L2 Projet AII3D : miniraytracer

February 28, 2015

1 Objectif

L'objectif de ce projet est d'implémenter un lancer de rayon $\mathcal{M}_{\text{ini}}\mathcal{R}^{\text{ay}}\mathcal{T}_{\text{racer}}$. Pour rappel, le lancer de rayon est une approche de synthèse d'images dans laquelle la génération d'une image se fait en calculant l'intersection de rayons avec la surface d'objets 3D. A chaque point d'intersection, l'interaction lumineuse est calculée pour déterminer quelle est la couleur qui part dans la direction de ce rayon. Les données en entrées sont représentées par une scène 3D. Il s'agit d'une liste d'objets 3D (forme + matériau), de lumières, et d'une caméra.

Pour mener à bien ce projet, le document vous fournit les exigences fonctionnelles, ainsi qu'un guide pour la réalisation de du projet.

2 Planning de TPs

TP1 : mise en place de l'environnement et premières images

TP2 : rayons primaires et intersection avec une sphere, un plan (bonus intersection avec un cylindre, un cône ...)

TP3 : éclairage, matériau, blinn-phong (bonus matériau damier, texture ...)

TP4 : ombre et réflexion (bonus structure d'accélération octree)

TP5 : fresnel, anti-aliasing (bonus réfraction)

3 Guide de programmation

Il ne s'agit que de conseils, vous pouvez programmer le projet comme bon vous semble et si vous pensez avoir une meilleur solution, n'hésitez pas à l'utiliser. Les conseils sont faits pour vous aider à finir le projet mais ne prétend aucunement vous amener à une solution optimale.

Vous êtes encouragés à effectuer des tests unitaires : vérifiez que vos fonctions retournent les résultats attendus. Pour cela, il suffit de faire un petit main() qui appelle certaines fonctions et vérifie leurs résultats.

3.1 Calcul vectoriel

Un raytracer est basé grandement sur le calcul vectoriel, pour les positions dans l'espace mais aussi pour les couleurs. La structure vecteur, ainsi que les fonctions de base sur les vecteurs sont fournis (vector.h, vector.c). La documentation des fonctions se trouve dans le fichier .h

Un vecteur de R^3 (x, y, z) n'est rien d'autre qu'un triplet de trois valeurs. Ces trois valeurs peuvent être interprétées comme un point dans l'espace (dans un repère donné), comme un vecteur (déplacement entre deux points) ou encore comme une couleur (la coordonnée x représente le rouge, y le vert, z le bleu). Pour faciliter votre implémentation, nous fournissons les alias de type suivant :

```
typedef vec3 point3;
typedef vec3 color;
```

Mais en C, le compilateur acceptera silencieusement une variable de type color alors qu'une variable de type point est attendue.

Les fonctions suivantes sont fournies : produit scalaire, produit vectoriel, addition de vecteurs, soustraction de vecteurs, multiplication par un scalaire, calcul de la norme d'un vecteur, normalisation d'un vecteur, restriction des valeurs des composantes d'un vecteur (*clamp*, utile pour les couleurs), calcul de vecteur réfléchi.

3.2 Les variables globales

Pour faciliter le développement, le fichier global.h déclare un certain nombre de variables globales, qui sont accessibles directement dans toutes vos fonctions, sans devoir passer par les paramètres. Les variables globales sont l'image à rendre, les données de la scène ainsi que la caméra. Les variables globales sont définies dans le fichier main.c

3.3 Gestion des images

Vous avez pour démarrer un programme main() qui sauvegarde une image dont les couleurs sont affectées dans main(). Pour changer la couleur vous devez affecter une valeur entre 0 et 255 à chacune des composantes (rouge, vert, bleu) d'un pixel. Cependant pour le calcul de la couleur lors du raytracing, nous utilisons des valeurs flottantes entre 0.f et 1.f et nous effectuons la conversion (multiplier par 255) au moment de la sauvegarde. Vous utiliserez donc la variable globale color output_image[HEIGHT∗WIDTH], qui a une taille de WIDTH $\times$ HEIGHT et est du type color défini ci-dessus. Ainsi la composante x du vecteur sera la valeur pour le rouge, y pour le vert et z pour le bleu. Concernant la variable globale pour le stockage de l'image, il est fréquent en informatique, pour diverses raisons que nous ne détaillerons pas ici, d'utiliser un bloc mémoire contigue, au moyen d'un tableau à une dimension, pour représenter un tableau de dimension plus élevée. Pour le cas d'un tableau 2D de dimension $w \times h$, l'index k d'une case de coordonnées $i \in [0, w-1], j \in [0, h-1]$ est $k = j \times w + i$.

3.4 Programme principal

Le code fourni pour le programme principal (main.c) analyse les paramètres de la ligne de commande pour en extraire un entier et une chaine de caractères. L'entier définit la scène utilisée pour l'exécution du programme. Une valeur entre 0 et 3 appelle les fonctions du premier TP. La chaine de caractère correspond au nom de l'image (sans l'extension) générée par votre programme. Ainsi ./mrt 0 resultat0 génère une image complètement rouge (une fois que vous avez fait la première fonction du TP1), nommée resultat0.ppm ou resultat0.png.

3.5 Fichiers fournis et compilation

Fichiers fournis :

- main.c : main de votre programme, initialise la scène et sauvegarde l'image en fonction des paramètres de la ligne de commande.
- Vous devez implémenter les fonction du tp 1 dans ce fichier, les déclarations des fonctions sont données
- globals.h : décalarations des variables globales et des defines, pour tester vous pouvez changer la taille de l'image ici.
- vector.h/vector.c : fonction mathématique sur les vecteurs, tout est fourni, vous n'avez rien à modifier ici.
- ray.h : structure et fonction pour représenter un rayon : vous n'avez rien à modifier ici.
- scene.h/scene.c : structure et fonction d'iniatlisation pour remprésenter une les objets 3D, vous n'avez rien à modifier ici.
- raytracer.h/raytracer.c : ici c'est pour faire le gros du TP, vous pouvez ajouter des fonctions si besoin.

g++ −g −O3 vector.c scene.c main.c octree.c raytracer.c −lm −o mrt

4 TP1

Vous pouvez maintenant commencer le premier TP.

- Compléter la fonction fill_red (color * img) qui affecte une couleur rouge (color.x = 1.f;) à tous les pixels de l'image.
- Compléter la fonction fill (color *img, color c) qui affecte la couleur c à tous les pixels de l'image.
- Compléter la fonction horizontal_gradient(color *img, color c1, color c2). Cette fonction remplit img avec un dégradé horizontal de c1 (en haut) à c2 en bas.
- Compléter la fonction fill_rect (color *img, color c, point3 p1, point3 p2) qui dessine un rectangle dans l'image. (int)p1.x, (int)p1.y et (int)p2.x et (int)p2.y sont les coordonnées en pixels des coins du rectangle (p1.z et p2.z sont ignorés). Cette fonction affecte c à tous les pixels de img dont les coordonnées sont comprises entre (int)p1.x, (int)p1.y et (int)p2.x et (int)p2.y
- Compléter la fonction fill_circle (img *img, float radius, point3 centre, color c) qui remplit un cercle dans l'image. Le plus simple est de parcourir l'ensemble des pixels de l'image et de tester la distance entre le pixel et le centre du cercle, si cette distance est inférieure au rayon, on affecte la couleur.

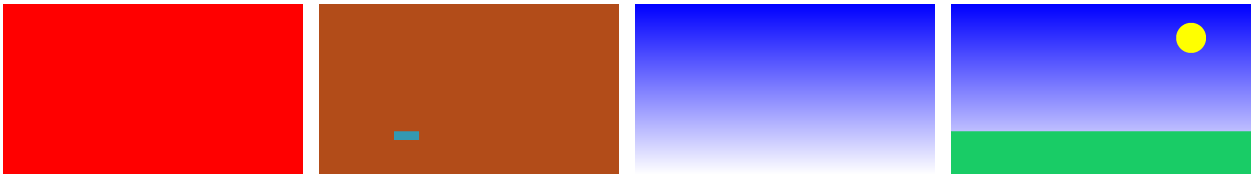
$$I'[x,y] = \begin{cases} c & \text{si } ||(x,y) - p|| < r \\ I[x,y] & \text{sinon} \end{cases}$$

  Avec I' l'image modifiée et I l'image d'origine.[1] On peut optimiser cette fonction en limitant les tests à la boite englobante du disque (on est sur que les pixels en dehors de cette boite englobante sont en dehors du disque). Cette boite est définie par les point $c - (r,r)$ et $c + (r,r)$. De plus un test sur la distance au carré est suffisant, et supprime l'utilisation (couteuse) de sqrt.

[1] dans la mesure du possible, les notations mathématiques et le code utilisent des noms de variables proches, mais en mathématiques le symbole = correspond à l'égalité, alors qu'en C il s'agit de l'affectation. Ce qui fait que l'on doit distinguer une variable avant et après modification en mathématiques, alors qu'en C, on peut modifier la valeur d'une variable

```
int dx = x - center.x;
int dy = y - center.y;
if(dx*dx+dy*dy<radius*radius) img[y*WIDTH + x] = c;
```

Vous devez obtenir les images de la figure 1



./mrt 0 resultat0 ./mrt 1 resultat1 ./mrt 2 resultat2 ./mrt 3 resultat3

Figure 1: Résultat de l'exécution de votre programme après le premier TP.

Vous pouvez maintenant générer un premier dessin à l'aide de ces fonctions, qui sera généré par la commande ./mrt 4 mondessin. Laissez libre cours à votre imagination.

5 TP2

Nous commençons maintenant l'implémentation du lancer de rayon.

5.1 Exigences fonctionnelles

Le raytracer devra gérer les primitives sphère et plan.

Pour les matériaux, le raytracer permettra d'avoir des réflexions, en plus d'une BRDF de type Blinn Phong.

Le raytracer permettra aussi la gestion de l'éclairage et des ombres.

L'exécution du raytracer produira une image nommée output.png dans le répertoire d'exécution. Cette image fait 1280 par 720. Ces dimensions sont référencées par WIDTH et HEIGHT dans la suite du document.

- Le choix d'une scène à tracer se fera en passant un entier en paramètre à l'appel du programme.
- Une profondeur de réflexion de 10 est attendue, si une telle profondeur est atteinte, la couleur renvoyée est le noir (0,0,0).
- Les lumières sont des lumières ponctuelles.
- Les ombres sont des ombres dures.

5.2 Rayon

Une structure de base est le rayon, constitué d'une origine et d'une direction. En plus, un rayon stocke le t_{max} qui est le paramètre 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. Par exemple pour le calcul des ombres, une intersection qui serait derrière la lumière n'est pas à prendre en compte.

Un rayon est donc défini par

$$r(t) = o + t\vec{d} \quad t \in [0, +\infty]$$

La structure fournie en C est la suivante :

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

L'entier int depth sert à stocké le nombre de "rebonds" fait par un rayon, pour arréter la recursion si besoin.

5.3 Caméra

Pour effectuer le rendu d'une scène, il faut placer une camera dans la scène. Celle-ci est définie par un repère affine : une origine (position), et trois vecteurs. Les vecteur forme une base orthonormée.

La structure fournie en C est la suivante :

```
typedef struct camera_s{
  point position;
  vec3 zdir;
  vec3 xdir;
  vec3 ydir;
  vec3 center;
  float fov;
  float aspect;
} camera;
```

Par convention, nous utiliserons x et y comme axes du plan image et lancerons les rayons dans la direction z. Il est plus facile de définir cette base en utilisant un vecteur direction de vue ($\vec{z}_{dir}$) et un vecteur "vers le haut" ($\vec{y}_{dir}$). Pour obtenir le vecteur $\vec{x}_{dir}$ –perpendiculaire au plan défini par $\vec{z}_{dir}$ et $\vec{y}_{dir}$– il faut faire le produit vectoriel entre $\vec{z}_{dir}$ et $\vec{y}_{dir}$. Ensuite pour avoir y qui forme une base orthonormée avec $\vec{x}_{dir}$ et $\vec{z}_{dir}$, on prend encore le produit vectoriel, cette fois ci entre $\vec{x}_{dir}$ et $\vec{z}_{dir}$. De plus, une caméra possède un angle d'ouverture, prendre cette angle suivant la direction x. Pour effectuer le rendu, vous devez créer des rayons passants par chacun des pixels de l'image. L'origine de ces rayons est la position de la caméra. Pour calculer la direction $\vec{d}$, il faut déterminer l'angle formé par le rayon et chacun des deux axes x et y de la caméra en fonction des coordonnées du pixel à tracer et des données de la caméra. La caméra de la scène sera stockée dans une variable globale.

Le plan image est entièrement contenu dans la pyramide définie par l'angle d'ouverture de la caméra. Chaque rayon passe par le centre d'un pixel. Voir figure 2.

Pour faciliter le calcul, vous pouvez définir des vecteurs $\vec{x}_r = \vec{x}_{dir}$, $\vec{y}_r = 1/aspect * \vec{y}_{dir}$ et $\vec{z}_r = \frac{1}{\tan(.5fov)}\vec{z}_{dir}$ tel que le pixel de coordonnées (i, j) ait pour direction

$$\vec{d} = \vec{z}_r + \frac{(i + 0.5 - WIDTH/2)}{(WIDTH/2)}\vec{x}_r + \frac{(j + 0.5 - HEIGHT/2)}{(HEIGHT/2)}\vec{y}_r$$

Vous pouvez maintenant écrire la fonction qui va lancer les rayons pour chaque pixel de la scène. Il s'agit de faire une boucle sur l'ensemble des pixels, de calculer la direction du pixel, puis d'appeler une fonction de calcul d'intersection avec la scène, que vous implémenterez plus tard. Commencez simplement par remplir votre image avec des valeurs en fonction de i, j, puis en fonction de la direction du rayon en x, et y. La figure 3 montre un exemple de tests pour voir si le rayon 0,0 est bien en bas a droite (pour le rayon 0,0 on a $-1x_r$ et $-1y_r$, pour avoir une couleur, on le ramène entre 0 et 1, et on l'affecte au rouge et au vert

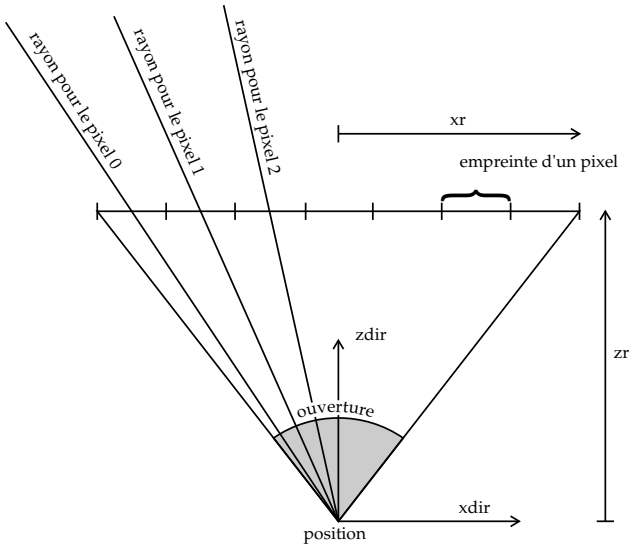


Figure 2: Schéma montrant les rayons envoyés depuis la caméra pour chaque pixel du plan image.

de la couleur du pixel correspondant). Vous pouvez aussi tester en remplissant l'image avec la valeur du produit scalaire entre la direction de votre rayon et le vecteur z de la caméra, ce produit scalaire doit être 1 pour les pixels vers le centre de l'image, diminuer vers les bords.

Vous pouvez maintenant implémenter la fonction void raytrace() du fichier raytracer.c. Cette fonction inialise un rayon pour chaque pixel de l'image à rendre, et appel la fonction de tracer pour ce rayon.

5.4 Lancer d'un rayon

Passez maintenant a la fonction de lancer de rayon. Cette fonction va calculer la couleur d'un rayon passé en paramètre. Dans une première étape, cette fonction renvoie la couleur du ciel – (0.2, 0.2, 0.6) dans les exemples –.

Une fois que vous aurez implémenté la fonction d'intersection avec la scène, cette fonction calculera l'intersection avec la scène. En cas d'intersection il faudra calculer la couleur de cette intersection, en fonction des lumières de la scène, puis éventuellement envoyer un rayon réfléchit.

5.5 Définition de la scène

Nous allons maintenant passer à la définition de la scène. Une scène 3D est définie par un ensemble d'objets. Nous vous proposons d'utiliser un tableau statique d'objets.

Chaque objet sera représenté par une structure en C. Pour permettre de placer les différents types d'objet dans le même tableau, nous utiliserons une unique structure pour tous les types d'objet, avec un champs int type. La structure object est définie dans scene.h, est la correspondance type vers entier dans globals.h.

```
// OBJECT
typedef struct object_s{
  int type;
  material mat;
  union{
    struct{
```

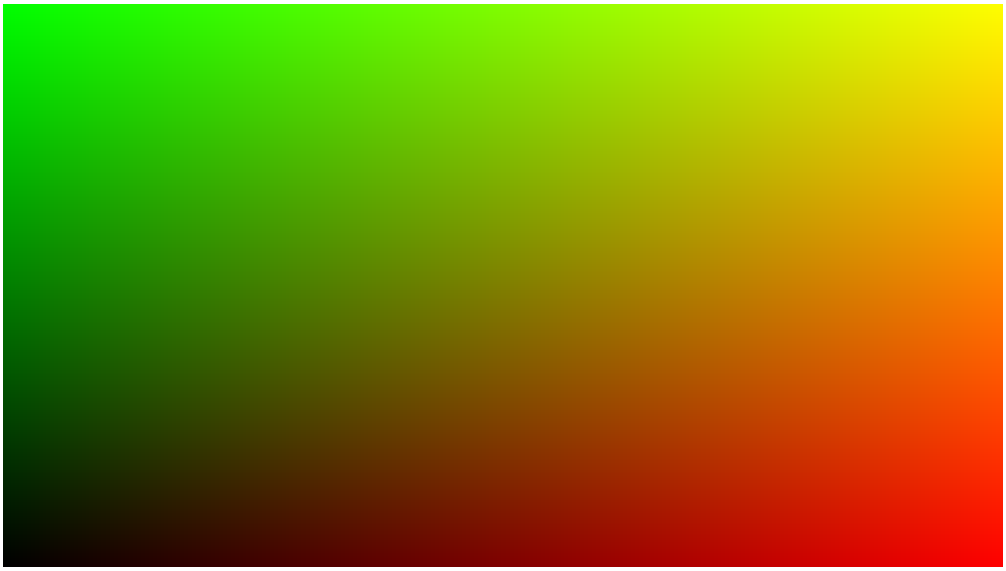


Figure 3: Sur cette image (R,V,B) = $(\frac{(i+0.5-WIDTH/2)}{(WIDTH/2)} * .5 + .5, \frac{(j+0.5-HEIGHT/2)}{(HEIGHT/2)} * .5 + .5, 0)$.

```
      // sphere
      vec3 center;
      float radius;
    };
    struct{
      // plan
      vec3 normal;
      float dist;
    };
  };
} object;
```

Une petite note : ici pour un soucis de facilité d'écriture du code, nous utilisons des structures anonymes. Cette utilisation n'est pas dans la norme de C99 (mais dans celle du C2011). De plus le programme est compilé en C++, pour diverses raisons. Les structures anonymes ne sont pas non plus dans la norme du C++, mais leur utilisation est possible avec g++. Pour accéder à un champ d'une variable object obj; on fera simplement obj.type ou obj.center. L'astuce est que l'on a des noms cohérents avec le type d'objet sans occuper plus de mémoire (nous vous laissons approfondir la syntaxe du C sur les unions). En tous cas vous ne devez accéder qu'aux champs de l'union qui correspondent au type de votre objet, et tout se passera pour le mieux.

Une variable globale tableau d'objets contient alors la scène, et l'entier object_count le nombre d'objets valides dans ce tableau.

5.6 Matériau

Chaque objet de la scène a un matériau. Pour commencer, le matériau a simplement une couleur diffuse

```
typedef struct material_s{
    color kd;
} material;
```

Vous pouvez ignorer les autres champs de la structures, et vous ajouterez éventuellement plus tard les textures procédurales ou à partir d'image.

5.7 Plan

Un plan est défini par une normale $\vec{n}$ et une distance à l'origine d. Un point p appartient au plan ssi $p.\vec{n} + d = 0$.

L'intersection avec un rayon se fait en remplaçant les coordonnées du rayon dans l'équation du plan :

$$(o + t\vec{d}).\vec{n} + d = 0$$

Qui a pour solution :

$$t = -\frac{o.\vec{n} + d}{\vec{d}.\vec{n}}$$

L'intersection est valide ssi $t \in [0, t_{max}]$ sinon soit elle est avant l'origine du rayon, soit elle est hors du segment qui nous intéresse.

5.8 Sphère

Avant d'aller plus loin vous allez aussi faire l'intersection avec une sphère. Une sphère est définie par un centre c et un rayon r, un point p appartient à la sphère si il vérifie l'équation :

$$(p_x - c_x)^2 + (p_y - c_y)^2 + (p_z - c_z)^2 - r^2 = 0$$

L'intersection avec un rayon se fait, comme pour le plan, en remplaçant les coordonnées du rayon dans l'équation de la sphère :

$$((o_x + t\vec{d_x}) - c_x)^2 + ((o_y + t\vec{d_y}) - c_y)^2 + ((o_z + t\vec{d_z}) - c_z)^2 - r^2 = 0$$

Qui peut se réécrire en fonction de l'inconnue t :

$$t^2(\vec{d_x}^2 + \vec{d_y}^2 + \vec{d_z}^2) + 2t(\vec{d_x}(\vec{o}_x - c_x) + \vec{d_y}(\vec{o}_y - c_y) + \vec{d_z}(\vec{o}_z - c_z)) + ((o_x - c_x)^2 + (o_y - c_y)^2 + (o_z - c_z)^2 - r^2) = 0$$

Ou encore, si on utilise le produit scalaire, on peut écrire :

$$t^2(\vec{d}.\vec{d}) + 2t(\vec{d}.(o - c)) + ((o - c).(o - c) - r^2) = 0$$

Suivant les solutions réelles t_0 et t_1 de cette équation du second degré, il y a 0, 1 ou 2 intersection(s). Celle qui nous intéresse est la plus petite appartenant à $[0, t_{max}]$.

5.9 Structure d'intersection

La fonction qui calcule si il y a intersection avec un objet de la scène doit retourner en plus d'un booléen (intersection ou non), les valeurs permettant le calcul de la couleur au point d'intersection. Il vous faut la normale et le matériau. La position de l'intersection servira lorsque vous voudrez faire rebondir un rayon à partir du point d'intersection (pour les ombres par exemple). Ces valeurs seront stockées dans une structure d'intersection.

```
typedef struct intersection_s{
  vec3 normal; //! the normal at the intersection point
  point position; //! the 3D position of the intersection point
  material mat; //! material of the intersected object
} intersection;
```

5.10 Intersection avec la scène

Pour effectuer la fonction d'intersection avec la scène, il faut tester l'intersection avec chacun des objets de la scène et garder la plus proche. Pour cela, les différentes fonctions d'intersection avec les objets ne vont considérer une intersection que si la position le long du rayon a un paramètre inférieur à t_{max}.

Implémentez cette fonction.

Maintenant, implémentez la fonction qui trace un rayon. Celle-ci renvoie la couleur que prend le rayon. La première étape est d'appeler l'intersection avec la scène. Si il y a une intersection, il faut calculer la couleur. Pour tester, on peut, par exemple, envoyer la couleur diffuse (k_d) de l'objet intersecté ou la couleur du ciel si il n'y a pas d'intersection.

Vous devez obtenir l'image no_shading.png avec la scène 0.

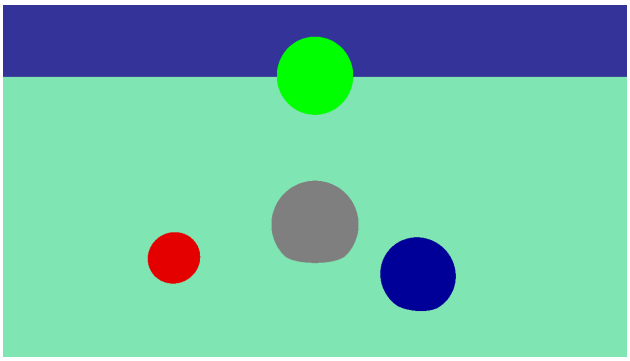


Figure 4: Scène 0 avec une couleur unie k_d par objet

6 TP3 : matériau

Pour obtenir un éclairage plus réaliste, implémentez une fonction d'éclairage de type Blinn-Phong :

$$c = I\vec{n}.\vec{l}\Big(\frac{k_d}{\pi} + (\vec{h}.\vec{n})^s \frac{s+8}{8\pi} k_s\Big) \tag{1}$$

Où I est la couleur de la lumière, k_d est la couleur diffuse du matériau, k_s est la couleur spéculaire du matériau et s le coefficient de spécularité.

Commencez par ajouter k_s et s à la structure matériau, puis implémentez une fonction qui calcule la couleur en fonction des vecteurs et du matériau. Pour rappel, $\vec{h}$ est le demi-vecteur entre la lumière $\vec{l}$ et le point de vue $\vec{v}$: $\vec{h} = (\vec{l} + \vec{v})/||\vec{l} + \vec{v}||$.

Si il y a plusieurs lumières dans la scène, il faut additionner les contributions de chacune des lumières.

7 TP4

7.1 Ombres

Pour tenir compte des ombres, il faut tester l'intersection entre un rayon partant du point d'intersection vers la lumière. Si il n'y a pas d'intersection entre le point d'intersection et la lumière, alors le point d'intersection est éclairé, sinon il est dans l'ombre. Si vous ne prenez pas de précaution particulière pour initialiser le point de départ du rayon, vous risquez d'obtenir de l'acné. Pour remédier à ce problème il faut décaler légèrement le point de départ du rayon dans la direction de la lumière.

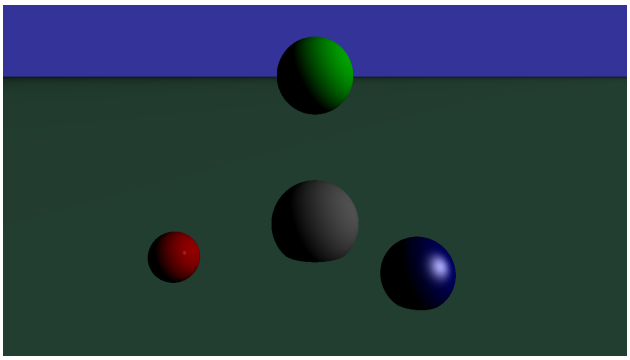


Figure 5: Scène 0 avec l'éclairage de Blinn-Phong

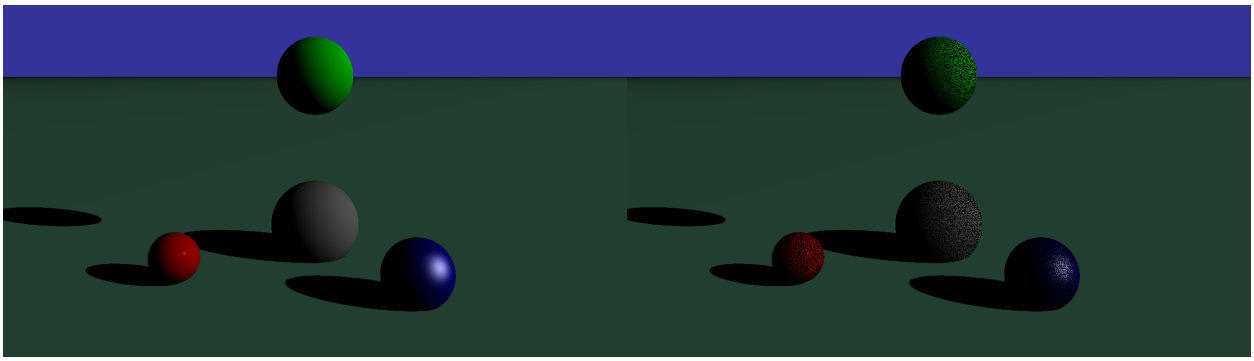


Figure 6: Scène 0 avec les ombres, l'image de droite montre le problème d'acné.

7.2 Réflexion

L'ajout des réflexions ressemble au calcul des ombres : il faut ajouter au calcul de la couleur au point d'intersection une contribution venant le long du rayon réfléchi en ce point. Pour cela, il faut calculer le rayon réfléchi, calculer l'intersection entre le rayon réfléchi et la scène, calculer la couleur c_r au point d'intersection et mélanger cette couleur avec la couleur obtenue pour l'éclairage *direct* c_d.

La première version pour le mélange des deux couleurs utilise la couleur k_s donnée par le matériau. Ainsi la couleur obtenue est

$$k_s c_r + c_d \tag{2}$$

Attention, pour ne pas avoir une infinité de rebonds, vous ajouterez un compteur à la structure de données rayon. Ce compteur est initialisé à 0, et incrémenté à chaque nouveau rebond. La fonction de tracé de rayon s'arrête directement si un rayon à un compteur dépassant la limite autorisée, dans ce cas elle renvoie une couleur spécifique (du noir).

8 TP5

8.1 Fresnel

Pour améliorer le réalisme des réflexions, vous pouvez utiliser les équations de Fresnel qui permettent de calculer quelle proportion d'un rayon est réfléchi et quelle proportion est réfractée. Ici, on s'intéresse à la réflexion. On peut utiliser l'approximation de Schlick :

$$
\begin{aligned}
reflexion &= F(k_r, \vec{n}, \vec{v}) \\
&= k_r + (1 - k_r)(1 - \vec{n}.\vec{v})^5 \\
k_r &= \left(\frac{n_1 - n_2}{n_1 + n_2}\right)^2
\end{aligned}
$$

avec k_r le coefficient de Fresnel, n_1 et n_2 les indices de réfraction du milieu externe et du milieu interne. Pour cela vous pouvez ajouter un champ k_r à la structure material, et modifier l'initialisation des scenes en conséquence.

reflexion remplace alors k_s dans l'équation 2. Vous pouvez aussi répercuter *reflexion* dans l'équation de Phong (Eq. 1) :

$$
c = I\ \vec{n}.\vec{l}\ \left(\frac{k_d}{\pi} + (\vec{h}.\vec{n})^s\ \frac{s+8}{8\pi}\ reflexion\right)
$$

Pour aller plus loin, nous pouvons utiliser des vrais indices de réfraction : un pour l'extérieur (en général, celui de l'air ou du vide) et un pour l'intérieur de chaque matériau réfléchissant. Vous pouvez trouver les indices de base de certains milieux ici : `https://en.wikipedia.org/wiki/List_of_refractive_indices`.

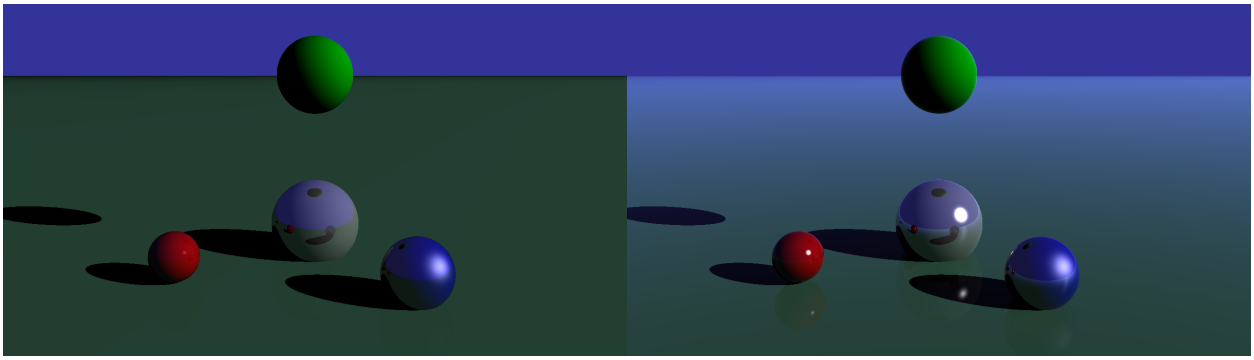


Figure 7: Réflexion sur la scène 0, à gauche en utilisant directement f comme facteur de réflexion, à droite en tenant compte du coefficient de Fresnel.

8.2 Anti-aliasing

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 approximée 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 `http://paulbourke.net/miscellaneous/aliasing/`. La figure 8 montre un exemple avec anti-aliasing.

9 Extensions

9.1 Cylindre

Ajoutez l'objet cylindre, ainsi que la fonction d'intersection avec un cylindre.

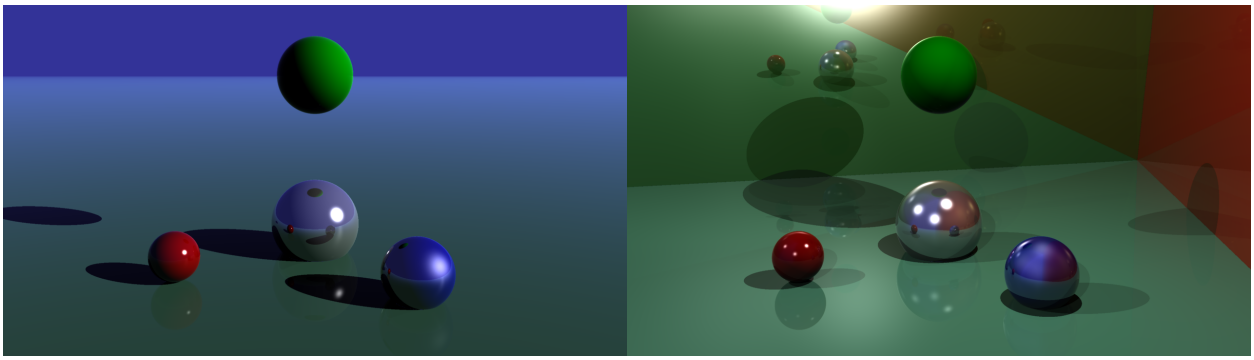


Figure 8: Rendu anti-aliasé des scènes 1 et 2.

9.2 Refraction

En suivant le même principe que pour les réflexions, on peut gérer des objets semi-transparents en prenant en compte les rayons réfractés par les matériaux. La proportion de lumière réfractée (ou *transmise* est :

$$transmise = 1 - reflexion$$

et le rayon réfracté a pour direction :

$$\vec{d}_{tran} = r + \left(r\ \vec{n}.\vec{l} - \sqrt{1 - r^2\left(1 - (\vec{n}.\vec{l})^2\right)} \right) \vec{n}$$

$$r = \frac{n_1}{n_2}$$

Lors d'une intersection avec un objet transparent, on pourrait lancer un rayon réfléchi et un rayon réfracté. Cependant, cela va augmenter de manière exponentielle la quantité de rayons lancés et ne permet donc pas de contrôler le temps de calcul. On préfère dans ce cas choisir de ne renvoyer qu'un seul rayon, au hasard en réflexion ou en réfraction. Pour cela, on tire au hasard un réel aléatoire $p \in [0, 1]$ et :
si ($p < reflexion$) alors
    on continue avec le rayon réfléchi
sinon
    on continue avec le rayon réfracté

9.3 Structure d'accélération : octree

Un octree représente une segmentation hiérarchique de l'espace. Il s'agit d'un arbre dont chacun des noeuds à huit fils. Chaque noeud contient une boite (alignée avec les axes). Chaque feuille contient zéro, un ou plusieurs objets de la scène. Ce sont les objets qui intersecte la boite correspondant à la feuille. Pour construire un octree, on commence par calculer la boite du noeud racine, c'est la boite englobante de la scène. Pour ne pas avoir de problème avec les plans infinis, ils seront traités à part et ne doivent pas entrer dans la construction de l'octree. On insère alors temporairement tous les objets de la scène dans le noeud racine. La suite de la construction est récursive.
    subdivise(noeud octree)
si(profondeur noeud >= profondeur max) stop subdiv
si(nombre object noeud <= nombre min) stop subdiv

sinon couper la boite du noeud en 8 boites b_i,
ajouter 8 fils
initialiser les boites des 8 fils avec les b_i
pour chaque objet *obj* du noeud
pour chaque fils f_i du noeud
si *obj* intersect b_i ajouter *obj* à la liste des objets de f_i
fin si
fin pour
fin pour
vider la liste des objets du noeud.

Notez qu'avec cette approche, les objets peuvent se trouver dupliquer plusieurs fois dans chaque noeud.

Ensuite, il vous faut utiliser l'octree pour accelérer le calcul d'intersection avec la scène : on teste l'intersection avec un noeud, si il y a intersection, alors on descend dans les fils. Quand on arrive à une feuille, on teste l'intersection avec les objets de cette feuille. L'intersection est valide que si elle est aussi dans le noeud.

9.4 Génération procédurale de scène

Vous devez construire une procédure qui peuple la scène avec des objets. La position des objets peut être inter-dépendante. On trouve beaucoup d'inspiration dans les fonctions fractales. Voici un petit exemple récursif : procédure peuple sphere (point P, rayon R, matrice m) Créer une sphère en P de rayon R Prendre trois directions d_i qui pointent dans l'hémisphère *haut* (à définir) défini par m créer sphère ($P + Rd_i + R/4$, $R/4$, matrice l'axe *haut* est d_i.

Laissez libre cours à votre imagination, en extrapolant un peu l'exemple, vous pourrez obtenir un image comme la figure 9. Vous devrez sans doute augmenter la limite du nombre d'objets de la scène. Pour que le rendu se calcule en un temps raisonnable, votre structure d'accélération doit être fonctionnelle.

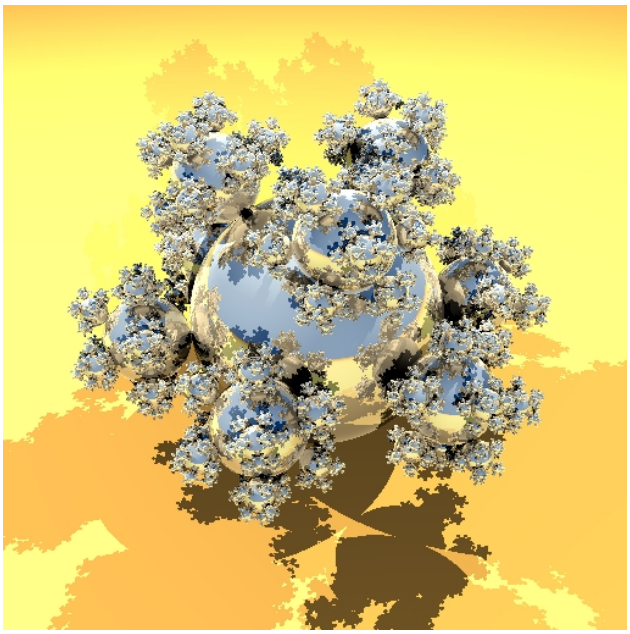


Figure 9: Modèle procédurale à base de sphère.