ÉCOLE CENTRALE LYON

MOS Informatique graphique

Rapport

Raytracing

Océane BLIGAND

Mars 2019

Table des matières

Introduction

Dans ce cours, nous avons étudié le raytracing : c'est une méthode permettant d'obtenir des rendus très réaliste basée sur le lancer de rayons. En simulant le trajet de la lumière grâce aux équations physiques nous pouvons ainsi obtenir des effets fidèles à la réalité tels que la réfraction, la réflexion, l'éclairage indirect... Le raytracing simule le comportement de la lumière. On pourrait simuler des rayons partant des sources lumineuses et récupérer ceux arrivant à la caméra, mais cela représenterait un faible pourcentage des rayons lancé, on aurait alors de nombreux calculs inutiles. Nous allons ici utiliser le backward raytracing : on simule les rayons en sens inverse, c'est-à-dire que nous lançons des rayons partant de la caméra au lieu des sources de lumière.

Visualisation de sphères

Nous allons d'abord tenter de visualiser une sphère simple sans nous occuper de la réflexion ou réfraction de la lumière. Nous lançons des rayons à partir de chaque pixel de la caméra, si le rayon intersecte une sphère, on met le pixel en blanc, sinon en noir.

La classe Vector

Nous allons manipuler des objets 3D, il est donc important de créer une classe Vector comportant une composante x, y et z. Nous pourrons également l'utiliser pour des points ou des couleurs (les composantes correspondront à rouge, vert, bleu).

La classe Ray

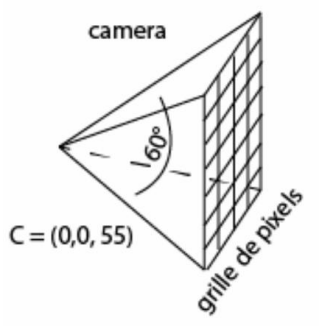
Cette classe permet de définir un rayon. Elle contient son origine (un point donc de la classe Vector) et sa direction (un Vector de préférence normalisé).

La classe sphère

Nous créons également une classe sphère contenant son centre et son rayon. Cela nous permettra de créer facilement plusieurs sphères et de les ajouter à notre image.

Lancer des rayons depuis la caméra

Nous plaçons notre caméra au point C. Notre caméra a un field of view (fov) de 60°. Nous allons lancer un rayon par pixel. Nos rayons auront tous pour origine C et devront passer par le centre de leur pixel.



Intersection avec une sphère

Pour chaque rayon, on cherche à savoir s'il intersecte la sphère. On sait que tous les points P appartenant à une sphère de centre O et de rayon R satisfait :

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

Si ce point P appartient à notre rayon il satisfait aussi :

$$P = C + t.V$$

Avec C l'origine du rayon et V sa direction (t étant un réel, positif car le rayon ne part pas à l'arrière de la caméra).

En réunissant les deux :

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

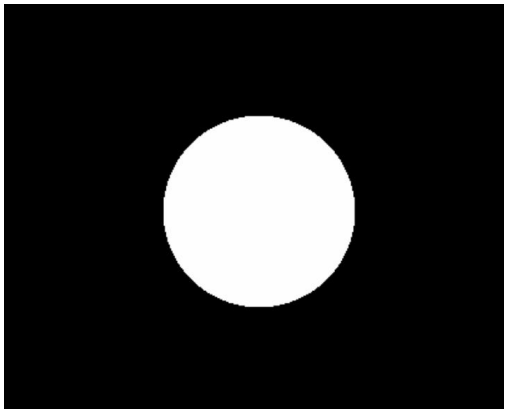
puis,

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

Il suffit alors de résoudre cette équation du second degré en t.

Si le delta est négatif ou si les solutions sont négatives il n'y a pas d'intersection et l'on place le pixel en noir. Sinon le pixel est blanc.

Nous créons grâce à ces formules une méthode intersect(rayon) dans la classe Sphere qui renvoie true si le rayon intersecte la sphere et false sinon.



Ajout d'une source de lumière

Nous allons maintenant ajouter une source de lumière. Lors d'une intersection, au lieu de mettre un pixel blanc nous récupérerons la lumière réfléchie par la sphère (qui différera selon sa couleur).

La lumière est un point et est définie par son intensité I. Nous ajoutons un attribut color à notre classe sphère (un Vector (rouge, vert, bleu)).

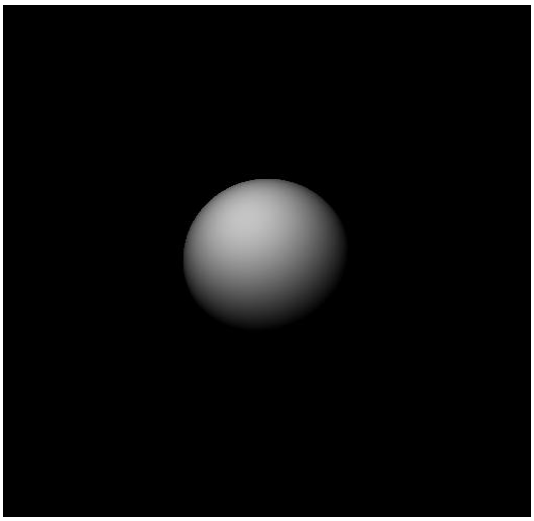
Pour connaître l'intensité i de la lumière réfléchie nous avons la formule suivante :

$$\max(0, \vec{l}.\vec{n})*I/d^2$$

Avec le vecteur l la direction du point de la sphère vers la lumière (ponctuelle), n est la normale à la sphère au point considéré, I est l'intensité de la lumière et d est la distance entre le point de la sphère et la lumière.

Nous multiplions ensuite cela par la couleur de la sphère (le vecteur color) pour obtenir la couleur à mettre dans le pixel considéré.

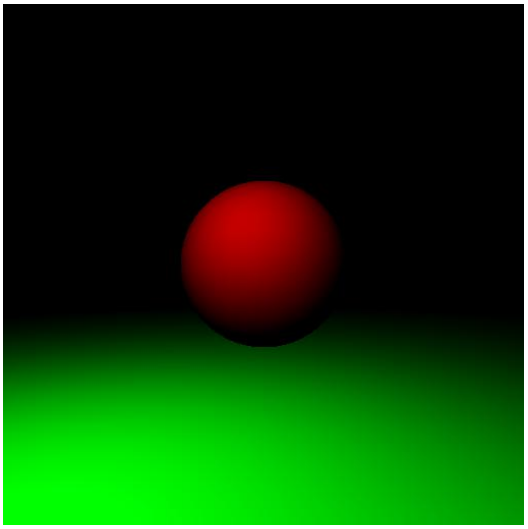
Nous obtenons l'image suivante :



La classe Scène

Nous souhaitons pouvoir ajouter facilement plusieurs sphères à notre image. Nous créons pour cela la classe Scène qui contient en attribut une liste de sphère. Elle possède une méthode addSphere permettant d'ajouter une sphère à cette liste. Elle possède également sa propre méthode intersect qui va effectuer toutes les méthodes intersect de ses sphères et renvoyer l'intersection la plus proche (car il y a possibilité qu'une sphère en cache une autre, il faut donc bien prendre la première sphère rencontrée par le rayon).

On ajoute alors une énorme sphère représentant le sol. On obtient la figure suivante :

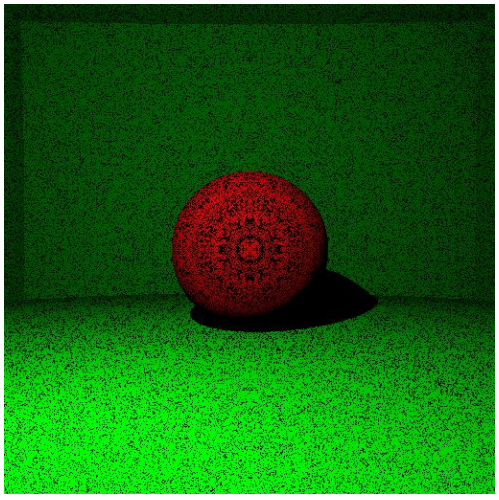


Correction gamma

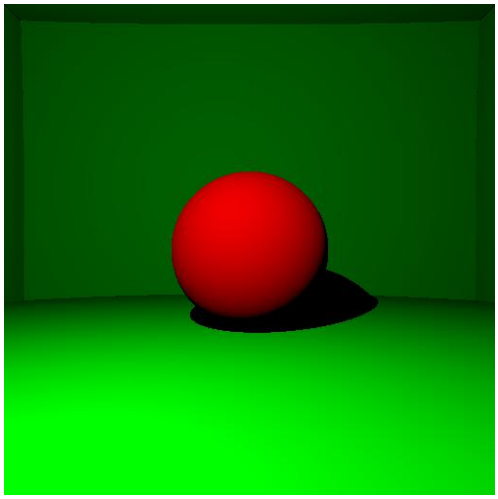
L'œil humain ne perçoit pas la lumière linéairement, les valeurs sombres ne sont pas aussi bien différenciées que les valeurs claires. On passe pour cela les valeurs des couleurs des pixels à la puissance 1/2.2.

Les ombres portées

Pour l'instant les sphères sont illuminées comme si elles étaient seules face à la lumière or il est possible qu'une sphère en cache une autre. Il devrait alors y avoir une ombre. Pour avant de calculer l'intensité de la lumière réfléchie par un point de la scène on vérifie qu'il n'y a pas d'objet entre ce point et la lumière. S'il existe un tel objet alors on renvoie du noir (une ombre). On obtient alors la figure suivante :



On voit que certains points sont noirs alors qu'ils ne devraient pas l'être cela est dû à des imprécisions numériques qui peuvent placer certains points, qui théoriquement devrait être à la surface des sphères, légèrement à l'intérieur de ces dernières. Ainsi lorsque que l'on cherche s'il y a un objet entre ce point et la lumière on trouve la sphère auquel appartient ce point et on met le pixel en noir alors qu'il ne devrait pas l'être. Pour éviter cela on ajoute un epsilon fois la normale de la sphère à ce point pour être sûr que le point est hors de la sphère. On obtient :



Miroirs

Nous souhaitons maintenant simuler des sphères miroirs, elles doivent réfléchir toute la lumière selon une direction bien déterminée.

On ajoute un attribut miroir à la classe sphère, c'est un booléen (true si la sphère est un miroir).

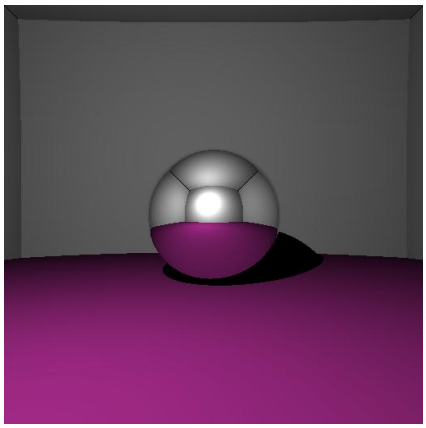
Pour trouver la couleur d'un pixel dont le rayon frappera un miroir il faudra rediriger ce rayon dans la direction réfléchie. Cette direction est :

$$\vec{j} = \vec{i} - 2 < (\vec{i}), \vec{N} > \vec{N}$$

Avec i la direction incidente et N la normale à la sphère au point d'arrivée du rayon.

On utilise une fonction récursive getColor qui chaque fois qu'un rayon tombe sur un miroir, redirige ce rayon dans la direction réfléchie et rappelle getColor dans cette nouvelle direction.

Toutefois, si deux miroirs se font face on pourrait avoir des boucles infinies, on définie alors un nombre de réflexions maximum. Posé ici à 5 rebonds. On obtient :



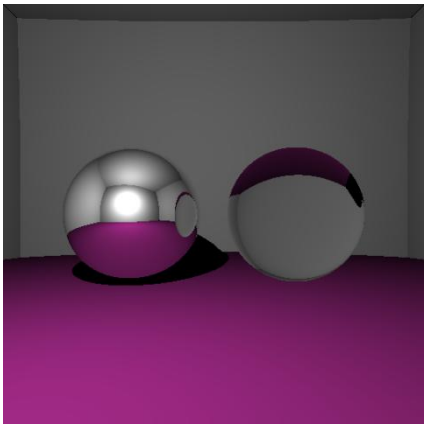
Transparence

On souhaite maintenant avoir des sphères simulant la transparence et des matériaux tel que le verre ayant des indices de réfraction différents de l'air. Pour cela on réutilise la méthode récursive getColor sauf qu'au lieu de réfléchir le rayon, ce dernier sera réfracté à l'intérieur de la sphère. La direction de réfraction est définie par :

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

Trouvée grâce à la loi de Snell Descartes. On garde la limite de rebonds imposés précédemment et la réfraction compte comme un rebond.

On obtient alors :



Avec une sphère miroir à gauche et une sphère transparente à droite avec un indice de réfraction de n=1.4.

Éclairage indirect

Jusqu'ici pour déterminer la lumière réfléchie par une sphère non transparente et non miroir, nous regardions juste la lumière parvenant à ce point venant de la source de lumière, mais, en réalité, les surfaces diffuse réfléchissent la lumière vers les autres objets. Il faudrait donc aussi prendre en compte la lumière venant des autres surfaces de la scène.

BRDF

La BRDF est une fonction de densité de probabilité. Elle représente la probabilité qu'a un rayon d'être diffusé dans une certaine direction après arriver sur une surface. Elle dépend du type de surface, de la longueur d'onde. Nous avons en fait déjà utilisé des BRDF précédemment : une constante pour les surfaces diffuses et un dirac pour les surfaces spéculaires (miroirs).

Équation du rendu

Pour obtenir l'intensité de la lumière en un point nous allons utiliser l'équation :

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

L0 est la lumière sortante en ce point, E est l'émissivité (si c'est une source de lumière), f est la BRDF est Li(x,i) est la lumière arrivant au point x venant de la direction i.

Toutefois, on remarque que cela fait une intégrale d'intégrale d'intégrale .... car Li est aussi calculée par cette formule. Pour avoir des calculs faisables, on utilise la méthode de Monte-Carlo pour l'intégration.

Méthode de Monte-Carlo

D'après la méthode de Monte-Carlo :

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

Où les xi sont n échantillons aléatoires suivant une loi aléatoire X et p(xi) la probabilité de X=xi.

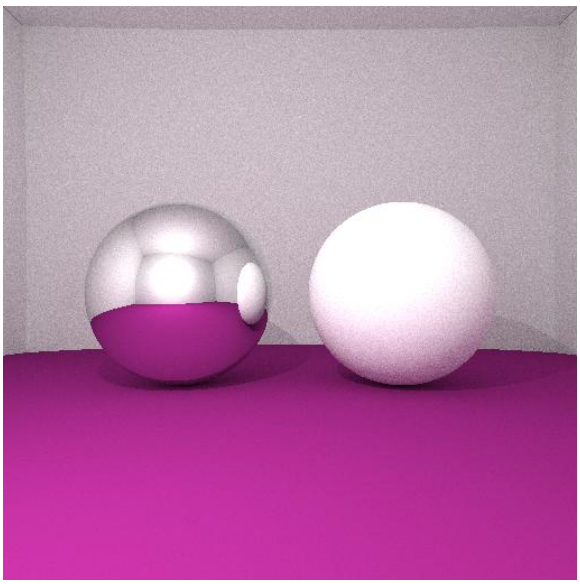
On essaie d'avoir une loi qui ressemble à f pour avoir de meilleurs résultats. Ici, nous allons utiliser une loi uniforme suivant le cosinus. On créé pour cela une fonction random_cos qui prend en argument un vecteur représentant la normale à une surface et qui renvoie une direction qui aura plus de probabilité de se trouver autour de la normale.

Dans le getColor, on récupère déjà la contribution directe de la lumière (comme précédemment) puis on rappelle getColor dans une direction obtenue avec random_cos. Cela compte comme un rebond.

Pour avoir de meilleurs résultats on envoie 128 rayons par pixel puis on fait la moyenne au lieu d'en envoyer un seul comme précédemment.

On effectue les calculs pour les différents rayons envoyés en parallèle pour gagner du temps.

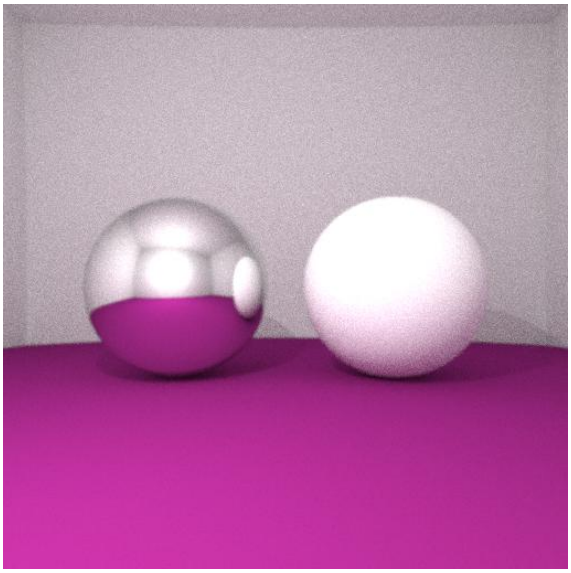
On obtient en 26s:



On voit que le rose se reflète dans la sphère blanche.

Anti-Aliasing

On remarque que dans les images précédentes les sphères font un effet «escalier» qui n'est pas très esthétique. Pour améliorer cela, au lieu d'envoyer les 128 rayons exactement au centre du pixel on les envoie aléatoirement dans ce pixel selon une gaussienne (plus au centre que sur les bords du pixels). Cela permet de lisser les bords. On obtient toujours en 26s :



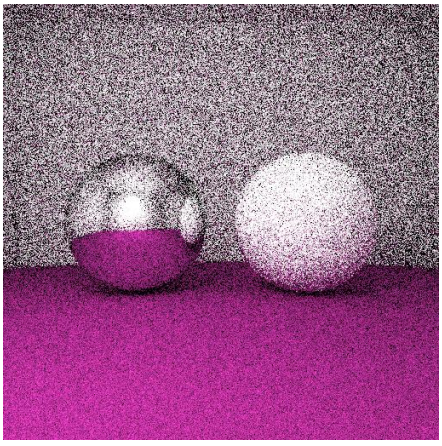
Ombres douces

Jusqu'à maintenant nous utilisions une lumière ponctuelle mais cela est peu réaliste et provoque des ombres brutales. Nous allons donc faire une sphère de lumière à la place. On place le centre de cette sphère à l'endroit de l'ancienne lumière ponctuelle.

Méthode naïve

Dans une première approche, on donne un attribut émissivité à la classe sphère qui sera true pour notre sphère lumière. On supprime la partie directe de getColor et on renvoie seulement l'intensité qui était calculée en directe dans la partie indirecte lorsque l'on rencontre une sphère émissive.

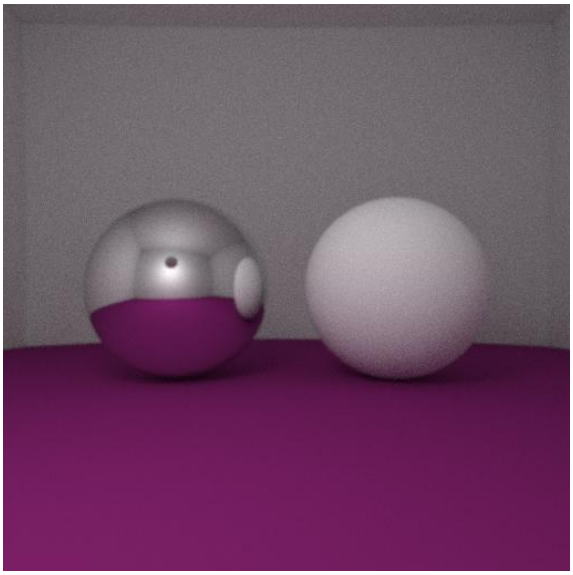
On obtient en 21s:



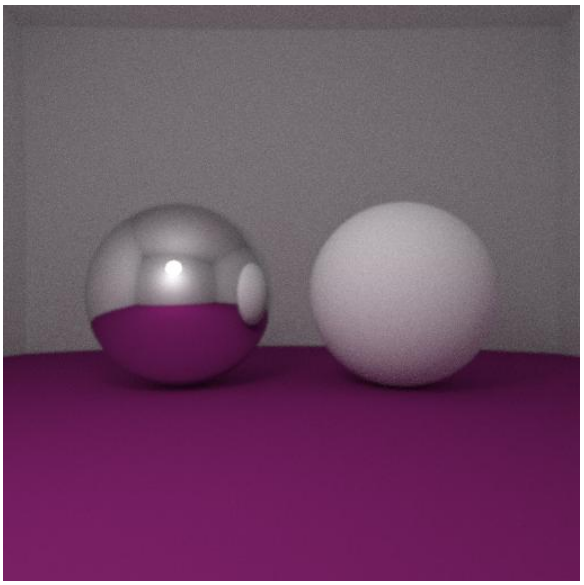
L'image est très bruitée, cela est dû au fait que les rayons n'arrivent pas souvent sur la lumière.

Réduire le bruit

Afin de réduire le bruit on va favoriser la direction de la lumière pour l'envoi des rayons tout en conservant l'éclairage indirect. On resépare le code en contribution directe et indirecte. Pour la contribution directe, on va envoyer aléatoirement mais vers la sphère lumière. On garde l'indirect tel quel. On obtient en 40s :



On remarque que dans un miroir la lumière apparaît noire. Pour résoudre il faut ajouter une condition : lorsque l'on réfléchie par un miroir on spécifie que l'on veut voir la lumière. Nous l'avons fait à travers un booléen «direct» passé en paramètre passé à true lors d'une réflexion sur un miroir.

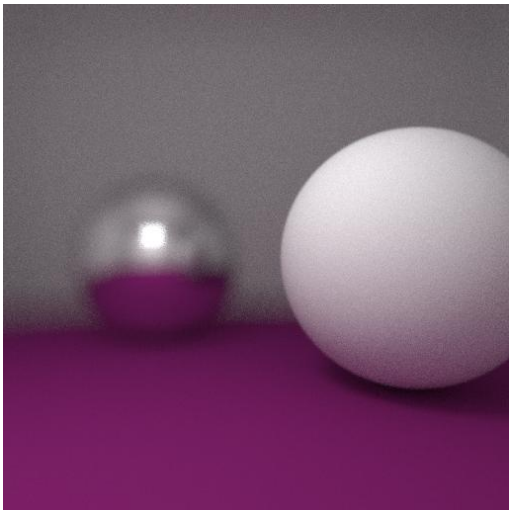


Profondeur de champ

On souhaite ici pouvoir visualiser un flou de profondeur. Seuls les objets sur le plan focal doivent être nets, les autres auront un effet flou. On définit alors une distance focale.

On simule en fait une lentille convergente dans la caméra et un diaphragme (obturateur). Plus le diaphragme sera ouvert plus le flou de profondeur sera prononcé. On va maintenant viser les pixels sur le plan focal, où l'image doit être nette. Au lieu de partir d'un point fixe comme avant on va partir aléatoirement d'un carré qui simule le diaphragme. Cela permet d'avoir de la netteté sur le plan focal et un flou de plus en plus prononcé plus on s'en éloigne.

On obtient :



Maillage

Classe Triangle

Un maillage représente une forme 3D à l'aide de triangles. Nous allons donc d'abord essayer de visualiser des triangles. Nous créons pour cela une classe triangle à l'image de la classe sphère mais un triangle est décrit par trois points et la méthode intersect est un peu différente de la méthode intersect des sphères. En effet pour tester une intersection avec un triangle, on teste d'abord l'intersection avec le plan décrit pas le triangle :

Un point P appartient au plan défini par le point P0 et la normale N si :

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

Avec l'équation du rayon :

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

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

On obtient l'intersection :

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

Il y a intersection seulement si t>0.

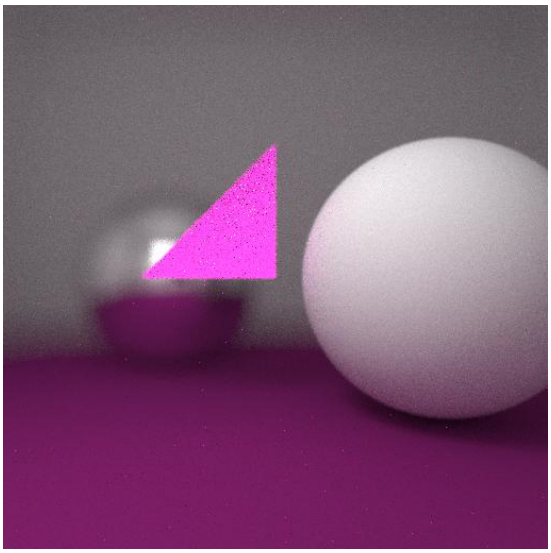
Si le rayon intersecte le plan on doit vérifier qu'il intersecte le triangle. Pour cela on utilise les coordonnées barycentriques. On trouve les coordonnées barycentriques du point P d'intersection.

$$\lambda_0 + \lambda_1 + \lambda_2 = 1 \text{ et } \lambda_0 P\vec{P}_0 + \lambda_1 P\vec{P}_1 + \lambda_2 P\vec{P}_2 = \vec{0}$$

Les lambda sont les coordonnées barycentriques et P0 P1 et P2 sont les sommets du triangle.

Si ces coordonnées sont toutes entre 0 et 1 alors le point P est à l'intérieur du triangle.

On obtient après avoir ajouté un triangle à la scène:



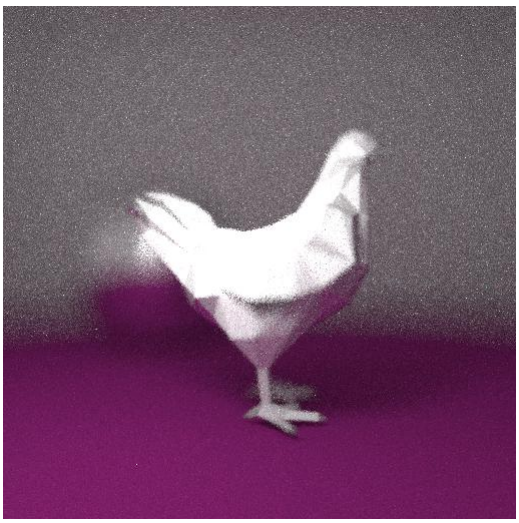
Lecture d'un maillage

On souhaite ensuite ajouter un maillage tout fait composé de plusieurs triangles. On réutilise pour cela un code déjà fait pour la lecture de fichiers .obj qui donne la liste des sommets et la liste des triangles du maillage. On met ce code dans une classe Geometry qui a pour vocation à être ajouté à la scène comme les sphères. On code une méthode intersect dans Geometry. Cet intersect teste l'intersection avec tous les triangles et donne l'intersection la plus proche (comme avec les sphères dans une scène).

Toutefois cela ne sert à rien de tester tous les triangles quand le rayons passe largement à côté du maillage. Pour éviter des calculs inutiles, on ajoute une boîte englobante à une Geometry. On crée une classe BoiteEnglobante qui contient deux points, le coin inférieur gauche et supérieur droit de la boîte. Une Geometry possède en attribut une boîte englobante calculée lors du constructeur.

Dans la méthode intersect on teste d'abord si le rayon intersect la boîte avant de tester les triangles.

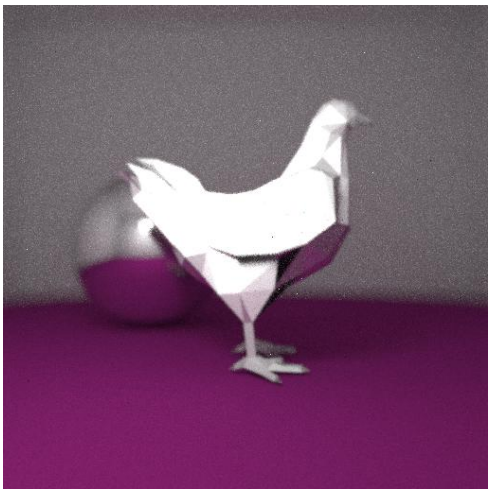
On obtient avec un maillage d'une poule (en 36s avec 16 rayons par pixel) :



BVH

Le BVH (Bounding Volume Hierarchies) permet de diminuer le temps de calculs des intersection en utilisant des boîtes englobante imbriquées formant un arbre de boîtes englobantes. On teste ainsi la racine puis s'il y a intersection on teste avec ses boîtes enfants et ainsi de suite. Cela permet de n'évaluer l'intersection qu'avec certains triangles et diminue ainsi drastiquement le temps de calcul.

On obtient l'image suivante en 40s avec 64 rayons :



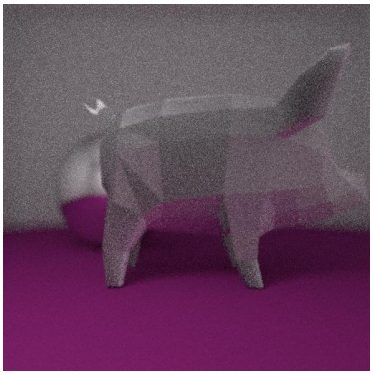
On a ainsi multiplié par 4 le nombre de rayons pour un temps d'exécution équivalent.

Lissage de Phong

On voit sur les images précédentes que notre modèle 3D possèdent des arêtes peu réalistes. Cela est du aux fait que le maillage est constitué de triangle et des triangles côte à côte ayant des normales différentes on verra des différences nettes de couleurs entre les triangles. Pour éviter, on va modifier les normales de façon continue sur un triangle et non brutalement aux arêtes. Pour obtenir la normale à un point à l'intérieur d'un triangle, on combine les normales des sommets de ce triangle chacun multiplié par la coordonnée barycentrique correspondant à ce sommet.

Toutefois pour cela le modèle 3D doit fournir les normales en chaque sommet, or le modèle de la poule utilisé précédemment n'en possédait pas. Nous avons donc changé de modèle.

On obtient en 16s avec 16 rayons par pixel:



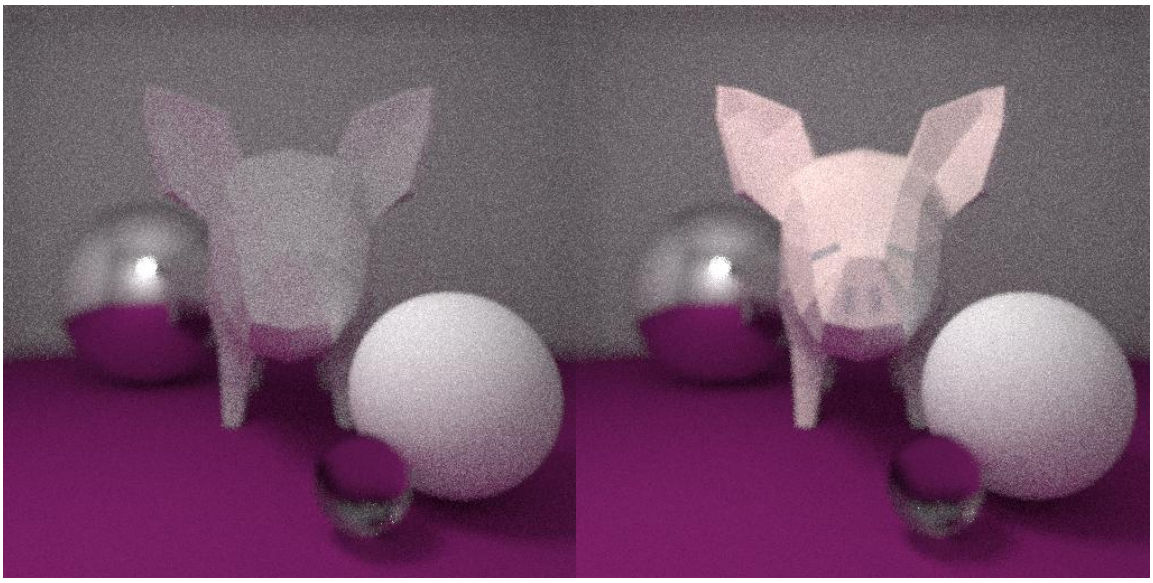
Textures

On cherche maintenant à ajouter des textures. La fonction readOBJ fournie nous donne pour chaque triangle la position de ce triangle dans le fichier de texture, appelé uvs.

Nous créons d'abord une méthode addTexture dans Geometry qui prend en entrée une image bmp (représentant la texture), la lit et rentre les données de l'image dans des attributs. Cette fonction est simplement une fonction trouvée sur internet permettant de lire une image bmp.

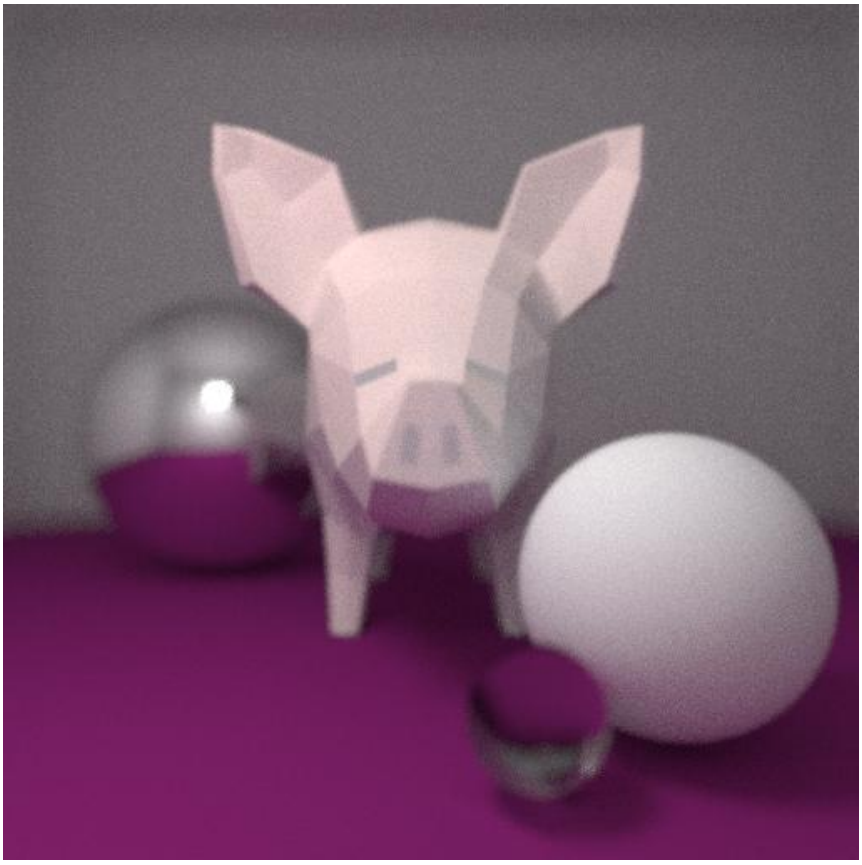
Maintenant quand nous appelons intersect nous souhaitons renvoyer la couleur au point d'intersection, pour cela on ajoute un vecteur couleur_texture dans toutes les fonctions intersect. Pour la sphère on renvoie seulement la couleur de la sphère. Dans geometry, il faut aller chercher la couleur du point correspondant dans la texture. On utilise pour cela uvs pour trouver les coordonnées 2D à aller chercher dans l'image de texture.

Après quelques mises au point à la main (inversion des coordonnées dans uvs, offset des valeurs obtenues), on obtient (le cochon a été mis de face car seuls les yeux et le museau ont des textures différentes du corps) en 13s avec 16 rayons par pixel:



sans textures avec textures

Avec 128 rayons par pixel (en 1m47s) :



Conclusion

Ce cours m'a permis de découvrir la méthode de raytracing et j'ai pu découvrir comment obtenir une image réaliste avec C++. J'ai découvert et implémenter tour à tour comment obtenir une image de sphère ,comment rendre cette sphère spéculaire, transparente, diffuse, puis comment la rendre plus réaliste en prenant en compte l'éclairage indirect. J'ai également découvert le fonctionnement des maillages pour les objets 3D (en particulier le format .obj) et comment les textures sont appliquées sur un maillage.