Introduction au Raytracing

par Nicolas Bonneel

Cours № 1

Préliminaires

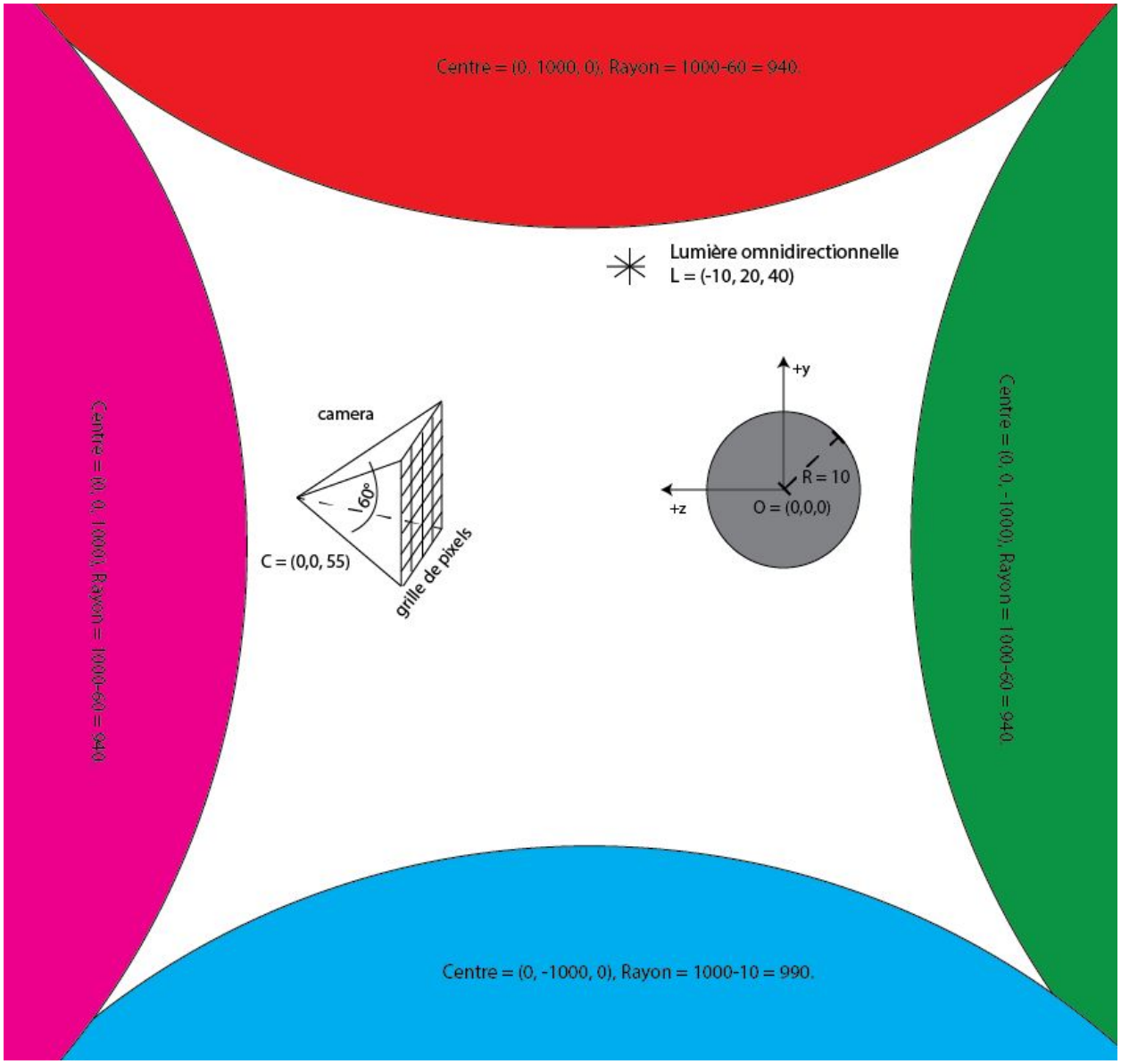
Il y a deux familles importantes de méthodes de rendu. Les méthodes basées sur de la “rasterization” consistent à projeter des formes géométriques (e.g., des triangles) à l’écran, comme le fait OpenGL que vous avez pu voir les semaines précédentes.
Les méthodes basées sur du lancer de rayons (ou Raytracing) consistent quant à elles à simuler le cheminement (ou cheminement inverse) de la lumière dans la scène.

Alors que les méthodes par rasterization sont très efficaces, et tirent souvent directement profit de la carte graphique, elles sont assez peu précises et les moindres effets physiques (réflexion, réfraction, éclairage indirect) sont difficiles à réaliser. Elles sont donc adaptées aux jeux, ou aux simulations interactives.

Les méthodes par lancer de rayon sont beaucoup plus lentes, mais simulent très bien et assez facilement toutes les interactions lumière-matière. Elles sont donc plus adaptées à l’industrie du cinéma ou de la simulation physique de l’éclairage, qui se permettent plusieurs heures de calcul par image (e.g., 47h par image pour Avatar, partagées sur des fermes de calcul).

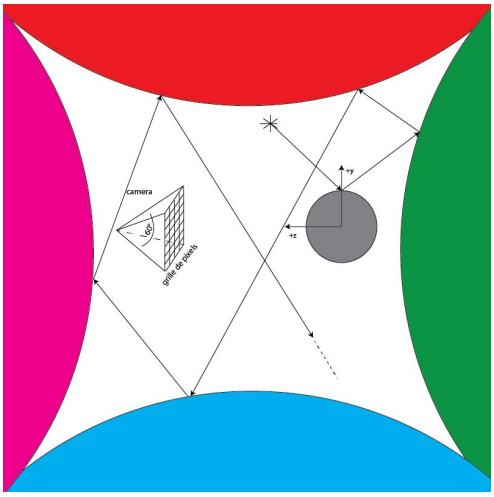
Le but de ces trois séances est de vous enseigner les bases du raytracing et vous faire implémenter un raytracer manipulant des primitives simples (sphères, plans, peut-être triangles). On améliorera ce raytracer en un “path-tracer”, i.e., en incluant les effets de l’éclairage indirect.

A titre indicatif, on pourra considérer la scène suivante, composée uniquement de sphères, que l’on améliorera au fil des séances (plusieurs sphères, des vrais plans, voire de la géométrie triangulée). Les murs latéraux ne sont pas indiqués par soucis de lisibilité.

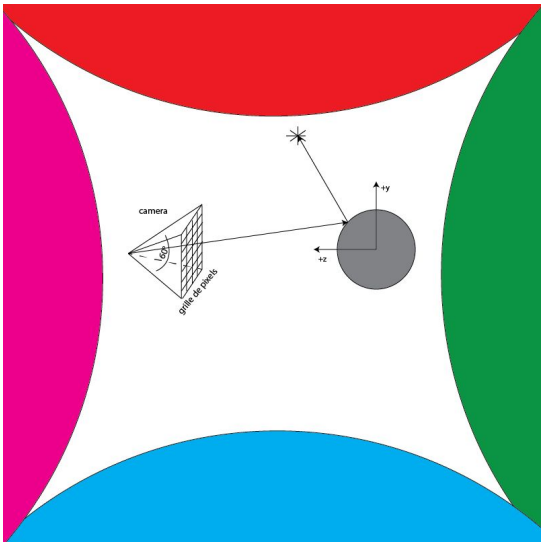

Centre = (0, 1000, 0), Rayon = 1000-60 = 940.
Lumière omnidirectionnelle
L = (-10, 20, 40)
camera
60°
C = (0,0, 55)
grille de pixels
+y
R = 10
+z
O = (0,0,0)
Centre = (0, 0, 1000), Rayon = 1000-60 = 940.
Centre = (0, 0, -1000), Rayon = 1000-60 = 940.
Centre = (0, -1000, 0), Rayon = 1000-10 = 990.

Principe

Le principe du raytracing est de simuler le comportement de la lumière. Si nous optons pour une stratégie qui envoie des rayons de lumière depuis toutes les sources de lumière, et simulons les interactions lumière-matière, très peu de ces rayons parviendraient au capteur (i.e., la caméra) ou au bout d'un temps très long comme l'indique la figure ci-après. *[NB : il existe cependant des techniques mixtes, tels que le photon mapping ou le bidirectional path tracing qui font cela ; nous n'allons pas étudier ces méthodes]*



Le (backward) raytracing, que nous allons étudier, bénéficie d'un principe fondamental: le principe de réciprocité d'Helmholtz. Ce principe dit que l'on peut suivre les rayons de lumière en sens inverse -- depuis le capteur, vers les sources de lumière -- et obtenir le même résultat. Nous allons donc lancer des rayons de lumière (des demi-droites), depuis la caméra et en direction de chaque pixel de l'image, et simuler des interactions lumière-matière.



Pour le premier cours, on considérera une scène composée de sphères, et on supposera toutes les surfaces diffuses : comme le plâtre, elles réfléchiront la lumière dans toutes les directions, indépendamment de la direction incidente. De plus, on ne considérera pas les différents rebonds de la lumière à travers la scène, mais uniquement l'éclairage direct (i.e., l'éclairage reçu par une source de lumière ponctuelle directement, et non l'éclairage ambiant dû aux autres interactions), et sans ombre portées.

Getting Started

La seule fonction que je vous laisserai recopier est la fonction qui permet de sauvegarder un fichier bmp. Je vous propose d'utiliser ce bout de code.

Cette fonction prend en parametre un tableau de valeurs qui représentent les valeurs rouge, verte, et bleue de chaque pixel.
Accéder aux composantes rouge, verte et bleue du pixel de la ligne i et colonne j s'écrit donc :

```
tableau_de_pixels[((hauteur-i-1)*largeur+j)*3] = rouge;
tableau_de_pixels[((hauteur-i-1)*largeur+j)*3 + 1] = vert;
tableau_de_pixels[((hauteur-i-1)*largeur+j)*3 + 2] = bleu;
```

On définira une classe Vector, fondamentale en 3D, qui permettra de manipuler des coordonnées 3D aisément. Cette classe contiendra des coordonnées x, y et z en double précision, une surcharge des opérateur de multiplication par un scalaire, addition/soustraction de deux vecteurs, le vecteur opposé (le "-" unaire), et contiendra des routines de produit scalaire, produit vectoriel et normalisation (ainsi que norme, et norme au carré). Si vous ne maîtrisez pas la surcharge d'opérateurs, implémentez ces opérations sous forme de fonctions.

On définira une classe Ray, qui représentera un rayon de lumière : une origine et une direction.
On définira une classe Sphere : une origine, et un rayon.

Les rayons et les intersections

Les opérations fondamentales dans un raytracer sont la génération de rayons, et le calcul d'intersections entre un rayon et une primitive géométrique.

Étant donnée la configuration de notre caméra centrée au point C, de champs visuel "fov" = 60 degrés, et regardant fixement vers les z négatifs, générer un rayon vers le pixel (i, j) revient à générer une direction grace au vecteur :
V = (j-largeur/2+0.5, i-hauteur/2+0.5, -hauteur/(2*tan(fov/2.)))

puis de le normaliser V ← V / norme(V), ainsi qu'une origine, qui est le point C.
Un peu de trigonométrie du collège vous permettra de retrouver le pourquoi du -1/(2*tan(fov/2.)). Attention à convertir le fov en radians pour pouvoir utiliser la fonction tan() !

Une fois le rayon généré, on peut ensuite tester si ce rayon intersecte une sphère (commençons par une seule sphère, et ignorons les murs et le sol).

Pour cela, on réalise que les points d'intersections entre un rayon et une sphère, s'ils existent, résolvent à la fois l'équation de la sphère, et l'équation du rayon.
L'ensemble des points P sur la sphère de centre O et rayon R est donné par :

$$||P - O||^2 = R^2$$

L'ensemble des points P décrits par le rayon de lumière s'écrit sous la forme :

$$P = C + t.V$$

où V est le vecteur directeur du rayon de lumière, C son origine, et t un paramètre le long du rayon. Bien que ce ne soit pas strictement nécessaire, on considérera toujours que les vecteurs directeurs (ainsi que les normales aux objets) sont unitaires (de norme 1).

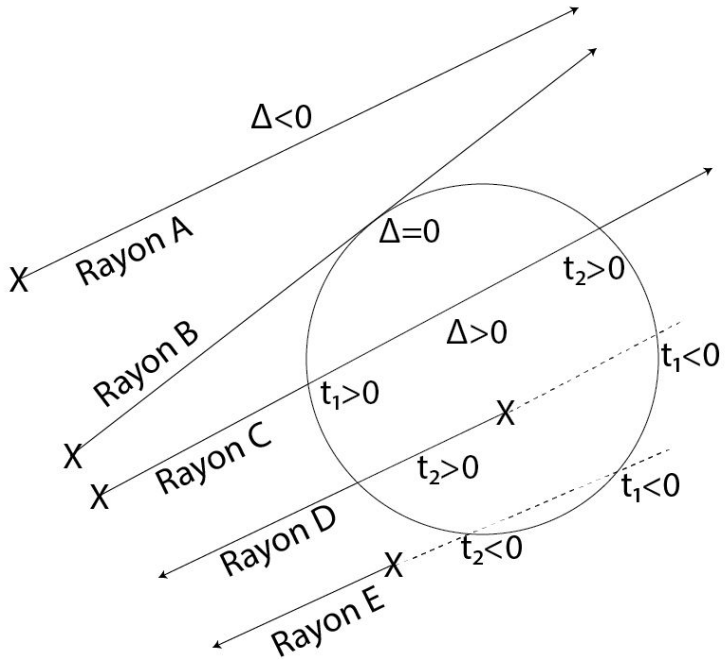
Les points satisfaisant ces deux équations sont donc donnés par l'ensemble des t tels que :

$$||C + t.V \ - \ O||^2 \ = R^2$$

En utilisant l'identité $||a + b||^2 = \ < a + b, \ a + b > \ = \ ||a||^2 \ + 2 < a, b > + ||b||^2$ on obtient le polynôme en t suivant :

$$t^2 \ + \ 2 \ < V , \ C - O > \ t \ + \ ||C - O||^2 - R^2 \ = \ 0$$

où j'ai considéré $||V||^2 = 1$
Il s'agit d'un polynôme de degré deux, qui peut avoir 0 ou deux solutions (ou deux solutions dégénérées en une seule), selon son discriminant Δ. En considérant toujours la solution t_1 inférieure à la solution t_2 (par exemple avec $t_1 = (- b - \sqrt{\Delta})/(2a)$ et $t_2 = (- b + \sqrt{\Delta})/(2a)$), on obtient les cas des figure ci-après :

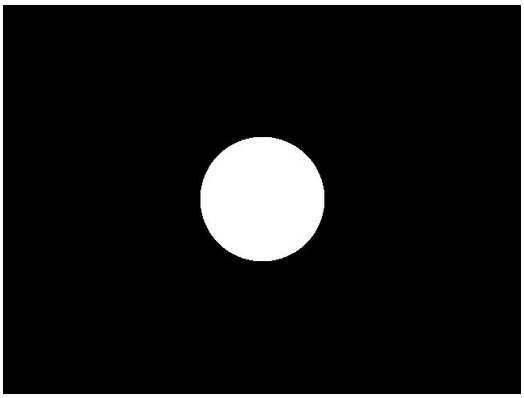


Dans cette figure, le rayon A n'intersecte pas la sphère car son discriminant est négatif. Le rayon B intersecte la sphère en un seul point car son discriminant est nul. Les rayons C, D et E intersectent deux fois la sphère, et doivent donc retourner l'un des points d'intersection. L'origine du rayon C est à l'extérieur de la sphère, et ses deux solutions sont positives : l'intersection la plus proche (t_1) est celle qui nous intéresse. En revanche, le rayon D démarre depuis l'intérieur de la sphère : seul t_2 est positif et nous intéresse (l'autre intersection se situe "derrière" le rayon). Le rayon E démarre en dehors de la sphère et ses deux solutions sont négatives : cela veut dire que les deux intersections se trouvent derrière la caméra, et il n'y a donc pas d'intersection qui nous intéresse : on considérera qu'il n'y a pas d'intersection du tout. Lorsqu'il y en a une, on récupère le point d'intersection P par $P = C + t.V$

Étant donnés ces calculs, on est en mesure de produire notre premier programme dont l'algorithme est le suivant :

```
Définition de la sphère
Pour chaque pixel (i,j)
        Générer un rayon vers le pixel (i,j)
        Si le rayon intersecte la sphère
                pixel(i,j) ← blanc
        Sinon
                pixel(i,j) ← noir
        Fin Si
Fin Pour
```

On obtiendra l'image suivante :

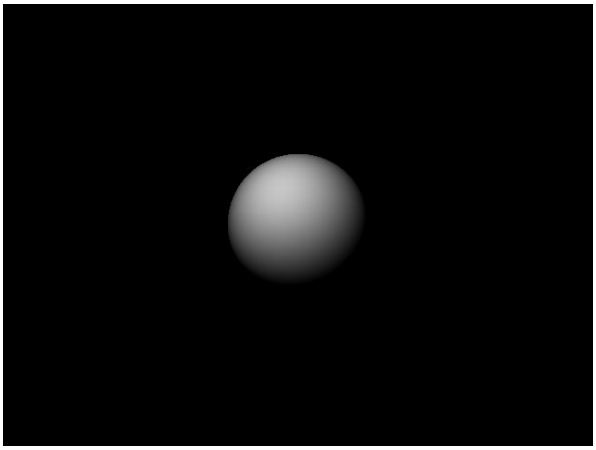


La lumière

Cette image manque de relief. Ceci va être apporté par l'éclairage. En considérant le matériau comme diffus (i.e., du plâtre), l'intensité du pixel est calculé comme :

pixel(i,j) ← max(0, $\vec{l}.\vec{n}$)*I/ d^2 [1] où $\vec{l}$ est un vecteur unitaire qui part du point d'intersection P et se dirige vers la source de lumière L, $\vec{n}$ est la normale de la sphère au point d'intersection, I est l'intensité de la lumière et d est la distance du point d'intersection à la lumière.

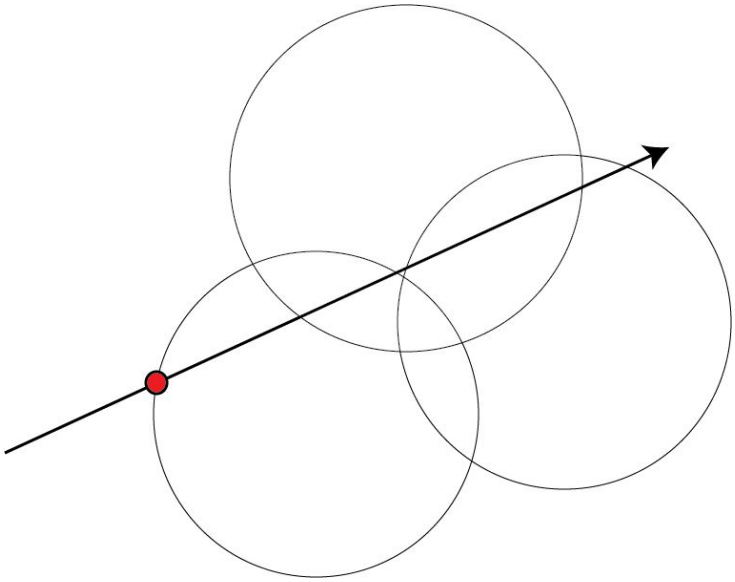
On obtient alors :



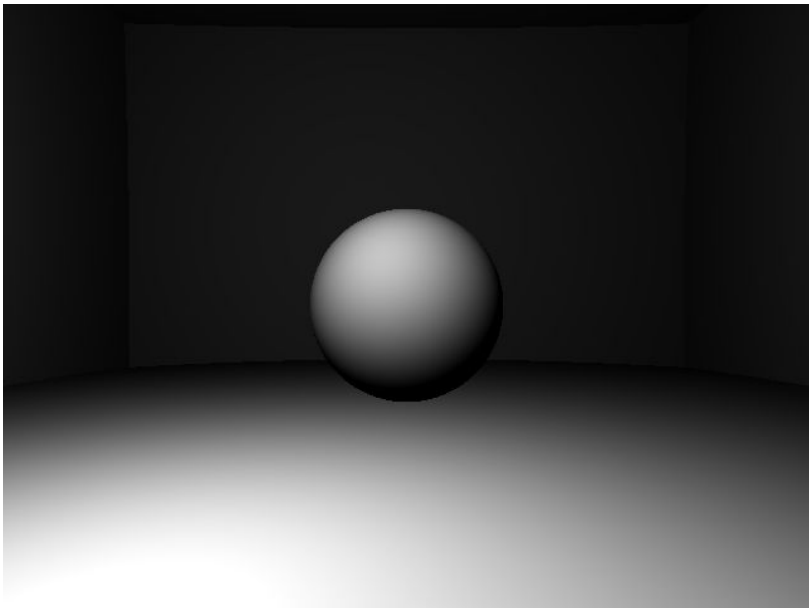
La suite va consister à pouvoir gérer plusieurs sphères. Pour ce faire, on va créer une classe "Scene", qui contiendra un tableau de sphères (std::vector<Sphere>). De la même manière que la classe Sphere possédait une méthode d'intersection avec un rayon de lumière, la classe Scene aura aussi une méthode similaire. La méthode d'intersection d'un rayon avec une Scene

[1] Note: Nous verrons par la suite qu'il faut diviser ce résultat par pi. Nous supposerons pour l'instant que ce facteur pi est inclus dans l'intensité de la lampe I.

sera défini comme l'intersection la plus proche de l'origine du rayon parmi les intersections avec toutes les sphères de la scène. Cette intersection a été matérialisée en rouge dans le dessin suivant :

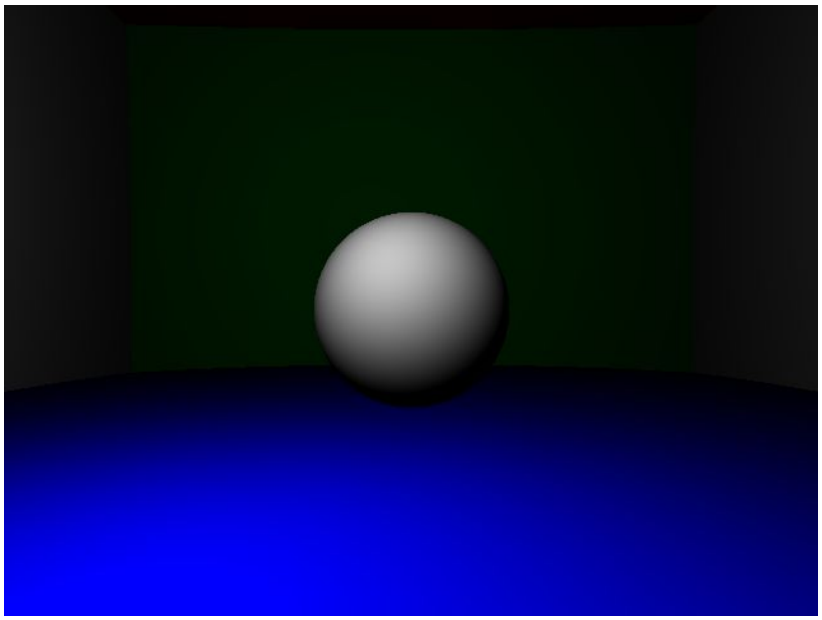


En faisant cela, et en utilisant la définition de la scène donnée en introduction (utilisant des sphères géantes pour modéliser des murs) nous obtenons l'image :



Par ailleurs, nous pouvons moduler les composantes rouge, verte et bleue de chaque sphère (il s'agit d'un simple facteur multiplicatif entre 0 et 1 pour chaque composante : la "couleur diffuse"), nous pouvons facilement colorer notre scène. Pour cela, on définira une classe

Material, qui contiendra une couleur diffuse (un Vector, par exemple), et qui sera assigné à chaque sphère. On obtient ainsi :



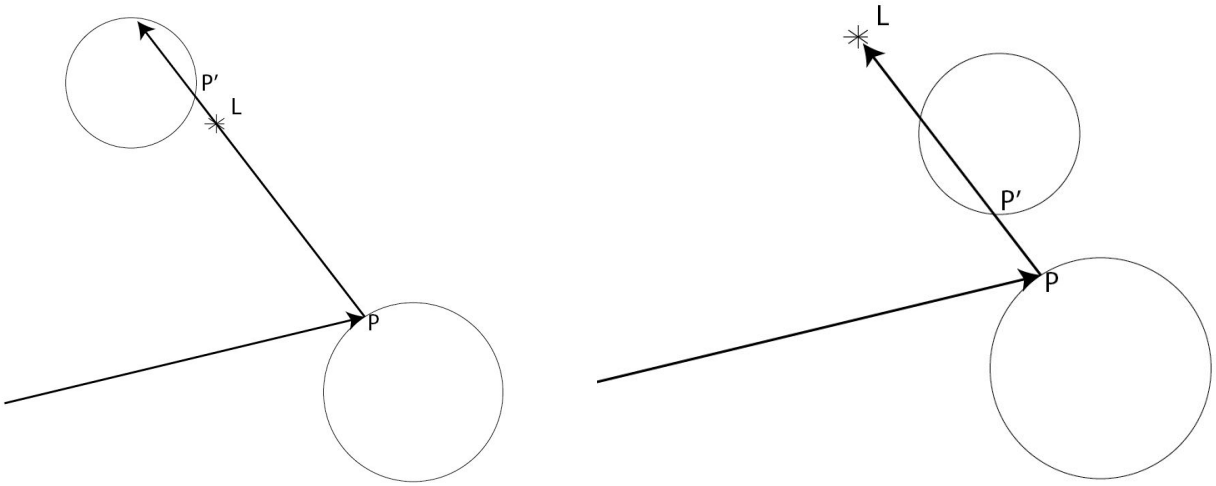
… et cela conclut la première séance.

Cours N°2

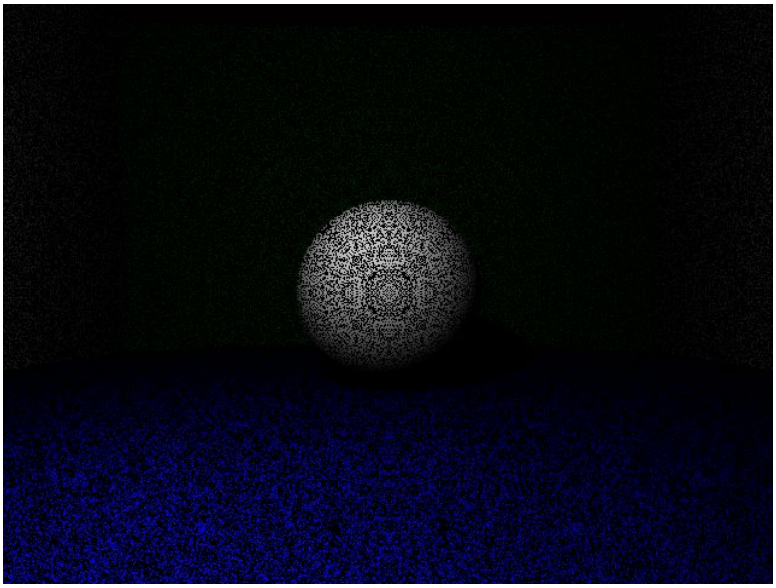
L'objectif de ce cours est de comprendre le fonctionnement des ombres portées, des surfaces "spéculaires pures" (i.e., des surfaces parfaitement miroir) et transparentes. Le point commun de ces effets est la nécessité de générer des rayons secondaires afin de les prendre en compte.

Les ombres portées

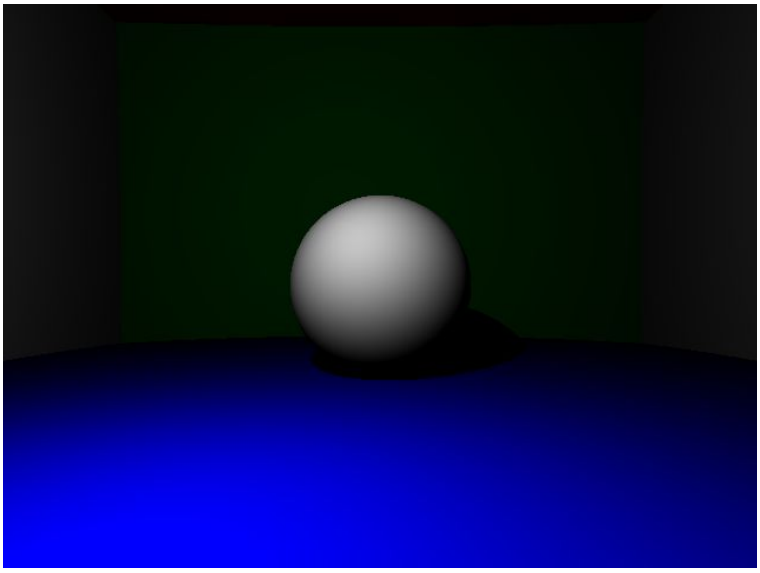
Le principe est simple. Lorsqu'une intersection P est trouvée, on génère un deuxième rayon de lumière partant de P et se dirigeant vers la source de lumière L, et on cherche une seconde fois les intersections avec les éléments de la scène. S'il y a une intersection, on détermine le point P' de cette intersection : si la distance de P à P' est inférieure à la distance de P à L, le point est ombré et sa couleur est donc noire.
Ci-après, la configuration de gauche ne produira pas d'ombre : la distance de P à l'intersection P' en direction de la lumière est supérieure à la distance entre P et L. En revanche, à droite, il y aura une ombre et le pixel sera noir.



Pour la scène ci-dessus, le résultat est :



Le bruit visible est dû à des imprécisions numériques. Pour les résoudre, il suffit de lancer le rayon secondaire depuis un point légèrement décollé de la surface, i.e., depuis le point P+epsilon*normale. On obtient alors :



Les surfaces spéculaires

ou surfaces “miroir”. Le principe est similaire, mais requiert maintenant une récursion.
Au lieu d’envoyer un rayon dans la direction de la lumière L, on va simplement réfléchir le rayon incident autour de la normale à l’objet, et renvoyer la couleurs de ce qui est réfléchit (la couleur

de ce nouveau rayon). Il se trouve que ce nouveau rayon pourrait taper dans un nouvel objet miroir, qui lui-même refléterait un nouvel objet etc.

L'algorithme pour déterminer la couleur d'un rayon partant depuis la caméra vers la scène est alors récursif, et donné par :

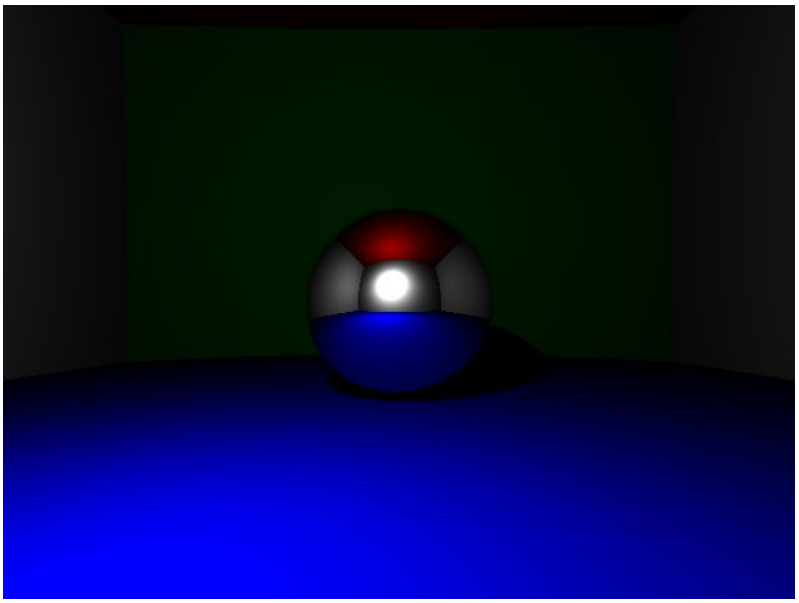
```
Couleur getColor(Rayon r, Entier numero_rebond)
        Couleur resultat ← noir
        Si la surface est spéculaire, et numero_rebond>0
                resultat ← getColor( réfléchir(r), numero_rebond - 1)
        Fin Si
        resultat ← resultat + partie diffuse.
Fin Fonction getColor
```

où l'on initialisera numero_rebond avec le nombre maximal de rebonds autorisés (par exemple, 5) afin d'éviter les cas où deux surfaces parallèles se font rebondir le même photon à l'infini.

La fonction "réfléchir" prend un rayon incident, et change sa direction afin qu'il soit réfléchit. Cela est donné par la formule $\vec{r} = \vec{i} - 2 < \vec{i}, \vec{n} > \vec{n}$ où $\vec{i}$ est le vecteur incident à la surface et $\vec{n}$ la normale à la surface. Vous devrez pouvoir la retrouver géométriquement assez simplement.

Le résultat peut éventuellement être modulé (multiplicativement) par une "couleur spéculaire".

On obtient alors l'image (ici, la couleur diffuse est noire, et la couleur spéculaire blanche) :



Les surfaces transparentes

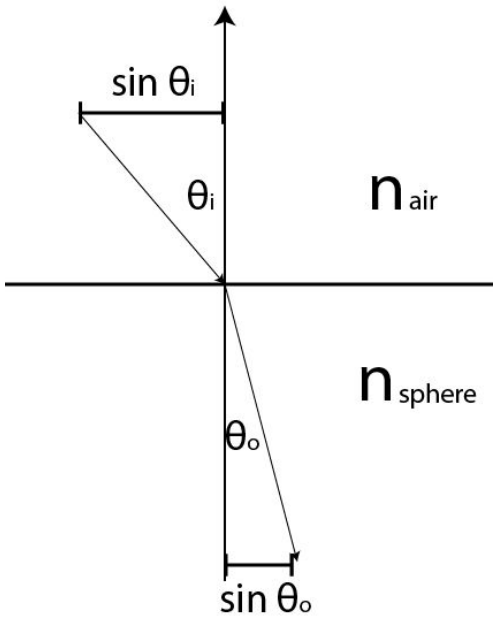
La gestion des surfaces transparentes est similaire, hormis l'appel à "réfléchir" qui est remplacé par une appel à "réfracter".

La formule permettant de réfracter un rayon est issu de la loi de Snell-Descartes :

$$n_{air} \sin\theta_i = n_{sphere} \sin\theta_o$$

où les n représentent les indices de réfraction de l'air et de la sphère, et θ_i et θ_o les angles incidents et réfractés.

Nous retrouverons la formule donnant le rayon réfracté grâce à la figure suivante :



$\vec{i} - <\vec{i}, \vec{n}>\vec{n}$ donne la composante tangentielle du rayon incident (i.e., on a supprimé sa composante selon la normale n).
On multiplie par $\frac{n_{air}}{n_{sphere}}$ cette composante afin d'obtenir la nouvelle composante tangentielle du rayon réfracté.
Le rayon réfracté aura donc une composante tangentielle : $\frac{n_{air}}{n_{sphere}}(\vec{i} - <\vec{i}, \vec{n}>\vec{n})$
On cherche maintenant sa composante normale. On sait que le rayon réfracté est unitaire, donc la composante normale est forcément $-\sqrt{1-\sin^2\theta_o}\ \vec{n}$ D'après Snell-Descartes, cela vaut $-\sqrt{1-(\frac{n_{air}}{n_{sphere}}\sin\theta_i)^2}\ \vec{n}$ ou encore $-\sqrt{1-\frac{n_{air}}{n_{sphere}}^2(1-<\vec{i},\vec{n}>^2)}\ \vec{n}$.

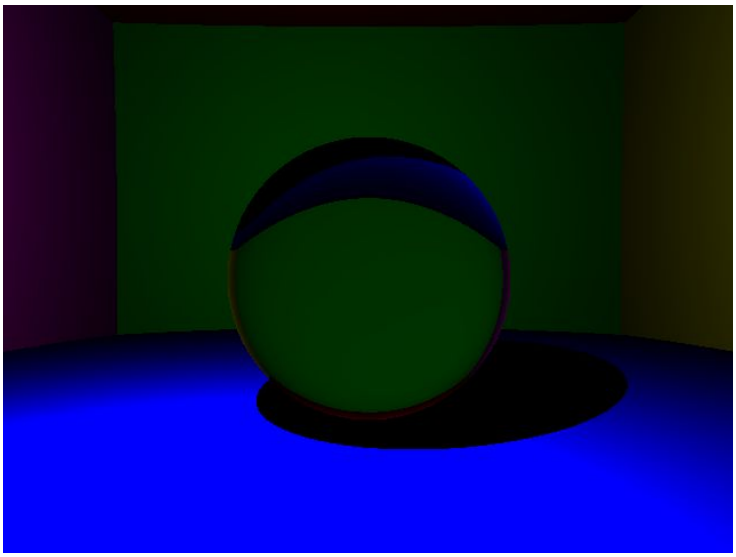
Bref, le rayon refracté s'écrit :

$$\vec{r} = \frac{n_{air}}{n_{sphere}}\vec{i} - \left(\frac{n_{air}}{n_{sphere}} < \vec{i}, \vec{n} > + \sqrt{1 - \left(\frac{n_{air}}{n_{sphere}}\right)^2 (1 - < \vec{i}, \vec{n} >^2)}\right)\vec{n}$$

Si la racine est complexe, cela veut dire qu'il n'y a pas de transmission, et la réflexion est totale.

A noter que cette formule ne vaut que si la normale est en effet orientée du même coté que le rayon incident, i.e., $< \vec{i}, \vec{n} > < 0$: le rayon rentre dans la sphère. Dans le cas contraire, la normale doit être inversée et les rôles de n_{air} et n_{sphere} sont inversés aussi.

Pour la scène ci-dessus (pour laquelle j'ai rapproché la sphère de la caméra, teinté les murs latéraux différemment et augmenté l'intensité de la lumière), nous obtenons le résultat ci-dessous.



A noter l'effet "lentille qui inverse l'image" : le sol se reflète en haut de la sphère, et les murs semblent inversés.

Ref supplémentaire : le livre Physically Based Rendering, de Matt Pharr et Greg Humphrey (ils ont obtenu un Oscar récemment pour leurs travaux, utiles pour les effets spéciaux au cinéma), qui est accompagné d'un moteur de rendu open source.

Cours N°3

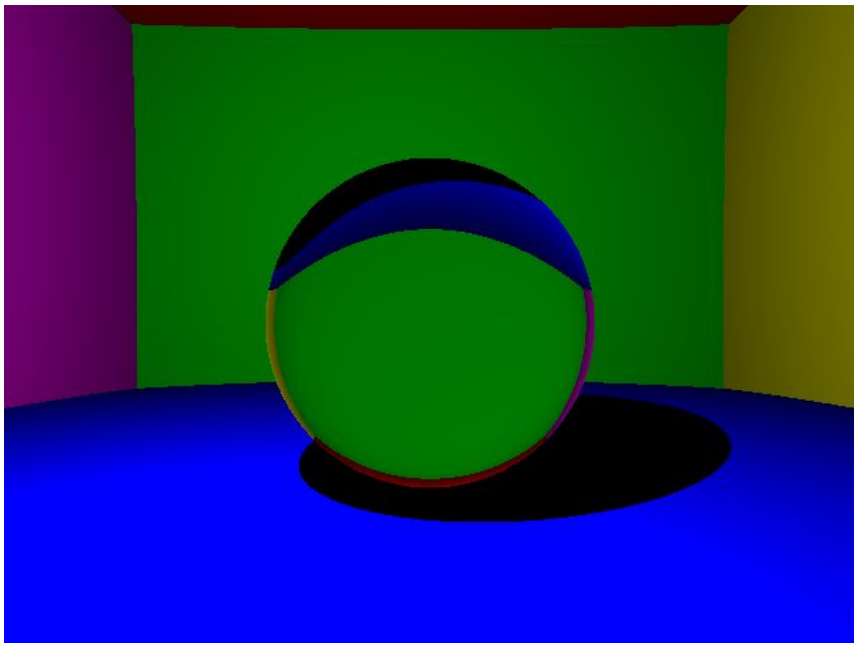
Nous allons voir comment gérer l'éclairage indirect, et, éventuellement les maillages. Mais d'abord, une note sur le capteur de la caméra et les écrans.

Correction gamma

Les intensités lumineuses affichées à l'écran ne semblent pas varier linéairement avec les valeurs qu'on lui donne : une intensité de 127 ne semble pas à mi-chemin entre du noir pur (0) et du blanc pur (255). En fait, les écrans ont un facteur gamma : une fonction de type luminosité = intensité^gamma, où gamma est souvent pris comme le coefficient 2.2.

Afin de compenser cette transformation, on encodera les images en leur appliquant la fonction x^(1./2.2) qui sera appliquée à chaque composante rouge, verte et bleue.

Ci-après la même image que précédemment, avec une correction gamma.



L’éclairage indirect

La prise en charge de l’éclairage indirect fonctionne de manière très similaire à celle des surfaces spéculaires. L’idée est, là aussi, de faire rebondir le rayon lumineux à travers la scène de manière récursive. Mais alors que la surface spéculaire ne reflétait que dans la direction “miroir”, les surfaces diffuses, elles, feront rebondir le rayon de lumière de manière aléatoire.

Mais d’abord, une note sur les BRDFs, l’équation du rendu et la méthode de Monte-Carlo.

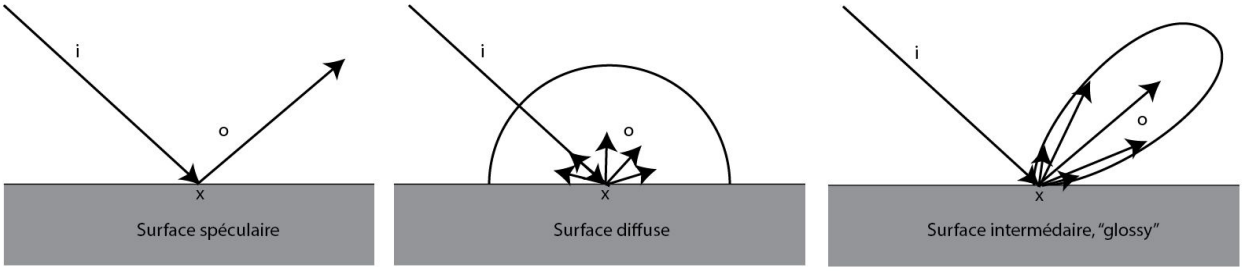
Les BRDFs

Les BRDFs, ou “Bidirectional Reflectance Distribution Function”, sont des fonctions caractérisant les matériaux. Elles décrivent la probabilité (*à un cosinus et absorption près*) pour un photon arrivant à la surface d’un objet avec une direction incidente $\vec{i}$ de rebondir dans la direction $\vec{o}$. Cette probabilité se note simplement $f(\vec{i},\ \vec{o})$ (par convention, on supposera le vecteur incident comme un vecteur partant de la surface, i.e., le vecteur $-\vec{i}$ par rapport aux dessins présentés précédemment).

Lors des séances précédentes, nous avons vu deux BRDFs, caractérisant deux types de matériaux différents :

- La BRDF spéculaire. Celle-ci donne une probabilité de 1 pour qu’un photon se reflète dans la direction “miroir” et une probabilité de 0 pour toutes les autres directions. Sa BRDF est donnée par $f(\vec{i},\ \vec{o}) = \frac{1}{\cos\theta_i}\ \delta(2 < \vec{i}, \vec{n} > \vec{n} - \vec{i} - \vec{o})$ où delta est un Dirac, et le cosinus deviendra plus clair par la suite.
- La BRDF diffuse. Celle-ci donne une probabilité uniforme pour qu’un rayon incident de direction quelconque se reflète dans une direction quelconque (sur une hémisphère complète). Sa BRDF est donc donnée par $f(\vec{i},\ \vec{o}) = \frac{1}{\pi}$ (qui est constante, et le facteur pi provient de la condition qu’une distribution de probabilité s’intègre à 1, et que $\int \cos(\theta)\ d\vec{i} = \pi$ où l’intégrale parcourt une hémisphère). Si le matériau absorbe un peu de lumière, cette constante peut être plus faible, voire différente pour chaque couleur (comme vous l’aviez programmé précédemment).

Mais il existe une multitude de BRDFs entre ces deux cas extrêmes, par exemple, des Gaussiennes centrées autour de la direction miroir. Nous ne les verrons pas dans ce cours, mais le schéma suivant illustre le concept. D’une manière générale, pour qu’une fonction $f(\vec{i},\ \vec{o})$ soit une BRDF il faut et il suffit qu’elle respecte le principe de reciprocité d’Helmholtz $f(\vec{i},\ \vec{o}) = f(\vec{o},\ \vec{i})$ et que $\int f(\vec{i},\ \vec{o})\ \cos\theta_i\ d\vec{i} < 1$ (voir ci-après). Nous avons aussi vu les matériaux transparents : il ne s’agit pas à proprement parler de BRDFs (“R” vient de reflexion, et non pas transmission), mais il s’agit d’un concept très similaire.

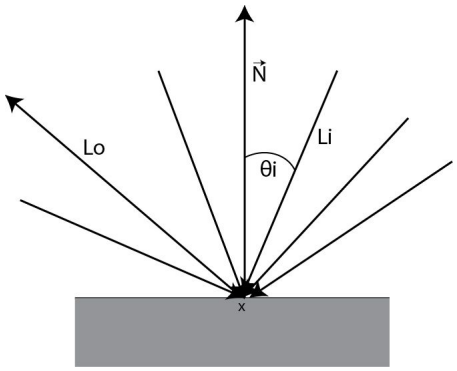


L'équation du rendu

C'est ultimement l'équation qui donne la couleur d'un pixel, mais elle s'exprime localement assez facilement.

$$L_o(x, \vec{o}) = E(x, \vec{o}) + \int f(\vec{i}, \vec{o})\, L_i(x, \vec{i})\, \cos\theta_i \; d\vec{i}$$

Ici, $E(x, \vec{o})$ est l'émissivité de la surface (nous n'avions considéré que des sources de lumière ponctuelles), $f(\vec{i}, \vec{o})$ est la BRDF, $L_i(x, \vec{i})$ l'intensité lumineuse arrivant au point d'intersection x, $L_o(x, \vec{o})$ l'intensité lumineuse qui en sort, et $\cos\theta_i$ le cosinus de l'angle entre le rayon incident $\vec{i}$ et la normale à la surface.



Le facteur $\cos\theta_i$ provient du fait que la surface collectera moins de lumière d'un rayon lumineux rasant celle-ci que d'un rayon lumineux d'incidence normale à la surface.

Cette équation dit simplement que la lumière sortante d'un point est la somme de toutes les lumières incidentes (ce qui inclut les sources de lumière indirectes comme les murs) multipliées par la BRDF (ainsi que le cosinus).
En fait, vous avez déjà résolu cette équation pour les surfaces spéculaires : rappelez vous que la BRDF d'une surface spéculaire est $f(\vec{i},\ \vec{o})\ =\ \frac{1}{\cos\theta_i}\ \delta(2 < \vec{i}, \vec{n} > \vec{n} - \vec{i} - \vec{o})$. Dans l'intégrale ci-dessus, le facteur en cosinus s'annule, et, si on suppose une surface non émissive (E=0), l'équation s'écrit:

$$L_o\,(x,\ \vec{o})\ =\ L_i(x,\ 2 < \vec{o}, \vec{n} > \vec{n} - \vec{o})$$

i.e., la somme ne se fait que sur une seule direction incidente : celle qui correspond à la réflexion de la direction sortante autour de la normale, et le résultat correspond donc à simplement aller interroger la couleur de cette direction réfléchie en envoyant un nouveau rayon.

L'équation du rendu nous permet aussi de comprendre la condition $\int f(\vec{i},\ \vec{o})\ \cos\theta_i\ d\vec{i}\ <\ 1$ pour qu'une fonction soit une BRDF : cela dit juste qu'une surface n'est pas censée renvoyer plus de lumière que ce qu'elle reçoit au total!

Les méthodes de Monte-Carlo

Il s'agit de méthodes stochastiques pour calculer des intégrales. On peut montrer d'une manière générale que calculer une intégrale $\int f(x)\ dx$ peut se faire en échantillonnant f aléatoirement.

Soient $x_0,\ x_1,\ ...,\ x_n$ des nombres suivant une loi de probabilité $p(x)$, on peut montrer que

$$\int f(x)\ dx\ =\ \frac{1}{n}\sum_{i=0}^{n} f(x_i)/p(x_i)\ +\ O(1/\sqrt{n})$$

En pratique, plus la distribution p est proche de f, plus la méthode sera précise.

Pour calculer l'intégrale de l'équation du rendu, on générera des directions aléatoires suivant une loi de probabilité aussi proche que possible du terme $f(\vec{i},\ \vec{o})\ \cos\theta_i$ (d'autres stratégies sont possibles!). L'intégrale s'approximera donc par :

$$\int f(\vec{i},\ \vec{o})\ L_i(x,\ \vec{i})\ \cos\theta_i\ d\vec{i}\ \approx\ \frac{1}{n}\sum_{i=0}^{n} f(\vec{i_i},\ \vec{o})\ L_i(x,\ \vec{i_i})\ \cos\theta_i\ /\ p(\vec{i_i})$$

D'une manière générale, pour une BRDF quelconque, générer une direction aléatoire avec une probabilité "proche de la BRDF" peut être très difficile. Heureusement, pour les surfaces diffuses, il s'agit d'échantillonner uniquement le facteur en $\cos\theta_i$ (la BRDF elle-même est constante), et une forme close est connue :

Pour les surfaces diffuses, on génère des coordonnées x, y, z par:

$$x = \cos(2\pi r_1)\sqrt{1 - r_2}$$
$$y = \sin(2\pi r_1)\sqrt{1 - r_2}$$
$$z = \sqrt{r_2}$$

où r_1 et r_2 sont des nombres aléatoires uniformes entre 0 et 1. Il vous faudra bien évidemment orienter ce rayon dans l'axe de la surface intersectée. On construira un repère en générant des vecteurs perpendiculaires à la normale à la surface, et on appliquera une formule de changement de repère. Attention à ne pas générer un vecteur du repère qui se retrouve colinéaire à la normale alors qu'il est censé lui être perpendiculaire!
La probabilité $p(\vec{i})$ s'écrit $p(\vec{i}) = \cos\theta_i / \pi$ où le facteur pi permet à cette loi de probabilité de s'intégrer à 1 sur l'hémisphère et θ_i est l'angle entre $\vec{i}$ et la normale. Puisqu'on a réussi à échantillonner de manière exacte la BRDF et non une vague probabilité s'y rapprochant, l'intégrale du rendu pour les surfaces diffuses échantillonnées avec la méthode ci-dessus s'écrit alors:

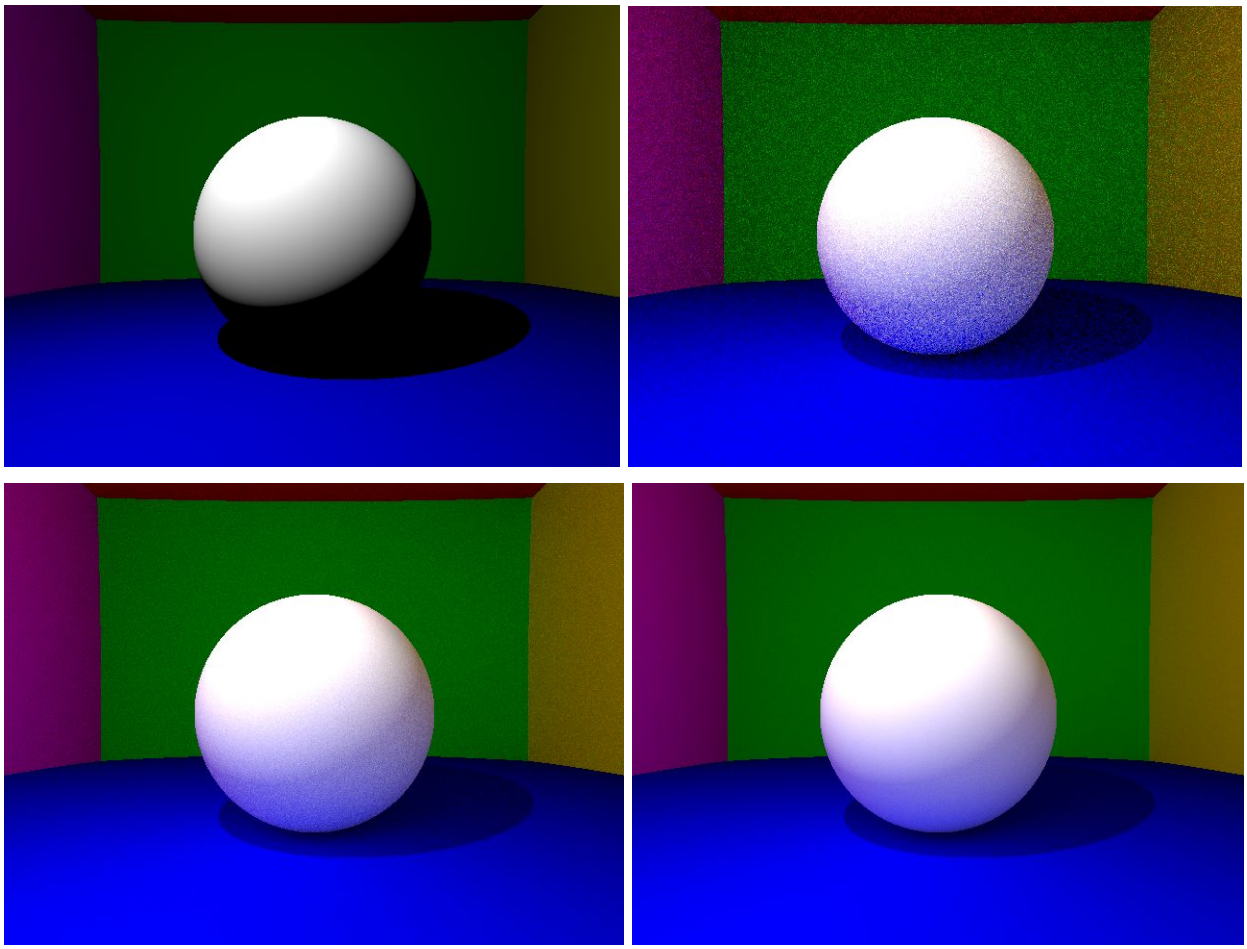
$$\int f(\vec{i},\ \vec{o})\, L_i(x,\ \vec{i})\ \cos\theta_i\ d\vec{i} \approx \frac{1}{n} \sum_{i=0}^{n} \rho_d\, L_i(x,\ \vec{i}_i)$$

où ρ_d est le coefficient diffus entre 0 et 1.
Les autre facteurs s'annulant (à noter aussi un facteur pi disparaissant ; voir la *Note 2* ci-après). On retrouve alors la même stratégie que pour les surfaces spéculaires : aller piocher les valeurs des intensités lumineuses, de manière récursive, mais en tirant des rayons de manière aléatoires plutôt que déterministe.

D'une manière générale, beaucoup de formules d'échantillonnage sont connues pour des BRDFs particulières, et sont compilées dans le Global Illumination Compendium de Philip Dutré.

Cette méthode converge pourtant assez lentement (en racine de n), et on s'attend à devoir tirer beaucoup d'échantillons pour obtenir une image nette. Idéalement, à chaque intersection, nous devrions tirer n rayons dans différentes directions, puis à chacune des intersections des rayons secondaires retirer n rayons etc. Cela nous conduirait à considérer n^r rayons, avec r le nombre de rebonds dans la scène.
Heureusement une autre stratégie est possible, et conduit au même résultat. Il suffit de n'envoyer qu'un seul rayon à chaque intersection de manière aléatoire. Cela conduit à une

image bruitée, évidemment. Mais le processus entier peut être répété afin d'obtenir une image nette.
La stratégie est donc de tirer n rayons par pixel, au lieu d'en tirer n par intersection.
Dans la scène ci-dessus avec une sphère diffuse (et un éclairage moins intense), avec n = 0, 10, 100, 1000, nous obtenons:



Note 1: le temps de calcul va s'accroître significativement, en particulier si vous n'avez pas fait attention aux performances (e.g., privilégiez toujours des calculs avec des normes au carrée plutôt que des normes, qui requièrent des calculs coûteux de racines). Par ailleurs, le calcul de chacun des pixels de l'image est indépendant. Il deviendra donc bénéfique d'utiliser le parallélisme de votre machine afin de réduire ce temps de calcul de manière importante.
Avec OpenMP, il suffit d'écrire avant votre boucle qui itère pour chaque ligne de l'image :

```
#pragma omp parallel for schedule(dynamic,1)
```

(dans CodeBlocks, utiliser -lgomp ; sous Visual Studio, activez le dans “”Language).

Note 2 : nous avions considéré la BRDF diffuse sous la forme $\rho_d \ max(0, <\vec{i}, \vec{n}>)$. Ceci ne respecte pas la condition de conservation de l'énergie des BRDFs. En fait, la BRDF diffuse s'écrit $\frac{\rho_d}{\pi} \ max(0, <\vec{i}, \vec{n}>)$. Cela ne dérangeait nullement pour l'éclairage direct (on considérait simplement une lumière pi fois plus intense pour compenser). Pour l'éclairage indirect, vous remarquerez probablement un éclairage bien trop important…. Ceci est dû à ce facteur pi, qu'il vous faudra corriger.

Note 3: l'utilisation de la fonction rand() du C++ pour générer des nombres aléatoires n'est pas recommandée. On préférera, si possible (i.e., si C++11 est géré), la version:

```
#include <random>
std::default_random_engine engine;
std::uniform_real_distribution<double> distrib(0,1);
double x = distrib(engine);
```

[optionnel à coder, mais à connaitre :]
Pour aller plus loin

Déplacement de la caméra

Pour notre caméra pointant vers (0,0,-1), nous générions des vecteurs:
V = (j-largeur/2+0.5, i-hauteur/2+0.5, -hauteur/(2*tan(fov/2.)))

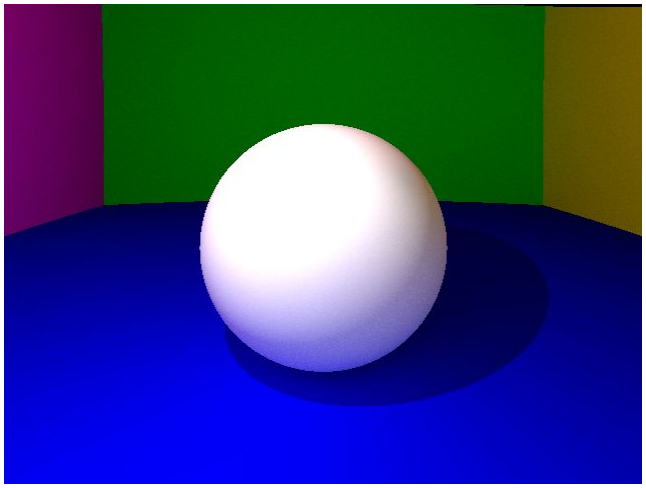
Pour tourner la caméra, il y a donc deux options simples:

- Ou bien on applique une transformation au vecteur V (e.g., une matrice de rotation)
- Ou bien on génère directement un vecteur dans la bonne direction. Pour cela, on stocke un vecteur “Direction” (la direction de visée) et un vecteur “Up” (la verticale de la caméra) unitaires, et on créé un vecteur “Right” (qui pointe vers la droite, avec un produit vectoriel). Il suffit alors de multiplier les coordonnées que nous avions par ces vecteurs. Ainsi :

V = (j-largeur/2+0.5)*Right + (i-hauteur/2+0.5)*Up + hauteur/(2*tan(fov/2))*Direction

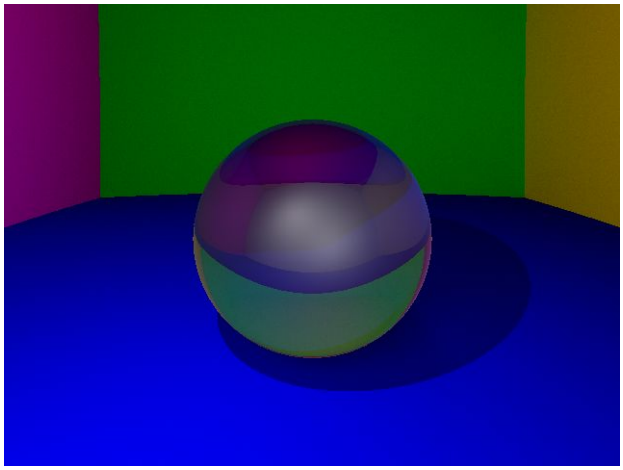
Vous pourrez ainsi créer une petite vidéo 3D où la caméra se déplace dans la scène et les boules aussi (par exemple, exportez les images individuellement et combinez-les avec un logiciel tel que VirtualDub).

La même scène avec une vue un peu plus plongeante:

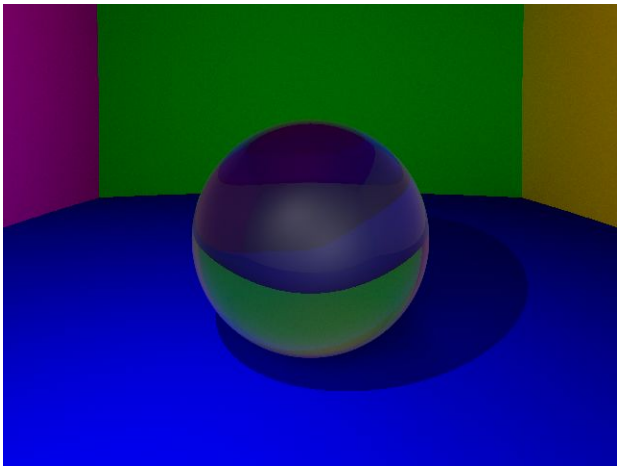


Les coefficients de Fresnel

Un matériau transparent réfléchit aussi la lumière, et les coefficients de réflexion et réfraction dépendent alors de l'angle d'incidence. On pourrait bien sur utiliser un coefficient de réflexion et réfraction constants. On obtiendrait l'image de gauche (ici, 0.9 pour la réfraction et 0.1 pour la réflexion). La prise en compte de manière physique de cette variation de coefficients se fait à travers les "coefficients de Fresnel", et on obtient alors l'image de droite.



Transparence et réflexion constantes



Avec coefficients de Fresnel

Les coefficients de Fresnel peuvent être coûteux à calculer, et une approximation efficace a été proposée par C.Schlick. Cette approximation donne le coefficient de réfraction par:

$$k_0 = (n_1 - n_2)^2/(n_1 + n_2)^2$$

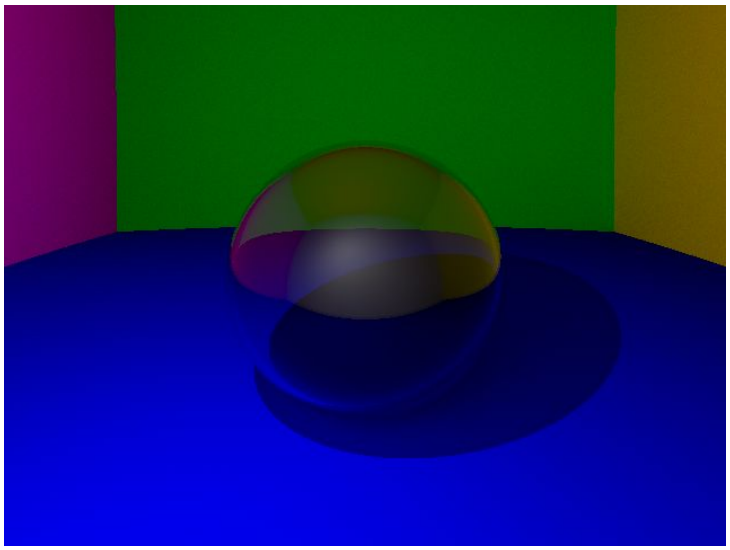
$$T = k_0 + (1 - k_0)(1 - <\vec{N}, \vec{i}>)^5$$

$$R = 1 - T$$

avec T le coefficient de réfraction, R le coefficient de réflexion, n_1 et n_2 les indices de réfraction des deux milieux, $\vec{N}$ la normale, et $\vec{i}$ le vecteur opposé au rayon incident si $n_1 < n_2$ ou la direction du rayon réfracté si $n_2 < n_1$.

Par ailleurs, multiplier par deux le nombre de rayons à chaque intersection (pour les rayons réfléchi et réfractés) n'est probablement pas une solution qui passe à l'échelle lorsqu'il y a de nombreux rebonds. On préférera dans ce cas choisir aléatoirement un comportement : soit réfléchi, soit réfracté, avec une importance proportionnelle à R et T. Moyenné sur de nombreux rayons, le résultat demeurera correct (voir échantillonnage de Monte-Carlo ci-dessus).

On peut aussi simplement simuler une sphère creuse en imbriquant deux sphères l'une dans l'autre, et en inversant artificiellement la normale de la sphère interne.



Les maillages

Nous avions des scènes constituées de sphères. Une scène était donc un tableau de sphères. Afin de diversifier nos scènes, nous considérerons une scène comme un tableau de pointeurs vers une classe abstraite "Object" (*Rappel*: une classe abstraite est une classe contenant au moins une méthode virtuelle pure, donc non implémentée: ce sera en particulier le cas de notre routine d'intersection qui sera adaptée au type d'objet). Dans ce cadre, la classe Sphere héritera de la classe Object, et implémentera la routine "Intersect" comme nous l'avions fait. Mais nous ajouterons une classe "Mesh" qui implémentera la routine "Intersect" différemment.

Représentation

On va représenter les maillages par des collections de triangles. Une représentation compacte et la plus couramment utilisée pour stocker un maillage est la donnée de:

- Une liste de sommets (= une liste de coordonnées 3D x, y, z)
- Une liste de triplets d'indices vers la liste de sommets (indiquant par exemple que le premier triangle consiste en les sommets 2, 18 et 53).
- Optionnellement: une normale par sommet ou par face, des coordonnées de texture par sommet ou une couleur par sommet ou par face, d'autres attributs...

Je vous invite à télécharger un fichier .obj léger sur http://tf3dm.com/ afin d'analyser cette représentation, et de faire un petite parser en C++ (de quelque lignes). Par exemple, ce modèle sera suffisamment simple (5484 triangles) pour pouvoir effectuer un rendu sans structure accélératrice (au moins pour de l'éclairage direct) (lien ci-dessus cassé ; ici une autre source ou ici un modele similaire).
En particulier, un .obj contient:

Les sommets préfixés par "v". Par exemple:
v 0.00268 0.02301 1.67202
est le premier sommet.
Des coordonnées de texture préfixées par "vt" (nous ne les utiliserons pas ici):
vt 0.40494 0.78046
Des normales préfixées par "vn" :
vn 0.07312 0.45169 0.88917
Les faces proprement dites, préfixées par "f" :
f 2/1/1 1/2/2 3/3/3
Ici, le format est:
f v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3
où v1 est l'indice des coordonnées 3D du premier sommet (attention, les indices commencent à 1 au lieu de 0!), vt1 l'indice des coordonnées de texture du premier sommet, vn1 est l'indice de la normale du premier sommet ; et respectivement pour les deux autres sommets du triangle (en effet, deux sommets peuvent se partager une même coordonnée de texture ou une même normale). Je vous fourni un code de chargement de fichier obj que vous pourrez utiliser tel quel ici.

Note: pour plus de simplicité, je vous conseille de normaliser vos objets, de telle manière qu'ils tiennent dans une boite plus ou moins unitaire et centrée à l'origine, puis de déplacer et/ou mettre à l'échelle les objets à votre convenance, afin de faciliter leur placement.
Aussi, le système de coordonnées peut différer, et peut nécessiter une symétrie ou rotation. Dans ce cas, n'oubliez pas d'appliquer cette même transformation aux normales!

Intersection Rayon - Plan

Pour déterminer le point d'intersection entre un rayon et un triangle, nous commençons pour trouver le point d'intersection entre le rayon et le plan supporté par ce triangle.
Un plan peut se définir par la donnée d'un point P0 lui appartenant et d'une normale N à celui-ci. L'équation de ce plan est alors donnée par l'ensemble des points P tels que le vecteur (P0, P) est perpendiculaire à N ; bref, que le produit scalaire suivant s'annule:

$$< P - P0,\ \vec{N} > \ = 0$$

De la même manière que pour la sphère, le point d'intersection rayon-plan est donnée par le point, s'il existe, qui satisfait à la fois la condition ci-dessus, ainsi que l'équation du rayon

$$P = C + t.\vec{V}$$

On a donc:

$$< C + t.\vec{V} - P0,\ \vec{N} > \ = 0$$

qui donne l'intersection:

$$t \ = \ - \ < C - P0,\ \vec{N} > \ / \ < \vec{V},\ \vec{N} >$$

Si le dénominateur s'annule, le rayon est tout juste parallèle au plan et il n'y a pas d'intersection. Dans le cas contraire, on produira une intersection seulement si t > 0.

Test d'appartenance à un triangle: les coordonnées barycentriques

Maintenant que nous connaissons le point d'intersection avec le plan du triangle, il nous faut vérifier si ce point se situe bien à l'intérieur du triangle ou bien à l'extérieur. S'il est à l'extérieur, c'est que le rayon n'a finalement pas intersecté le triangle.

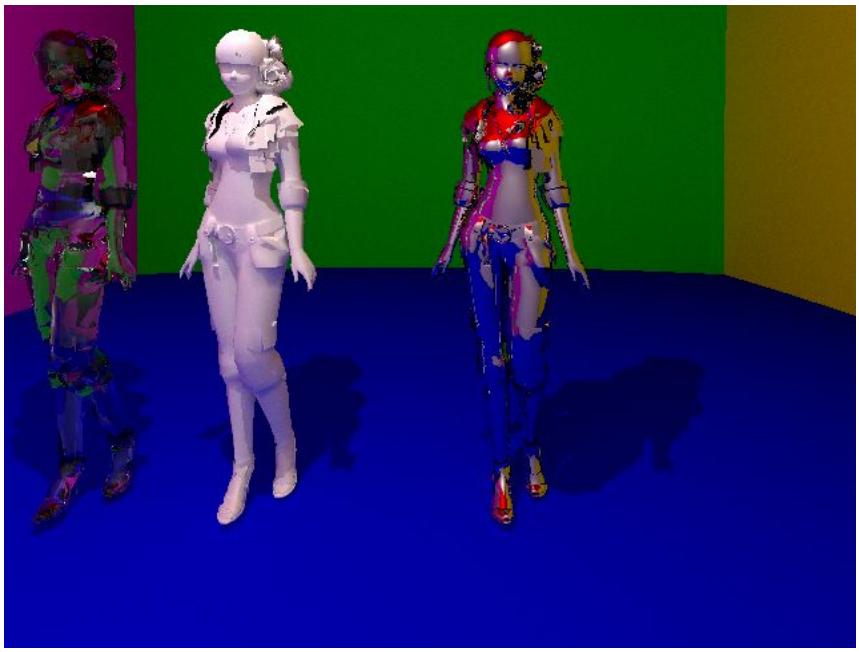
Pour cela, on fait appel aux coordonnées barycentriques d'un point. Les coordonnées barycentriques d'un point P en 2D (i.e., sur notre plan) par rapport à 3 points de ce plan (en l'occurence, notre triangle) $P_0,\ P_1,\ P_2$ sont la donnée de scalaires $\lambda_0,\ \lambda_1,\ \lambda_2$ tels que $\lambda_0 + \lambda_1 + \lambda_2 = 1$ et $\lambda_0 P\vec{P}_0 \ + \lambda_1 P\vec{P}_1 \ + \ \lambda_2 P\vec{P}_2 \ = \ \vec{0}$. Cela se généralise évidemment en dimension supérieure, mais nous n'en aurons pas besoin.
En fait, il s'agit simplement de déterminer la position du point P comme un barycentre des sommets du triangle. Si les coordonnées barycentriques sont toutes entre 0 et 1, alors P est une combinaison convexe des 3 sommets et se situe donc à l'intérieur du triangle.

Résoudre ce système à 2 inconnues (puisque $\lambda_2 \ = \ 1 - \lambda_0 - \lambda_1$) peut se faire avec la méthode de Cramer, qui dévoile aussi une autre interprétation géométrique de ces coordonnées en terme de ratio entre l'aire (signée) de triangles : $\lambda_0 \ = \ Aire(P P_1 P_2) \ / \ Aire(P_0 P_1 P_2).$

On trouvera un (pseudo) code efficace de test d'appartenance à un triangle en bas de page du site http://www.blackpawn.com/texts/pointinpoly/

Par ailleurs, bien que nous pouvons renvoyer la normale du triangle en la calculant avec un produite vectoriel afin de calculer l'éclairage, nous préférerons interpoler la normale fournie à

chaque sommet (le champs “vn” des fichiers .obj). L’interpolation peut se faire avec les coordonnées barycentriques calculées précédemment : $\vec{N} = \lambda_0 \vec{N_0} + \lambda_1 \vec{N_1} + \lambda_2 \vec{N_2}$



Les structures d'accélération

Alors que vous pouvez dès à présent gérer de petits modèles (de quelques milliers de triangles tout au plus), la plupart des modèles sont beaucoup plus complexes. Pour accélérer les rendus, on remarquera:

- Que si un rayon n’intersecte pas la boite englobante d’un objet, il n’intersectera pas l’objet du tout : on peut alors économiser tous les tests d’intersection
- Récursivement: qu’un objet est composé de plein de “sous-objets”, qui eux-même possèdent des boites englobantes. Ces sous-objets ne sont qu’un sous-ensemble des triangles de l’objet initial, et peuvent être déterminés de plusieurs manières (e.g., une subdivision régulière de l’espace, ou par un choix d’un plan qui répartit de manière la plus équilibrée possible les triangles d’un coté et de l’autre de celui-ci etc…).
- Que si une intersection existe avec un triangle dont la distance à l’origine du rayon est “t”, alors les boites englobantes dont l’intersection se situe à une distance plus grande peuvent être ignorées.

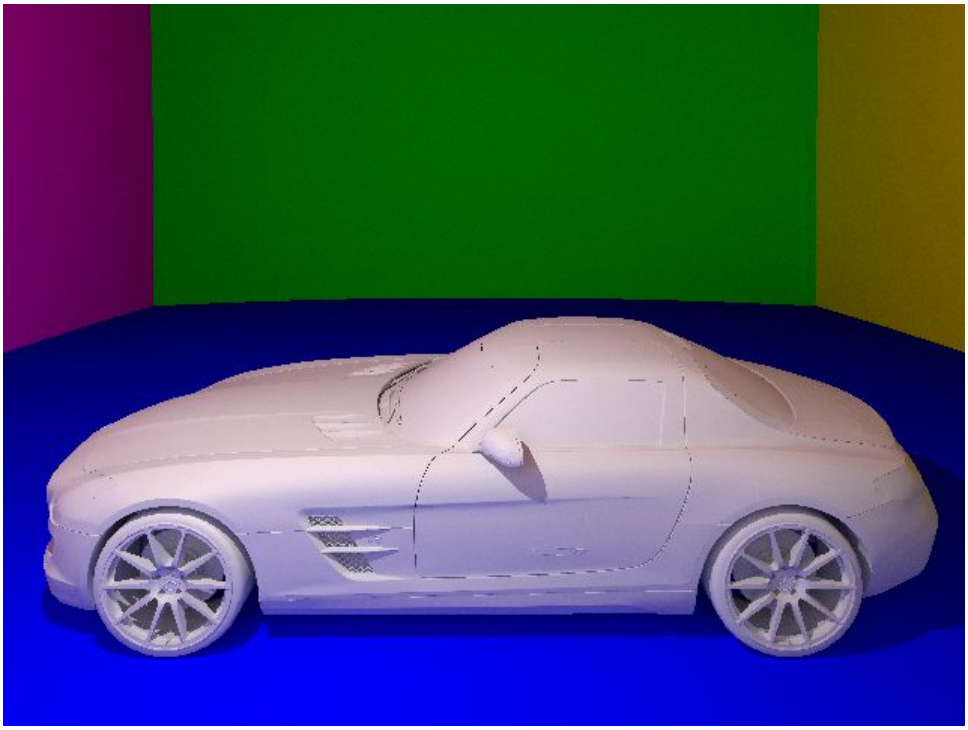
Au coeur de ces remarque est la routine d’intersection entre un rayon et une boite englobante (que l’on prendra alignée avec les axes du repère XYZ). Il s’agit de simples intersections rayon-plans avec des tests supplémentaires, et une implémentation efficace se trouve à :

http://tog.acm.org/resources/GraphicsGems/gems/RayBox.c

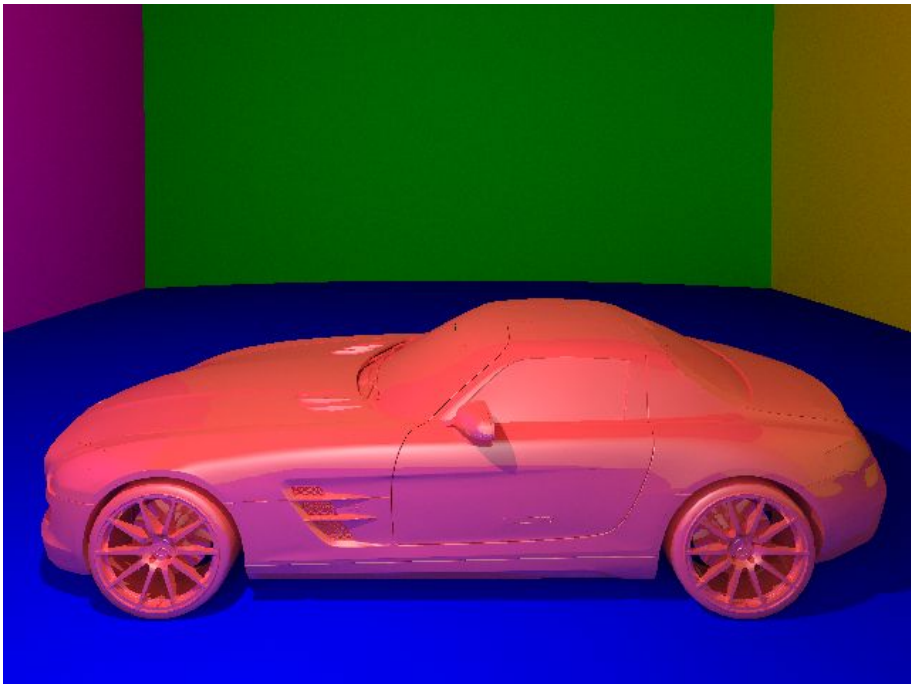
Notez qu'une liste de routines d'intersection entre primitive se situe à:
http://www.realtimerendering.com/intersections.html

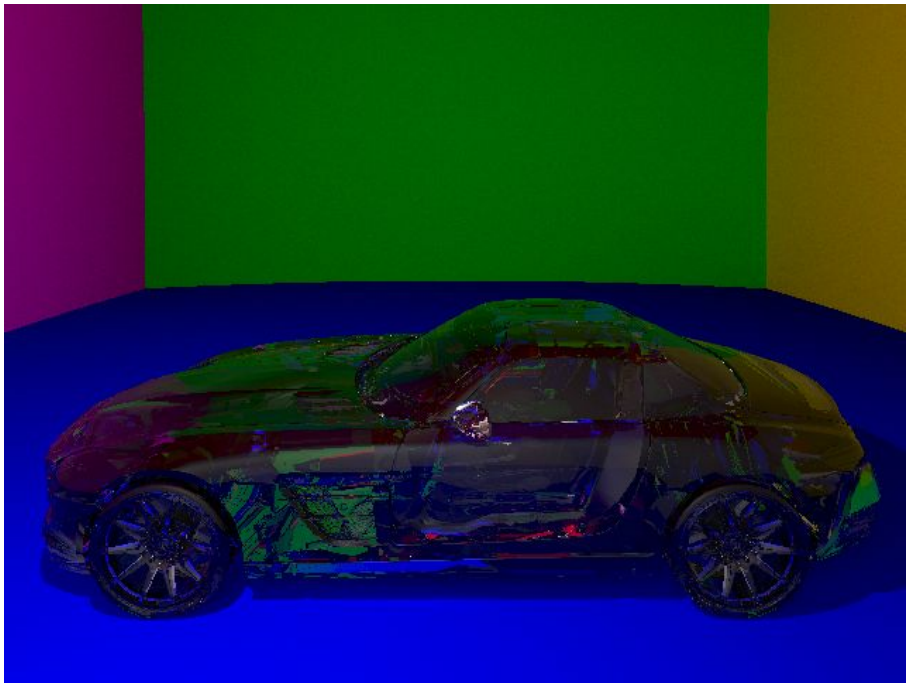
Tester si un triangle **n'est pas** inclus dans une boite de dimension donnée peut se faire par le théorème de séparation des convexes. Ce théorème dit que deux convexes ne se rencontrant pas sont séparés par un plan. On peut donc considérer les 6 plans de la boite et vérifier que le triangle ne se trouve pas entièrement du même coté des 6 plans et les 4 autres sommets de le boite de l'autre coté, et réciproquement, que les 8 sommets de la boite ne se trouvent pas du même coté du plan du triangle. Un code efficace qui fait cela se trouve à
http://fileadmin.cs.lth.se/cs/Personal/Tomas_Akenine-Moller/code/tribox3.txt
Il n'est cependant pas *nécessaire* d'utiliser cette routine pour créer une hiérarchie de boites englobantes.

Sur la voiture de 180339 triangles disponible à:
http://tf3dm.com/3d-model/mercedes-benz-sls-53003.html
nous obtenons:



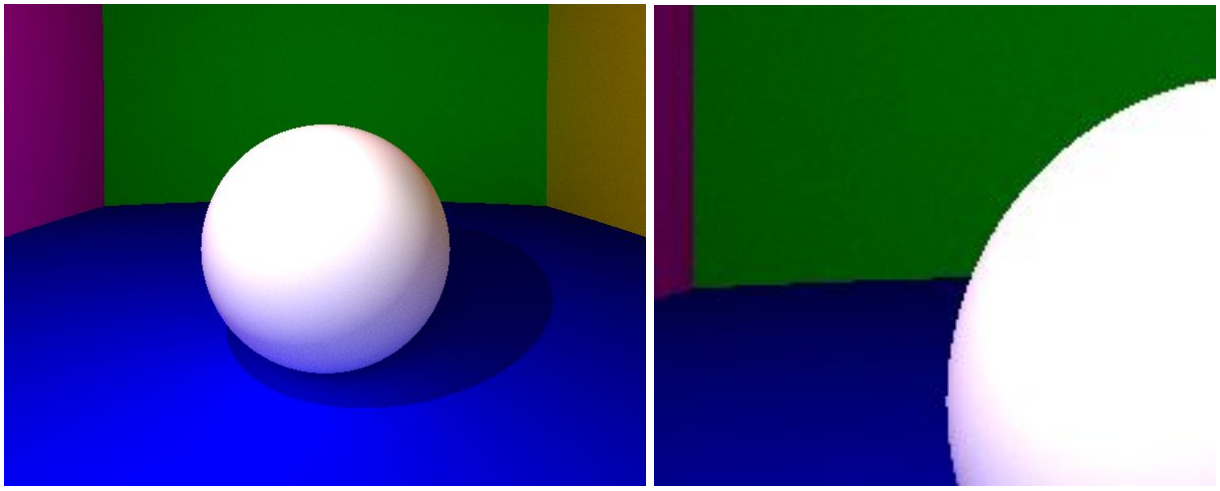
(1h de rendu avec 500 rayons par pixel, et une structure d'accélération assez naïve de deux grilles régulières imbriquées)
Ou en plus lent, avec du diffus+spéculaire, ou même de la transparence :



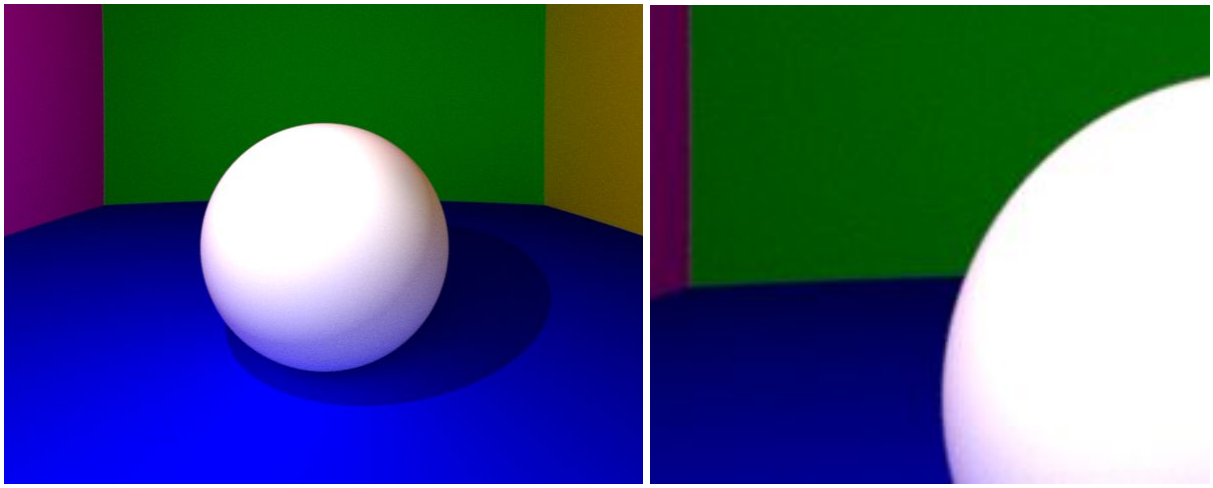


L’Anti-Aliasing

Jusqu'à présent nous envoyions nos rayons dans le centre de chaque pixel. Ceci résulte en un effet de crénelage (ou Aliasing) que l'on observe sur les contours des objets, et qui est très gênant dans les scènes animées. Par exemple, la sphère (zoomée) donne:



Alors qu'une technique d'Anti-Aliasing permet d'obtenir des arêtes plus douces.



Pour effectuer de l'antialiasing, plusieurs solutions:

- Le full-screen antialiasing : on génère une image 4 fois plus grande, et on la redimensionne avec un filtrage approprié à la fin. Ce n'est pas une technique très efficace.
- L'échantillonnage de rayons aléatoirement à l'intérieur de chaque pixel. Au lieu d'envoyer 500 rayons au centre de chaque pixel, on peut perturber la direction de ces rayons afin qu'ils recouvre la surface du pixel complet (voire de pixels voisins!). De plus, on souhaiterait pondérer d'avantage les rayons proches du centre du pixel que les rayons loin du centre.
  Pour cela, on pourra soit échantillonner uniformément autour du centre du pixel, et

pondérer le résultat (par exemple par une Gaussienne centrée au milieu du pixel), ou bien, on pourra directement générer un échantillonnage Gaussien centré au milieu du pixel, et nous n'aurons pas besoin de pondération (pour vous en convaincre, relisez la partie sur l'échantillonnage de Monte-Carlo).
Nous avons choisi cette dernière technique, et employé la méthode de Box-Muller pour générer des échantillons Gaussiens :

```
depth = H/(2*tan(fov*0.5));
Ray generateRay(int i, int j) {
	Vector right = cross(direction, up);

	double x = drand48(), y = drand48(), R=sqrt(-2*log(x));
	double u =  R*cos(2*3.1416*y)*0.5;
	double v =  R*sin(2*3.1416*y)*0.5;
	Ray r( position, (j-W/2+u-0.5)*right + (i-H/2+v-0.5)*up + depth*direction);
	r.direction.normalize();
	return r;
};
```

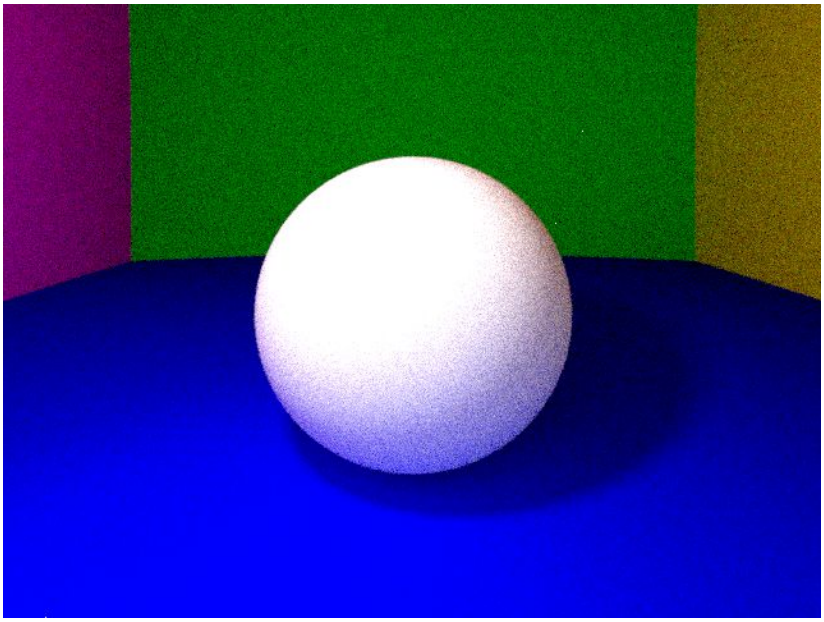
J'ai ici multiplié u et v par 0.5 afin de générer une Gaussienne d'écart-type 0.5 et éviter une image trop floue.

Des techniques adaptatives existent aussi, et cherchent à échantillonner d'avantage lorsqu'il y a des arêtes (ou du contenu haute fréquence en général).

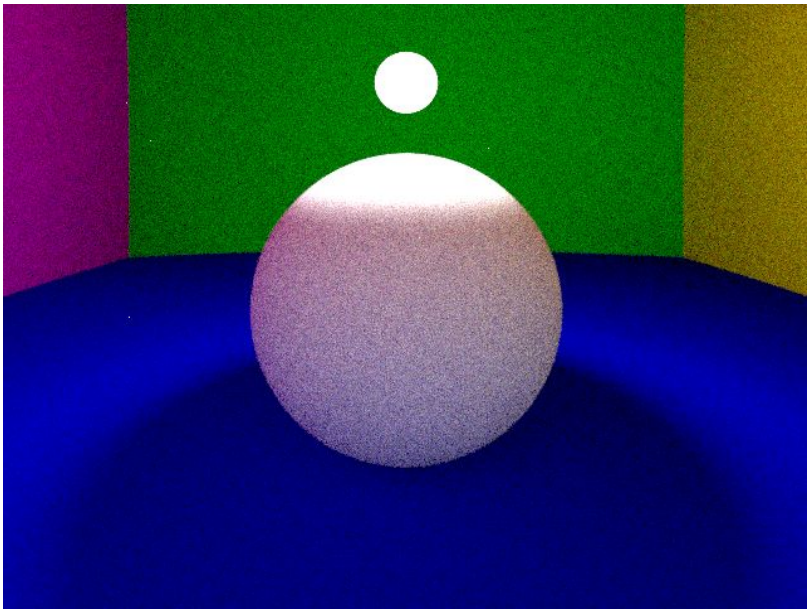
Les lumières étendues et les ombres douces

Nous avons considéré jusqu'à présent une seule lumière ponctuelle, ce qui produit des ombres franches. Les ombres douces sont produites par des lumières étendues sur une surface complète.
Nous pourrions donc simplement supprimer notre lumière ponctuelle, ajouter un paramètre "émissivité" à chaque objet, et avoir dans notre scène un triangle (ou des objets) très émissifs, et ainsi ne rien changer à notre moteur de rendu. Cela n'est pas tout à fait satisfaisant: en effet, si notre triangle émissif est relativement petit, notre échantillonnage des surfaces diffuses aura très peu de chances de produire des rayons qui impacterons ces triangles émissifs. Un pixel sera alors coloré si au moins un des chemins complets aura atterri sur la lampe après de multiples rebonds aléatoires ; autrement, il restera noir. La scène sera alors très bruitée.

La scène précédente donne le rendu suivant montrant des début d'ombre douce, mais qui possède déjà 10000 rayons par pixel et qui a mis près d'une heure à calculer. C'est dommage pour une simple sphère!



Même scène mais j'ai déplacé la lumière afin qu'elle soit dans le champs visuel, pour avoir une idée de sa taille.



L'idée est alors d'échantillonner soit dans la direction de la lumière, soit dans la composante diffuse comme nous l'avions fait, avec une certaine probabilité.
Si on note α la probabilité d'échantillonner la lumière plutôt que la composante diffuse, la probabilité totale que l'on échantillonnera est donnée par :

$$p(\vec{i}) = \alpha\, p_{lampe}(\vec{i}) + (1-\alpha)\, p_{diffus}(\vec{i})$$

où p_{diffus} est la probabilité associée à la composante diffuse, et est donnée (rappel) par : $p_{diffus} = \cos\theta / \pi$. Une autre solution est de considérer qu'on sépare l'hemisphère d'intégration

en deux parties: la partie qui contient l'éclairage direct (on tirera alors un rayon vers la sphère lumineuse) et la partie qui ne contient pas d'éclairage direct (on tirera alors un rayon aléatoire dans tout l'hémisphère, mais s'il tombe sur la sphère lumineuse, sa contribution devra être nulle) -- c'est l'approche discutée dans la vidéo.
Nous allons discuter de $p_{lampe}(\vec{i})$ liée à l'échantillonnage de la lampe.

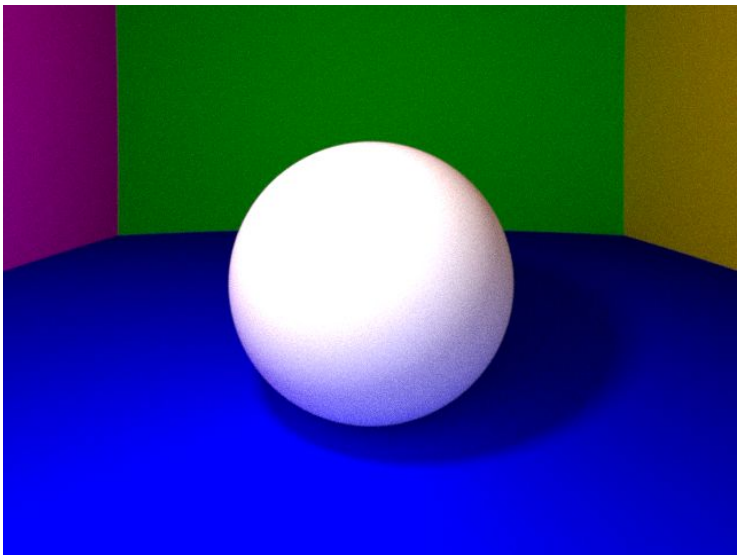
Pour générer des échantillons dans la direction des lumières, nous pourrions directement échantillonner la surface des lumières. Il faudrait faire attention alors à un terme de changement de variable (nous échantillonnerions en fait des positions et non des directions, et un nouveau terme en cosinus et une distance au carrée apparaîtraient…). A la place, nous allons générer des vecteurs (i.e., réellement des directions et non des positions) en direction des lampes, et pour cela, nous allons considérer que nos lampes sont sphériques.
Sous cette condition, l'échantillonnage en direction d'une sphère est décrite page 31 du cours http://www.cs.rutgers.edu/~decarlo/readings/mcrt-sg03c.pdf (ou page 19 du Global Illumination Compendium)

L'estimateur de Monte-Carlo de l'intensité réflechie par une surface diffuse est alors:

$$\int f(\vec{i},\ \vec{o})\ L_i(x,\ \vec{i})\ \cos\theta_i\ d\vec{i}\ \approx\ \frac{1}{n}\sum_{i=0}^{n}\frac{\rho_d}{\pi}\ L_i(x,\ \vec{i}_i)\ \cos\theta_i\ /\ (\alpha\ p_{lampe}(\vec{i})\ +\ (1-\alpha)\ p_{diffus}(\vec{i}))$$

lorsque les échantillons sont générés en générant avec une probabilité α la lampe avec la méthode ci-dessus et $(1-\alpha)$ le diffus avec la méthode initialement vue en cours.
Note: les rayons générés sont tous les dans la même hémisphère (/aux erreurs numériques près!), ce pourquoi j'ai noté $\cos\theta_i$ au lieu de $max(0,\ \cos\theta_i)$. Cependant, les erreurs numériques peuvent requérir la fonction max pour éviter des contributions négatives!.

Sphère obtenue après moins de 5 minutes de calcul avec 1000 rayons par pixel, dont 80% de diffus ($\alpha = 0.2$) :



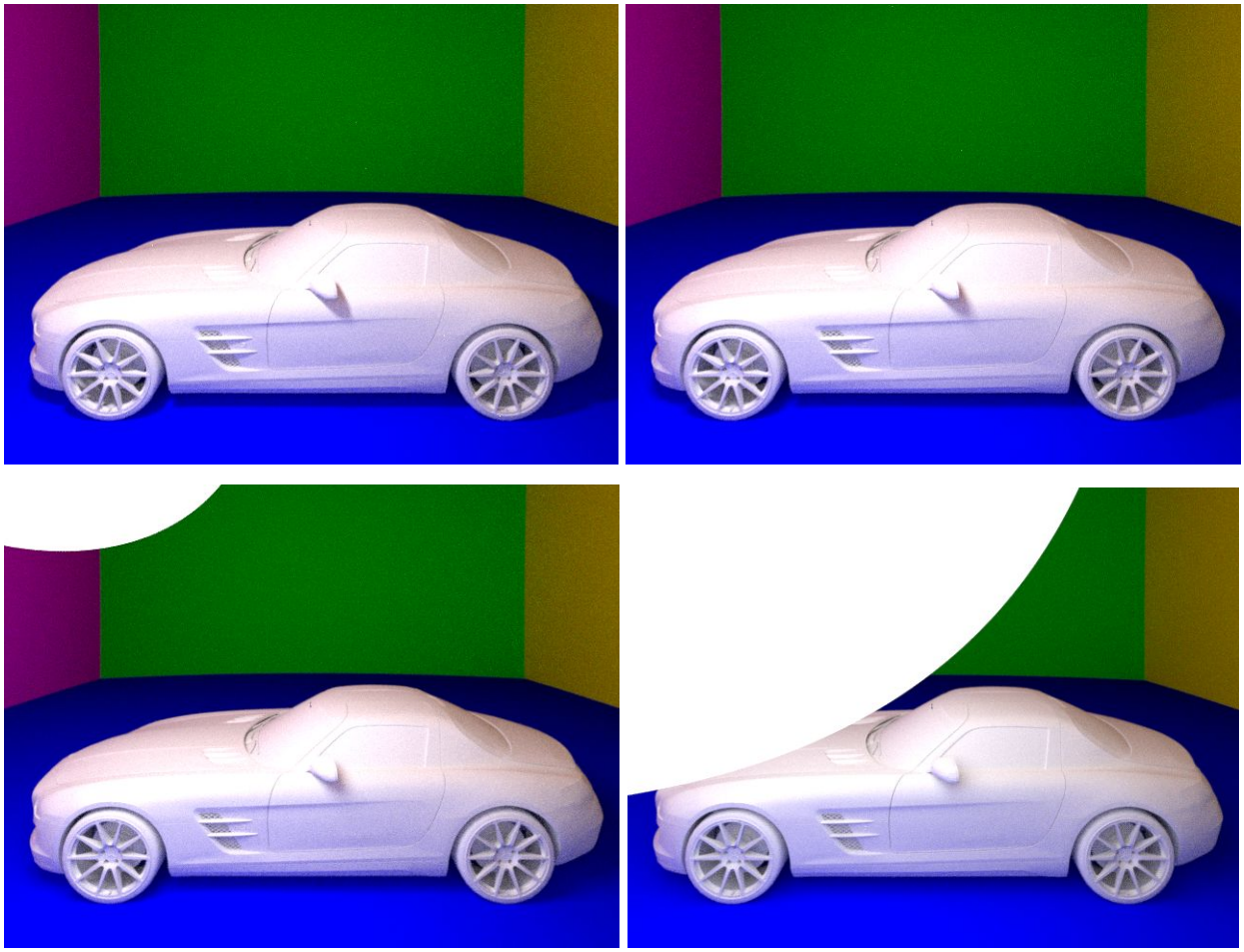
Note fonction getColor s'écrit dorénavant:

```
Couleur getColor(Rayon r, Entier numero_rebond)
        Couleur resultat ← noir
        Si la surface est spéculaire, et numero_rebond>0
                resultat ← getColor( réfléchir(r), numero_rebond - 1)
        Fin Si
        Si la surface est transparente, et numero_rebond>0
                resultat ← resultat + getColor( réfracter(r), numero_rebond - 1)
        Fin Si
        Si la surface est diffuse, et numero_rebond>0
                Si random < alpha
                        rayon_reflechi = générer direction aléatoire vers la sphère lumineuse
                Sinon
                        rayon_reflechi = générer direction aléatoire avec la formule du diffus
                Fin Si
                pdf = alpha*p_lampe(rayon_reflechi) + (1-alpha)*p_diffus(rayon_reflechi)
                resultat ← resultat  + coeff_diffus/pi*getColor(rayon_reflechi,
                        numero_rebond-1)*cos(theta)/pdf
        Fin Si
        Si la surface est émissive
                resultat ← resultat + émissivité
        Fin Si
Fin Fonction getColor
```

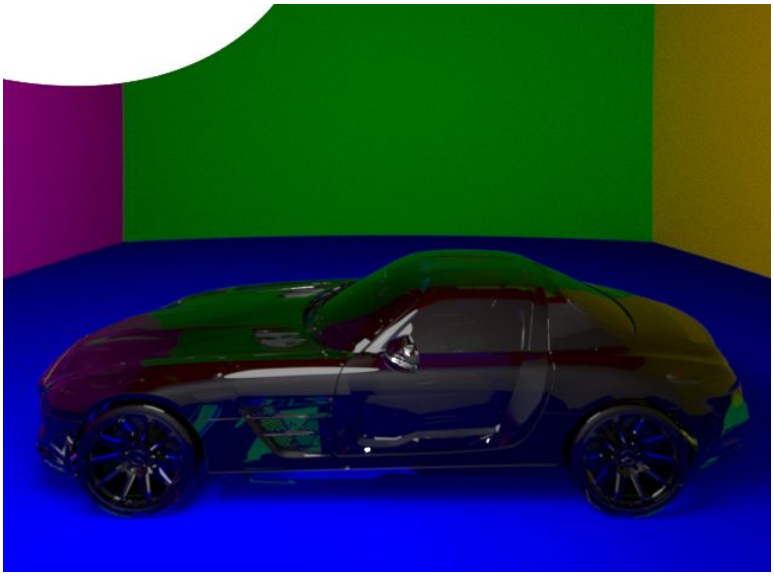
A noter qu'il n'y a plus besoin de tracer des rayons vers la source de lumière ponctuelle pour tester la présence d'ombre (puisqu'il n'y a plus de sources ponctuelles!), ni de diviser par un facteur en distance au carrée à la source lumineuse. Aussi, un rayon tel que $< rayon, \vec{N} > \ < 0$ n'a pas besoin d'être évalué: sa contribution est nulle.

On peut aussi améliorer le pseudo-code précédent en générant un nombre aléatoire indiquant si le rayon sera réfracté, réfléchi, ou diffus, en fonction des coefficients de transparence, spéculaire et diffus.

Sur la voiture, rendue avec 500 rayons par pixel, alpha=0.5, avec des sphères qui illuminent de rayon 2, 4, 8 et 16, produisant des ombres de plus en plus douces:



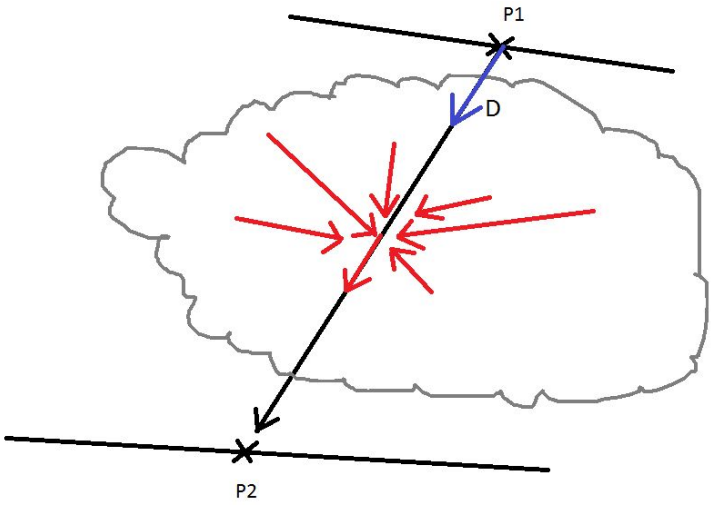
Avec la transparence, le Fresnel, l’anti-aliasing, et les ombres douces:



Les milieux participants

Un milieu participant est un milieu tel que l'intensité lumineuse émise par un point P1 dans une direction D n'est pas égale à l'intensité lumineuse reçue par le point P2, qui est la première intersection rencontrée le long de D. C'est en particulier le cas lorsqu'il y a du brouillard, de la fumée, ou de la diffusion lumineuse en général (de l'atmosphère par exemple).
Dans ce cas, la lumière est :

- d'une part absorbée par le milieu
- d'autre part, renforcée par des contributions secondaires



(désolé, mon Illustrator est en panne!)
Ici, j'ai représenté en bleu l'intensité lumineuse partant de P1 qui va être absorbée par le nuage, et en rouge les contributions secondaires qui vont s'ajouter.

L'intensité lumineuse arrivant au point P2 s'écrit alors :
$L(P_2) = T(s)\, L_o(P_1, \vec{D}) + L_v$
où T(s) est la fonction de transmittance évaluée au paramètre s qui est la distance de P1 à P2 (c'est l'atténuation de la lumière), $L_o(P_1, \vec{D})$ est l'intensité lumineuse partant de P1 dans la direction D et L_v est la contribution du milieu participant (les flèches en rouge).

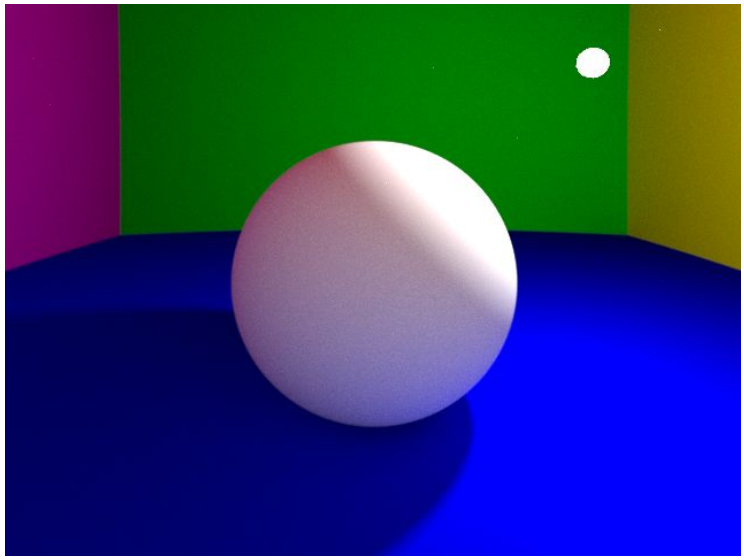
La partie la plus facile est la fonction de transmittance, qui s'écrit:

$$T(t) = \exp(-\int_0^t \sigma_t(x_w)\, dw)$$

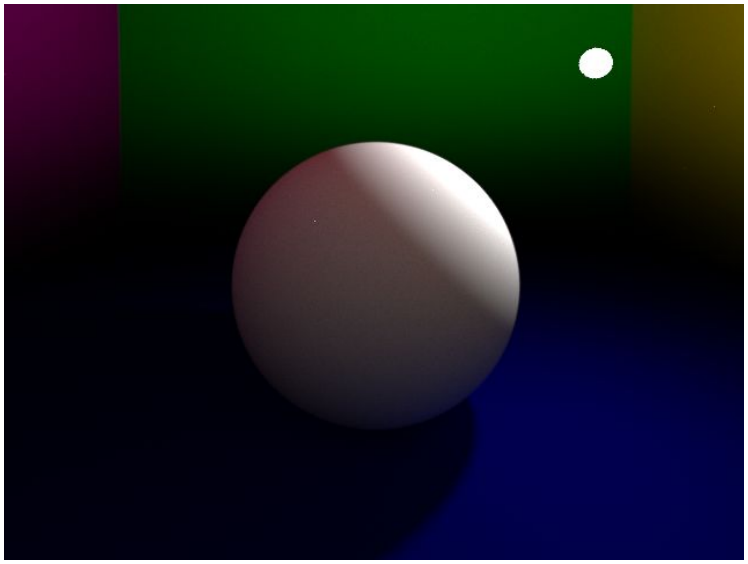
où x_w est le point $P_1 + w.\vec{D}$, et σ_t est le coefficient d'extinction du milieu, i.e., la densité du gaz. Bref, c'est juste la densité du gaz cumulée le long du rayon, à laquelle on applique la fonction exponentielle. Et il se trouve qu'il y a plein de densité de gazs pour lesquels cette intégrale a une forme close.

On notera notamment que si le gaz a une densité constante (un milieu homogène), on a donc $T(t) = \exp(-\alpha t)$. De manière similaire, si la densité du milieu est une exponentielle décroissante avec l'altitude (e.g., de l'atmosphère, du brouillard…), on a $\int_0^t \sigma_t(x_w) = \frac{\exp(-\lambda y_0)}{\lambda D_y}(1 - \exp(-\lambda D_y t))$ où D_y est la composante y du rayon, λ est la décroissance exponentielle du milieu, et y_0 est l'altitude initiale du rayon.

Effectuer un rendu avec seulement ce facteur d'atténuation permet de passer de l'image:



à:



(avec une atténuation exponentielle avec l'altitude)

Cette image est sombre, et ce qui manque est la contribution du milieu participant à l'éclairage. La formule est un peu plus difficile -- pas à comprendre, mais à évaluer.

$$L_v = \int_0^s \sigma_s(x_t)\, T(t) \int_{S^2} \rho(\vec{w},\ \vec{v})\, L(x_t,\ \vec{v})\, d\vec{v}\, dt$$

où σ_s est le coefficient de diffusion du milieu (qu'on pourra prendre exponentiel aussi par exemple), T est la fonction définie précédemment, $\rho(\vec{w},\ \vec{v})$ est la fonction de phase (l'équivalent de la BRDF mais pour une particule de gaz : c'est la probabilité qu'un photon incident sur cette particule dans une direction $\vec{w}$ soit réfléchi dans la direction $\vec{v}$; dans notre cas, on considérera cette fonction comme une simple constante: $1/4\pi$), et $L(x_t,\ \vec{v})$ est l'intensité lumineuse arrivant au point x_t le long du rayon (bref, notre "getColor"). La seconde intégrale se fait sur la sphère de toutes les directions, et non sur une hémisphère comme on faisait avant dans le cas de l'équation du rendu sur des surfaces opaques. Cette formule indique juste qu'il faut sommer pour chaque point le long du rayon les contributions de toutes les directions autour de ce point, et l'atténuer par la fonction de transmittance.

Évaluer cette fonction peut être très coûteux, car il y a une double intégrale qu'il nous faudrait discrétiser à chaque intersection (et ce, de manière récursive car elle fait appel à getColor!). Heureusement, on peut là encore faire appel à l'intégration de Monte-Carlo est n'ajouter qu'un seul terme de cette double intégrale de manière stochastique.
En particulier, pour chaque intersection, il suffira d'ajouter à notre getColor qu'une seule valeur : $\sigma_s(x_t)\, T(t)\, \rho(\vec{w},\ \vec{v})\, L(x_t,\ \vec{v}) / p(t,\ \vec{v})$ pour un seul paramètre échantillonné pour "t" et pour $\vec{v}$. On n'oubliera alors pas la division par $p(t,\ \vec{v})$ qui est la probabilité d'avoir échantillonné à la fois "t" et $\vec{v}$, qui est le produit de la probabilité d'avoir échantillonné t et d'avoir échantillonné $\vec{v}$ (si les deux tirages sont indépendants!).

Échantillonner $\vec{v}$ peut se faire de manière uniforme, en tirant aléatoirement une direction sur la sphère (Global Illumination Compendium, partie (33) page 19). Mais on tirera profit de la méthode décrite dans la section précédente, qui choisit aléatoirement de tirer une direction sur la sphère ou bien une direction vers notre source de lumière.

Tirer un paramètre t peut être difficile à faire correctement:

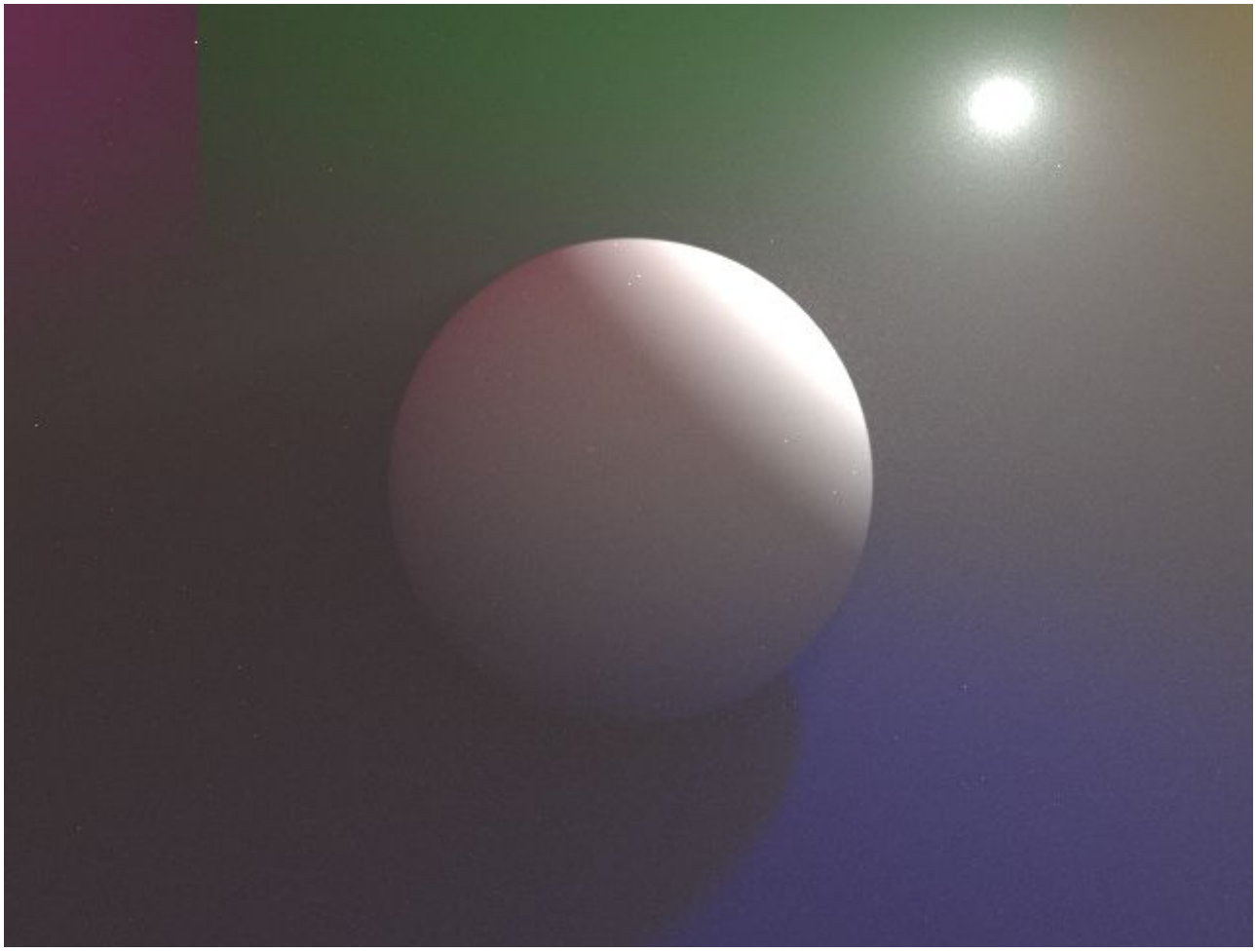
- On peut échantillonner t de manière uniforme entre 0 et la première intersection. Le problème est que l'absorption est exponentielle, et les échantillons pour les fortes valeurs de t auront une contribution quasiment nulle à chaque fois.
- On peut échantillonner t de manière exponentielle (on tire t avec une loi exponentielle) afin de prendre en compte l'absorption. Rappel: tirer une variable aléatoire selon une loi de probabilité $p(t)$ se fait en calculant la cdf ($P(t) = \int_{-\infty}^{t} p(s)\, ds$), en inversant cette fonction, et en tirant un nombre aléatoire uniforme sur [0, 1]. Pour la loi exponentielle, on a $p(t) = \lambda \exp(-\lambda t)$ donc $P(t) = 1 - \exp(-\lambda t)$, donc l'inverse s'écrit

$P^{-1}(y) = -\log(1-y)/\lambda$ et donc générer un nombre aléatoire suivant une loi exponentielle se fait en générant un nombre aléatoire uniforme χ entre 0 et 1, et en lui appliquant la fonction $P^{-1}(\chi)$.

- On peut à la place essayer d'échantillonner le paramètre qui donnera le plus de contribution par la source de lumière. On se référera à cette formule pour ce calcul (simple). Cette dernière version marche souvent mieux lorsqu'il n'y a pas trop d'absorption.

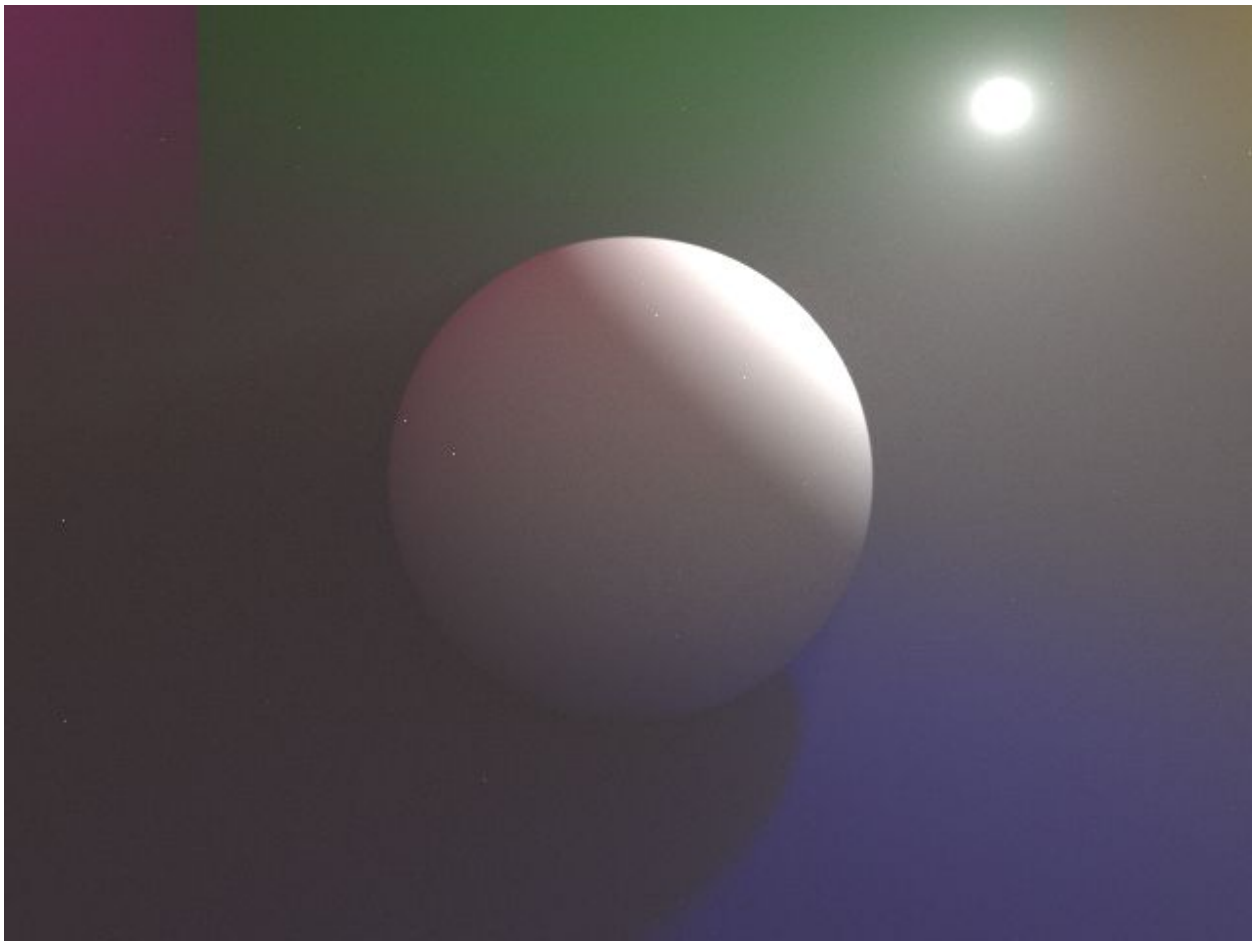
Avec 2000 rayons par pixel, nous obtenons les résultats suivants (attention, notre méthode reste relativement naïve et ne représente pas l'état de l'art! ne pas utiliser ces résultats à des fins de comparaisons dans des articles scientifiques!) :

- Méthode qui échantillonne t avec une loi exponentielle :



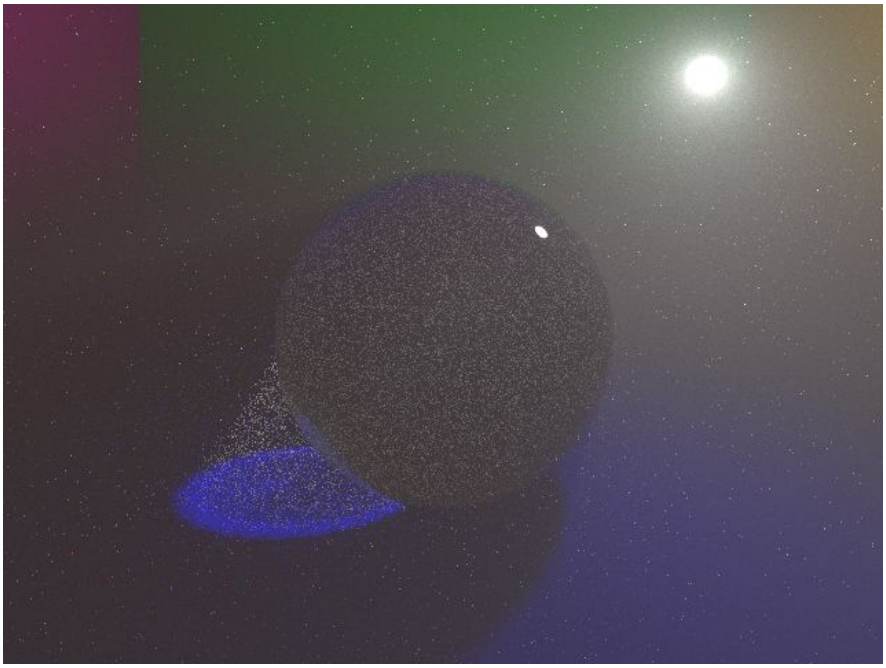
(les occasionnels pixels blancs sont appelés des “fireflies” et sont dûs à des événements très peu probables, dont la division par la probabilité produit des valeurs extrêmes)

- Méthode qui échantillonne en maximisant la contribution de la lumière:

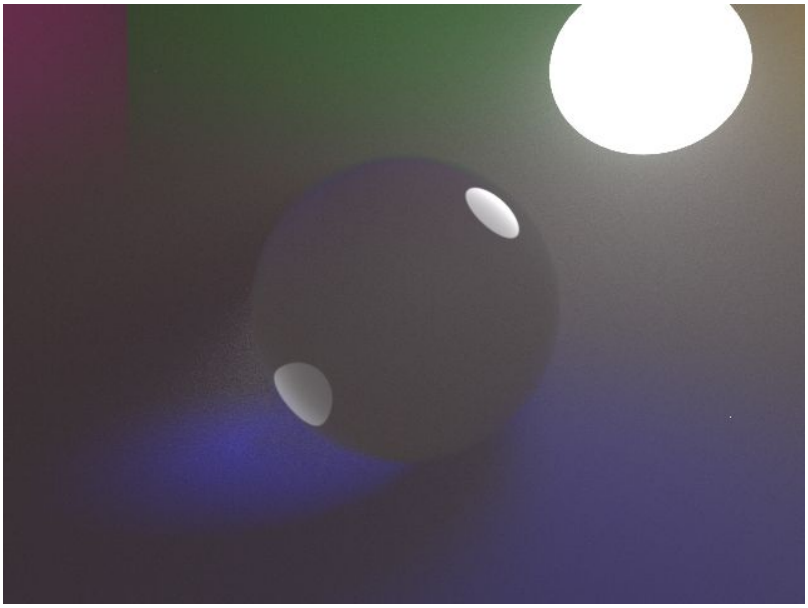


Et sur la voiture, avec une plus grosse source de lumière:

En revanche, si la sphère est transparente, nous obtenons le résultat suivant, avec 1000 rayons par pixel:



Ici, le problème est différent : dans la zone de caustique (la partie illuminée à gauche de la sphère), il n'existe pas vraiment de bonne stratégie qui permette d'envoyer des rayons qui auront une contribution non nulle, i.e., qui toucheront la source de lumière. En effet, au niveau de la caustique, soit nous envoyons des rayons dans des directions arbitraires mais avons peu de chance de tomber sur la lumière, soit nous visons directement la lumière, mais la réfraction au sein de la sphère nous fera manquer la cible. Augmenter la taille de la source de lumière résout ce problème puisque cela augmente la probabilité pour un rayon d'atteindre la source, mais ce n'est pas forcément l'effet désiré (la caustique est donc aussi plus floue) :



Pour pallier à ce genre de problème, il existe des méthodes alternatives tels que le "Bidirectional Path Tracing" ou bien le lancer de photons : ces méthodes envoient des rayons de lumière depuis les sources lumineuses afin de bien reconstruire ces caustiques.

Les autres BRDFs

Nous avons vu la BRDF constante (diffuse), et la BRDF spéculaire pure. Il existe toute une gamme de BRDFs différentes (voir cours 3).
Notamment, nous parlerons de deux modèles représentatifs de leurs classes:

- Le modèle de Phong. Il s'agit d'un modèle phénoménologique : un simple lobe Gaussien autour de la direction spéculaire, qui décrit bien les matériaux dits "glossy" (un peu brillants). C'est le modèle géré par OpenGL lorsque vous faisiez du rendu interactif.
- Le modèle de Cook-Torrance. Il s'agit d'un modèle physique, basé sur la théorie des microfacettes. Une surface est vue comme un collection de microfacettes orientées dans toutes les directions de manière aléatoire. Si cet aléatoire a tendance à orienter les microfacettes dans la même direction, on obtiendra une surface spéculaire, alors que si l'aléatoire désordonne complètement les facettes, la surface sera globalement plus rugueuse et donc diffuse.

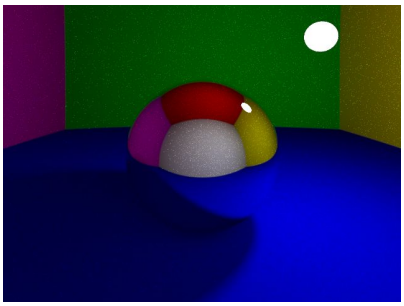
L'intérêt des ces deux modèles est qu'un échantillonnage efficace est disponible dans le Global Illumination Compendium , page 32 et 34, et page 20 et dans ce papier (page 18). Voir ces pages pour l'expression de ces BRDFs, une méthode d'échantillonnage et la densité de probabilité associée.

Pour les implémenter, on pourra rendre notre classe Material abstraite en produisant des méthode "sample" (qui échantillonne), "BRDF" (qui donne la valeur de la BRDF), et "pdf" (qui donne la densité de probabilité associée à un échantillon).
Par exemple, un matériau diffus héritera de la classe Material, et implémentera sample par la méthode vue au cours 3, implémentera BRDF en retournant simplement le coefficient diffus divisé par Pi (et, si vous préférer l'y intégrer, le produit scalaire avec la normale), et implémentera pdf en retournant le produit scalaire entre l'échantillon généré et la normale divisé par Pi.

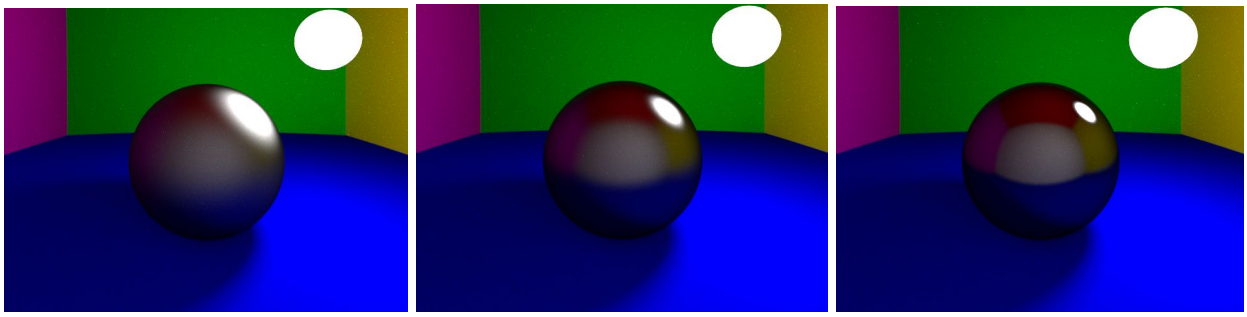
Cumuler les contributions lumineuses s'écrit alors:

```
Vector I = material->sample(...);
color = color + material->BRDF(I, O)*getColor(I, ...)
                *dot(I, N)/material->pdf(I, O)
```

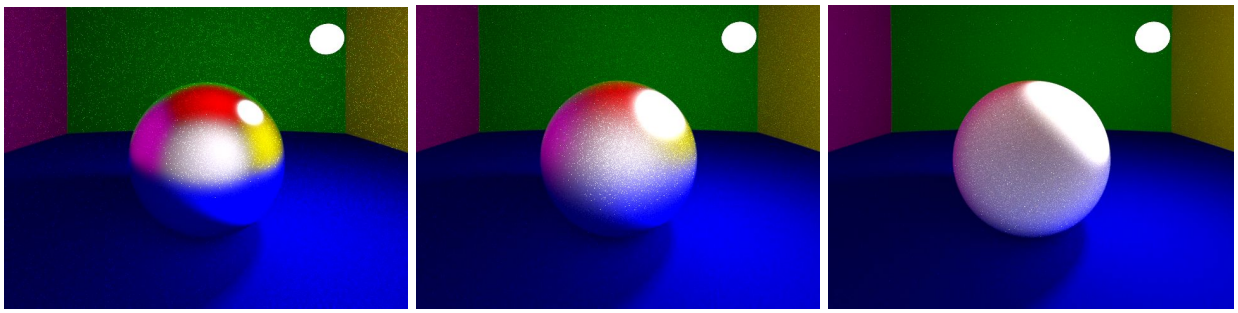
Sphère spéculaire pure :



Modèle de Phong avec exposants 10, 100, puis 1000:



Modèle de Cook-Torrance pour un paramètre m = 0.1, 0.3 et 1 (attention, la lumière est deux fois plus grosse!) :

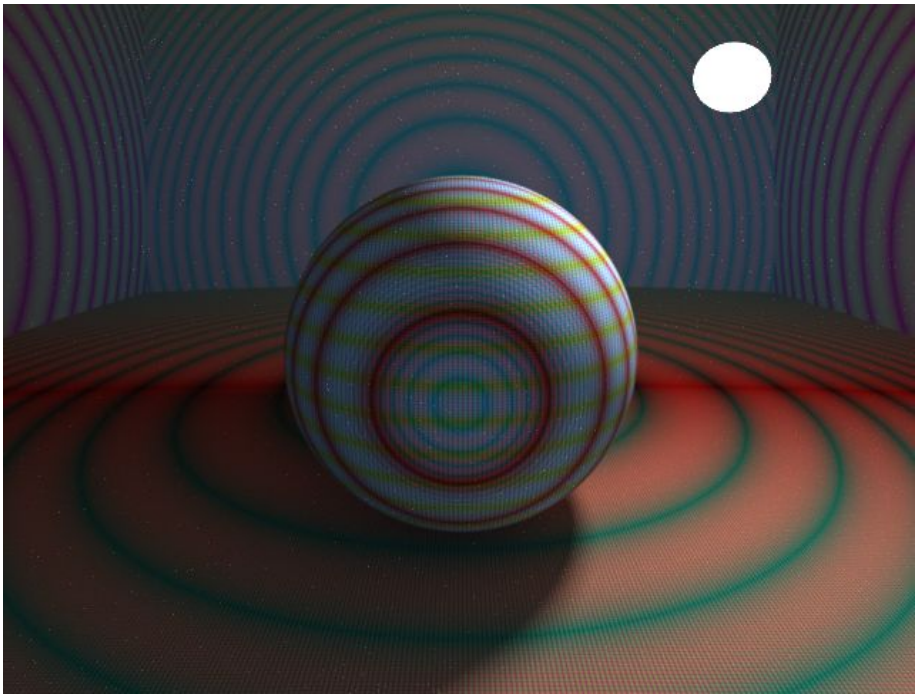


Malgré l'échantillonnage en $\exp(-\tan^2\theta/m^2)$ et 2000 échantillons par pixel, ce résultat reste bruité : l'échantillonnage et la réduction de variance pour les BRDFs glossy est un sujet actif de recherche.
(Note: la différence de luminance entre les deux modèles est sûrement due à un bug!).

Les textures

Ajouter des textures est maintenant enfantin. Il existe deux familles de texture:

- Les textures procédurales : il s'agit d'une simple fonction des coordonnées du point d'intersection et/ou de tout autre paramètre (la normale etc.).

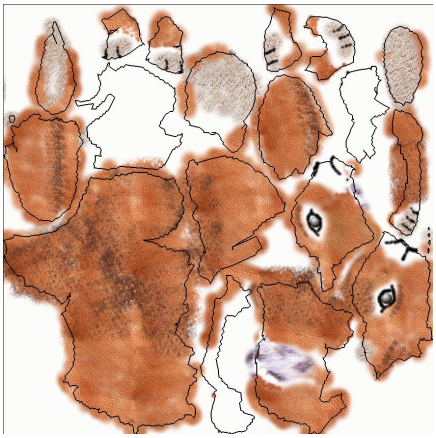


J'ai ici appliqué de simples fonctions cosinus et sinus aux coordonnées et normales au point d'intersection, et utilisé le résultat comme couleur diffuse. C'est évidemment une formule naïve, mais des solutions plus élaborées permettent de très bons résultats : c'est le fer de lance de sociétés telles que Allegorithmic en France.

- Les textures bitmap : Il s'agit d'une image stockée.

Une texture bitmap est représentée par deux choses:

- une image comportant des pixels colorés

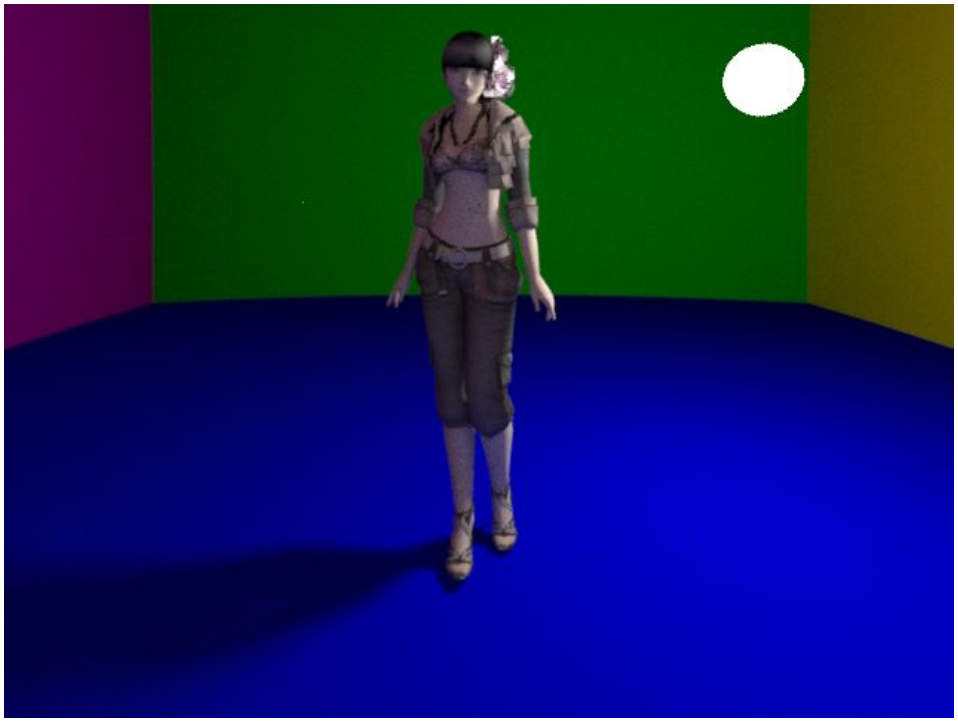


(http://alice.loria.fr/index.php/publications.html?Paper=lscm@2002)

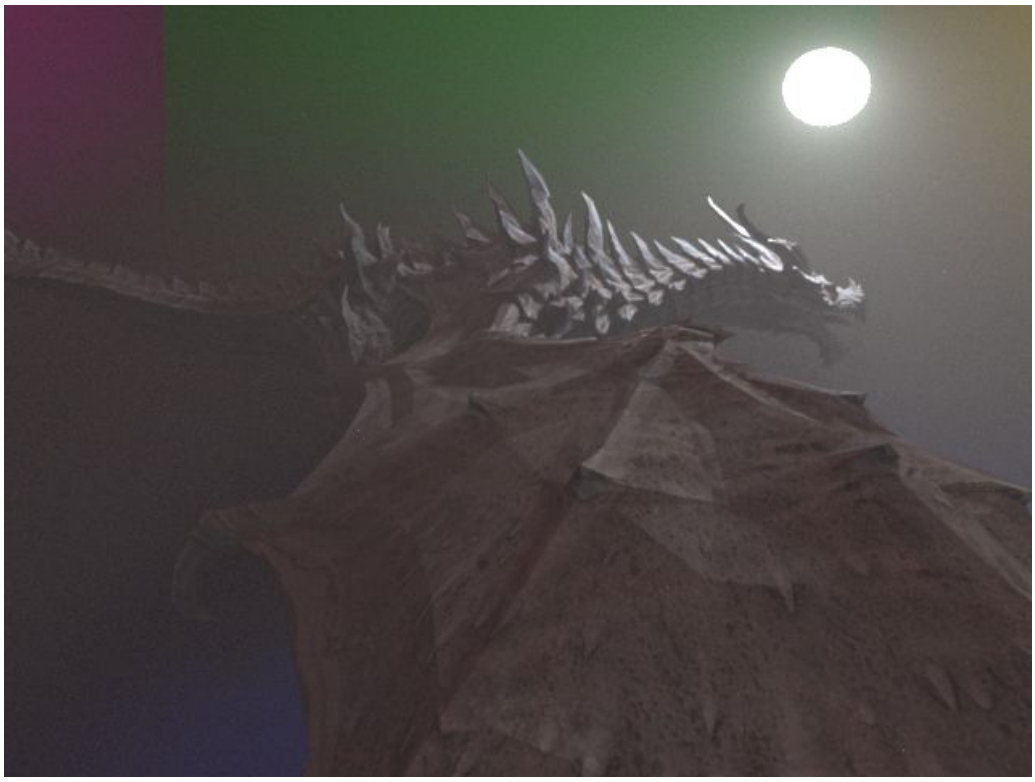
- une paramétrisation du maillage qui indique pour chaque sommet du maillage à quel pixel de l'image il correspond. Cela correspond aux coordonnées UV. On ne discutera pas de *comment* est générée cette paramétrisation : il s'agit d'un problème de géométrie un peu complexe (voir l'article du lien ci-dessus pour un exemple). On considérera que ces coordonnées UV sont fournies dans le fichier .obj que l'on a lu.

(http://alice.loria.fr/index.php/publications.html?Paper=lscm@2002)

Les textures peuvent ainsi être utilisées pour moduler la couleur diffuse des surfaces (ou, plus rarement, les coefficients spéculaires, voire perturber les normales ou la position de points). Ainsi à chaque intersection d'un rayon avec triangle, on regarde les coordonnées UV des trois sommets qui forment ce triangle, on interpole ces coordonnées UV en utilisant les coordonnées barycentriques de la même manière que nous avions interpolé les normales (voir section sur les intersections avec les triangles), et on pioche ensuite la valeur du pixel qui se trouve à la position (u, v) de l'image (attention, les coordonnées UVs sont le plus souvent normalisées entre 0 et 1, de telle manière qu'il faille multiplier ces valeurs par la hauteur et largeur de la texture).
On pourra faire appel à la librairie CImg pour lire les images. Attention, la coordonnée "y" est inversée, et certains modèles requièrent plusieurs textures (par exemple, sur l'image ci-après il y en a 6 pour différentes parties du corps).



Et avec le dragon texturé à http://tf3dm.com/3d-model/alduin-dragon-rigged--76875.html et de la brume (attention, il ne possède pas de normales par sommet, possède certains quads que vous pourrez convertir en paires de triangles, et a un système de coordonnées différent!).



à venir…

La diffusion de sous-surface

Le Tone Mapping

Remerciements:
Guillaume Bouchard et Jean-Claude Iehl pour de nombreuses discussions! Et Mathias Paulin pour m’avoir initié à tout cela il y a bien longtemps.