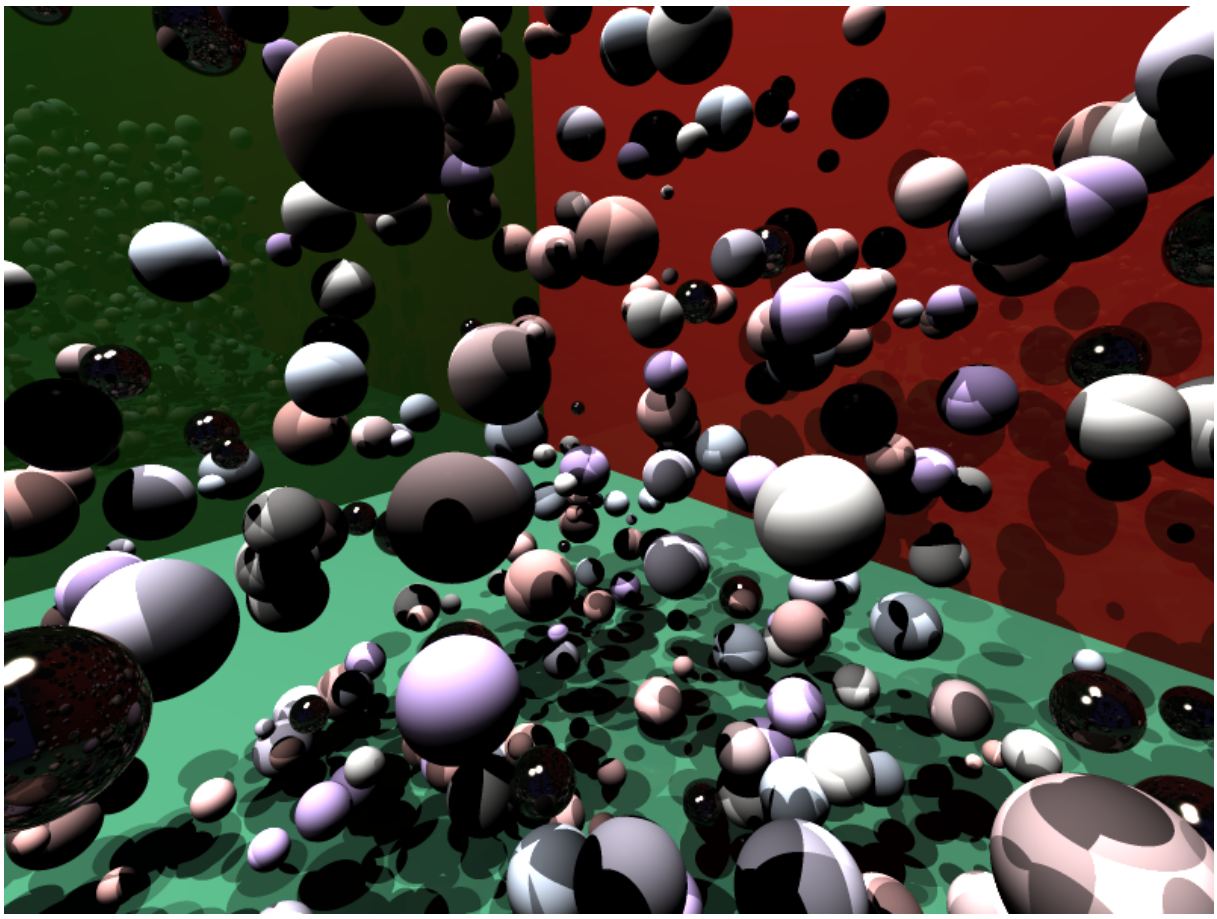
Cette section est une présentation très rapide, vous devez lire la références pour arriver à vos fins.

Référence ici http://www.eng.utah.edu/~cs6965/papers/kdtree.pdf

Un kd-tree est une structure d'accélération permettant d'éviter de tester l'intersection d'un rayon avec tous les objets de la scène. L'idée est de ranger les objets de la scène dans des boîtes disjointes et de ne tester l'intersection des objets contenus dans une boîte que si le rayon considéré intersecte cette boîte. Le kd-tree permet de représenter une hiérarchie de boîtes. La boîte située à la racine de l'arbre correspond à la boîte englobante de la totalité de la scène, et contient donc l'ensemble des objets de la scène. Cette boîte est ensuite découpée en deux, chaque objet de la scène est rangé dans la sous-boîte correspondante. Ce découpage est relancé récursivement sur chaque sous-boîte, afin de construire une hiérarchie de boîtes. Cette construction récursive a comme critère d'arrêt la profondeur maximum de la structure ou le nombre minimum d'objets par boîte.

L'ajout d'un kd-tree au projet se fait en deux étapes : la construction du kd-tree avant de commencer à tester des intersections, puis la traversée du kd-tree lors des calculs d'intersection.

Vu les objets que vous avez ajoutés à votre scène, nous traiterons différemment les plans infinis, qui ne seront pas dans le kd-tree, et les sphères qui seront dans le kd-tree. Chaque boîte du kd-tree contiendra donc les sphères qui sont entièrement ou partiellement contenues dans la boîte. Certaines sphères pourront donc être contenues par plusieurs boîtes. *Sur notre implémentation de référence, le kd-tree permet de diviser le temps de rendu par 4 pour une scène telle que ci-dessous (600 sphères).*



3.3 Autres objets

Implémentez les fonctions d'intersection pour cylindre, cône, triangle, et créez une scène contenant ces différents objets.

3.4 Texture

Pour plaquer une texture sur un objet, il faut d'une part avoir une paramétrisation 2D de l'objet et d'autre part une fonction de texture, définie par exemple par une image. L'image peut servir à moduler n'importe quel paramètre de la BSDF. Vous pouvez montrer, par exemple, une texture plaquée sur un plan.

3.5 Faites nous rêver

Vous pouvez, par exemple, implémenter un effet de profondeur de champ, ou encore de l'éclairage global en envoyant plus d'un rayon réfléchi ...