Le lancer de rayons - 1ère partie

Illumination, ombres, réflexions

Marc Lorenzi

13 octobre 2018

[1]
```
import matplotlib.pyplot as plt
from math import *
```

[2]
```
plt.rcParams['figure.figsize'] = (15, 10)
```

Le **lancer de rayons** (Ray Tracing en anglais) est une technique permettant le *rendu* sur un écran d'une **scène** composée d'**objets**.

Dans toute notre discussion, nous travaillerons dans l'espace $\mathbb{R}^3$ euclidien. Le repère canonique $(O, \overrightarrow{i}, \overrightarrow{j}, \overrightarrow{k})$ sera orienté un peu différemment de ce que l'on fait d'habitude : nous prendrons $\overrightarrow{i} = (0, 1, 0)$, $\overrightarrow{j} = (0, 0, 1)$ et $\overrightarrow{k} = (0, 0, -1)$. Pour faire simple, le plan $(O, \overrightarrow{i}, \overrightarrow{j})$ est supposé "vertical".

1. Opérations sur les vecteurs et les points

Nous aurons besoin d'effectuer toutes les opérations classiques sur les points et les vecteurs, que nous représenterons par des triplets de flottants : addition, soustraction, produit scalaire, distance, etc. Ces opérations se définissent sans difficulté.

1.1 L'espace vectoriel $\mathbb{R}^3$.

[3]
```
def add(u, v):
```

```
    return (u[0] + v[0], u[1] + v[1], u[2] + v[2])
```

[4]
```
add((1, 2, 3), (4, 5, 6))
```

```
(5, 7, 9)
```

[5]
```
def sub(u, v):
    return (u[0] - v[0], u[1] - v[1], u[2] - v[2])
```

[6]
```
sub((1, 2, 3), (4, 5, 6))
```

```
(-3, -3, -3)
```

[7]
```
def oppose(u):
    return (-u[0], -u[1], -u[2])
```

[8]
```
oppose((1, 2, 3))
```

```
(-1, -2, -3)
```

[9]
```
def prod(mu, u):
    return (mu * u[0], mu * u[1], mu * u[2])
```

[10]
```
prod(2, (1, 2, 3))
```

```
(2, 4, 6)
```

1.3 Vecteur $\overrightarrow{AB}$

[11]
```
def vecteur(A, B):
    return sub(B, A)
```

[12]
```
vecteur((1, 2, 3), (4, 5, 6))
```

```
(3, 3, 3)
```

1.4 Produit scalaire, grandeurs associées

[13]
```
def prodscal(u, v):
    return u[0] * v[0] + u[1] * v[1] + u[2] * v[2]
```

[14]
```
prodscal((1, 2, 3), (4, 5, 6))
```

```
32
```

[15]
```
def norme(u):
    return sqrt(prodscal(u, u))
```

[16]
```
norme((1, 2, 3))
```

```
3.7416573867739413
```

[17]
```
def normaliser(u):
    return prod(1 / norme(u), u)
```

[18]
```
normaliser((1, 2, 3))
```

```
(0.2672612419124244, 0.5345224838248488, 0.8017837257372732)
```

[19]
```
def norme2(u):
    return prodscal(u, u)
```

[20]
```
norme2((1, 2, 3))
```

[21]
```
def cosangle(u, v):
    return prodscal(u, v) / (norme(u) * norme(v))
```

[22]
```
cosangle((1, 2, 3), (4, 5, 6))
```

```
0.9746318461970762
```

Et le sinus de l'angle, me direz-vous ? Nous en aurons besoin lorsque nous aurons à modéliser les lois de Snell-Descartes pour la réfraction. Nous le ferons le moment venu.

[23]
```
def distance(A, B):
    return norme(vecteur(A, B))
```

[24]
```
distance((1, 2, 3), (4, 5, 6))
```

```
5.196152422706632
```

1.5 Produit vectoriel

[25]
```
def prodvect(u, v):
    x = u[1] * v[2] - u[2] * v[1]
    y = u[2] * v[0] - u[0] * v[2]
    z = u[0] * v[1] - u[1] * v[0]
    return (x, y, z)
```

[26]
```
prodvect((1, 2, 3), (4, 5, 6))
```

```
(-3, 6, -3)
```

2. Caméras

Pour observer une scène, il faut une **caméra**. La caméra est placée à une certaine position `pos`. Elle pointe dans une certaine direction `dirVue`. Ensuite, tout bon photographe fait en sorte que son appareil ne soit pas tout de travers. On associe à la caméra une base orthonormée directe $(\overrightarrow{u}, \overrightarrow{v}, \overrightarrow{w})$ telle que $\overrightarrow{w}$ soit opposé à la direction de vue et $\overrightarrow{u}$ soit horizontal. Enfin, il est souhaitable que la caméra ne soit pas à l'envers : le vecteur $\overrightarrow{v}$ doit pointer vers le haut.

Avoir une caméra c'est bien, mais il faut ensuite projeter le film. On projette sur un plan perpendiculaire à la direction de vue, à une distance d de la caméra. L'écran a une certaine résolution (mesurée en pixels) et une certaine étendue mesurée, disons, en mètres : les bornes de l'écran.

La classe `Camera` résume tout cela. Elle définit également une méthode `point`. Celle ci prend deux enrtiers i et j en paramètres, qui sont des coordonnées de pixels de l'écran. Elle renvoie le trpilet (x, y, z) des coordonnées du point concerné dans le repère canonique.

[27]
```
class Camera:

    def __init__(self, pos, dirVue, d, resolEcran, bornes):
        self.nx, self.ny = resolEcran
        self.xmin, self.xmax, self.ymin, self.ymax = bornes
        self.pos = pos
        self.dirVue = dirVue
        self.w = normaliser(oppose(dirVue))
        self.d = d
        x, y, z = dirVue
        self.u = normaliser((-self.w[2], 0, self.w[0]))
        self.v = prodvect(self.w, self.u)
        if self.v[1] < 0:
            self.u = oppose(self.u)
            self.v = oppose(self.v)

    def point(self, i, j):
        x = self.xmin + i * (self.xmax - self.xmin) / self.nx
        y = self.ymin + j * (self.ymax - self.ymin) / self.ny
        z = -self.d
        P = add(self.pos,add(prod(x, self.u),add(prod(y,
self.v),prod(z, self.w))))
        return P
```

Voici la caméra que nous utiliserons de façon courante dans le notebook.

[28]
```
cameraSD = Camera(pos=(0, 0, 5), dirVue=(0, 0, -1), d = 5,
resolEcran=(300, 200), bornes=(-3, 3, -2, 2))
```

[29]
```
cameraSD.u, cameraSD.v, cameraSD.w
```

```
((1.0, -0.0, -0.0), (-0.0, 1.0, -0.0), (0.0, 0.0, 1.0))
```

[30]
```
cameraSD.point(163, 189)
```

```
(0.2599999999999998, 1.7799999999999998, 0.0)
```

Je définis également une caméra HD, possédant un écran avec plus de pixels.

[31]
```
cameraHD = Camera(pos=(0, 0, 5), dirVue=(0, 0, -1), d = 5,
resolEcran=(600, 400), bornes=(-3, 3, -2, 2))
```

Note sur les temps de calcul : au fur et à mesure que nous avancerons dans l'étude du lancer de rayons, nos algorithmes deviendront de plus en plus gourmands en temps de calcul. La caméra HD, par exemple, nécessite un temps de calcul 4 fois supérieur à la caméra classique. Plus la scène que nous voulons photographier possède d'obkets, plus elle possède de lumières, plus les calculs seront longs.

Lorsque nous aborderons la réflexion, puis, enfin, la transparence, la complexité des algoriothmes va "exploser". La prise en compte de la transparence donne à l'algorithme de lancer de rayon une complexité ... exponentielle.

Moralité : restons modestes. Je travaillerai sur des scènes ayant 3 ou 4 objets, et une ou deux lumières. La caméra HD ne sera utilisée que rarement. Mais libre à vous, bien sûr, de faire tous les essais que vous voudrez !

[32]
```
cameraUHD = Camera(pos=(0, 0, 5), dirVue=(0, 0, -1), d = 5,
resolEcran=(1200, 800), bornes=(-3, 3, -2, 2))
```

3. Objets et scènes

Une scène est composée de plusieurs objets. Nous ne considérerons que des objets très simples : les **sphères** et les **plans**. Rien ne vous empêche de considérer d'autres types d'objets, notre programme va être structuré pour qu'il soit très facile de rajouter de nouveaux types d'objets : cylindres, polygones, etc.

Nous allons donc définir une classe **Sphere** et une classe **Plan**. Une sphère est caractérisée par son centre et son rayon. Un plan, quant à lui, est défini par un point `org` et un vecteur normal `normal`.

3.1 Sphères

[33]
```
class Sphere:

    def __init__(self, centre, rayon):
        self.centre = centre
        self.rayon = rayon
```

[34]
```
s = Sphere(centre=(1,2,3), rayon=2)
```

[35]
```
s.centre, s.rayon
```

```
((1, 2, 3), 2)
```

3.2 Plans

[36]
```
class Plan:

    def __init__(self, org, normal):
        self.org = org
        self.normal = normal
```

[37]
```
p = Plan(org=(1, 1, 1), normal=(2, 0, 1))
```

[38]
```
p.org, p.normal
```

```
((1, 1, 1), (2, 0, 1))
```

3.3 Rayons

Si notre algorithme du jour s'appelle le lancer de rayons, il va bien falloir se demander ce qu'est un rayon : c'est une demi-droite. Un rayon est donc défini par une origine `org` et une direction `u`.

[39]
```
class Rayon:

    def __init__(self, org, u):
        self.org = org
        self.dir = u
```

[40]
```
r = Rayon(org=(1, 2, 3), u=(1, 1, 1))
```

[41]
```
r.org, r.dir
```

```
((1, 2, 3), (1, 1, 1))
```

3.4 Scène

Définissions enfin la classe `Scene`. Une scène est composée d'objets ... et de lumières, parce que sinon on n'y voit rien :-). Pour l'instant, ne nous occupons pas des lumières, cela viendra en temps voulu.

[42]
```
class Scene:

    def __init__(self, objets, lumieres):
        self.objets = objets
        self.lumieres = lumieres
```

4. Calculs d'intersections

Il est temps d'expliquer le principe du lancer de rayon. Nous disposons d'une caméra avec un écran, et d'une scène. Pour chaque pixel de l'écran, la caméra lance un rayon qui passe par ce pixel. Le rayon intersecte la scène en un ou plusieurs objets. On calcule alors la valeur de l'éclairement pour ce pixel en fonction des points des objets frappés par le rayon.

Bien entendu, tout est dans le "On calcule alors la valeur de l'éclairement". Le reste du notebook et le notebook suivant sont là pour mettre en place des modèles permettant ce calcul. Mais pour l'instant, occupons nous d'intersections !

4.1 Intersecter un rayon avec une sphere

Soit r un rayon d'origine le point A, dirigé par un vecteur $\overrightarrow{u}$. Un point quelconque du rayon est de la forme $M = A + \mu\overrightarrow{u}$, où μ est un réel positif ou nul.

Soit S la sphère de centre B et de rayon R. Le point M est sur la sphère si et seulement si $||\overrightarrow{BM}|| = R$, ce qui s'écrit encore $||\overrightarrow{BA} + \mu\overrightarrow{u}|| = R$. Élevons au carré pour obtenir $< \overrightarrow{BA} + \mu\overrightarrow{u}, \overrightarrow{BA} + \mu\overrightarrow{u} >= R^2$, où les crochets désignent le produit scalaire. On développe :

$$||\overrightarrow{u}||^2\mu^2 + 2 < \overrightarrow{BA}, \overrightarrow{u} > \mu + ||\overrightarrow{BA}||^2 - R^2 = 0$$

On obtient donc une équation du second degré en μ du type $a\mu^2 + b\mu + c = 0$, où $a = ||\overrightarrow{u}||^2, b = 2 < \overrightarrow{BA}, \overrightarrow{u} >$ et $c = ||\overrightarrow{BA}||^2 - R^2$. Selon le signe du discriminant de cette équation, le rayon a 0, 1 ou deux points d'intersection avec la sphère.

Nous redéfinissons donc la classe `Sphere`pour qu'une sphère soit capable de calculer son intersection avec un rayon. La méthode `intersection`renvoie un couple formé de la sphère et de la liste, éventuellement vide, des points d'intersection. La méthode reprend point par point le cheminement du calcul ci-dessus.

J'en profite pour rajouter une méthode `vecteur_normal` qui calcule un vecteur normal à une sphère en un point (ceci est immédiat).

[43]
```
class Sphere:

    def __init__(self, centre, rayon):
        self.centre = centre
        self.rayon = rayon

    def intersection(self, rayon):
```

```
            A, u = rayon.org, rayon.dir
            B, R = self.centre, self.rayon
            v = vecteur(B, A)
            a = norme(u) ** 2
            b = 2 * prodscal(u, v)
            c = norme(v) ** 2 - R ** 2
            Delta = b ** 2 - 4 * a * c
            if Delta < 0: return []
            elif Delta == 0:
                mu = - b / (2 * a)
                if mu > 1e-5:
                    return [(self, add(A, prod(mu, u)))]
                else: return 0
            elif Delta > 0:
                s = []
                delta = sqrt(Delta)
                mu1 = (-b - delta) / (2 * a)
                if mu1 > 1e-5:
                    s.append((self, add(A, prod(mu1, u))))
                mu2 = (-b + delta) / (2 * a)
                if mu2 > 1e-5:
                    s.append((self, add(A, prod(mu2, u))))
                return s

        def vecteur_normal(self, A):
            return vecteur(self.centre, A)
```

[44]
```
s = Sphere((1, 0, -3), -2)
s.intersection(Rayon((0, 0, 5), (0, 0, -1)))
```

```
[(<__main__.Sphere at 0x1127cf7b8>, (0.0, 0.0, -1.2679491924311188)),
 (<__main__.Sphere at 0x1127cf7b8>, (0.0, 0.0, -4.732050807568882))]
```

4.2 Intersecter un rayon avec un plan

Soit r un rayon d'origine le point A, dirigé par un vecteur $\overrightarrow{u}$. Un point quelconque du rayon est de la forme $M = A + \mu\overrightarrow{u}$, où μ est un réel positif ou nul.

Soit P le plan passant par B et de vecteur normal $\overrightarrow{n}$. Le point M est dans le plan P si et seulement si $< \overrightarrow{BM}, \overrightarrow{n} >= 0$, ou encore $< \overrightarrow{BA} + \mu\overrightarrow{u}, \overrightarrow{n} >= 0$. On développe pour obtenir une équation de degré 1 en μ :

$$< \overrightarrow{BA}, \overrightarrow{n} > + < \overrightarrow{u}, \overrightarrow{n} > \mu = 0$$

Cette équation possède en général une unique solution sauf, bien évidemment, si les vecteur $\overrightarrow{u}$ et $\overrightarrow{n}$ sont orthogonaux, puisque dans ce cas le rayon est parallèle au plan.

Comme pour la classe `Sphere`, redéfinissons la classe `Plan`.

[45]
```
class Plan:

    def __init__(self,org, normal):
        self.org = org
        self.normal = normal

    def intersection(self, rayon):
        A, u = rayon.org, rayon.dir
        B, n = self.org, self.normal
        v = vecteur(B, A)
        a = prodscal(u, n)
        b = prodscal(v, n)
        if a == 0: return []
        else:
            mu = - b / a
            if mu > 1e-5:
                return [(self, add(A, prod(mu, u)))]
            else:
                return []

    def vecteur_normal(self, A):
        return self.normal
```

4.3 Premiers tests

Créons un écran (une matrice de pixels) et une scène.

[46]
```
def ecran(m, n):
    t = m * [None]
    for k in range(m):
        t[k] = n * [(0.3, 0.3, 1.)]
    return t
```

[47]
```
ecran(5, 3)
```

```
[[(0.3, 0.3, 1.0), (0.3, 0.3, 1.0), (0.3, 0.3, 1.0)],
 [(0.3, 0.3, 1.0), (0.3, 0.3, 1.0), (0.3, 0.3, 1.0)],
```

```
[(0.3, 0.3, 1.0), (0.3, 0.3, 1.0), (0.3, 0.3, 1.0)],
[(0.3, 0.3, 1.0), (0.3, 0.3, 1.0), (0.3, 0.3, 1.0)],
[(0.3, 0.3, 1.0), (0.3, 0.3, 1.0), (0.3, 0.3, 1.0)]]
```

Notre scène comporte trois sphères posées sur un plan.

[48]
```
sphere1 = Sphere(centre=(2, 0, -5), rayon=2)
sphere2 = Sphere(centre=(-1, -1, -3), rayon=1)
sphere3 = Sphere(centre=(0, 1, -20), rayon=3)
plan = Plan(org=(0, -2, 0), normal=(0, 1, 0))
scene = Scene([sphere1, sphere2, sphere3, plan], [])
```

La fonction `lancer0` lance des rayons depuis la caméra vers des pixels de l'écran. Lorsqu'un rayon rencontre un objet de la scène, on allume le pixel correspondant en blanc.

Deux mots sur les couleurs : une couleur est représentée par un triplet (r, g, b) où $r, g, b \in [0, 1]$ représentent des quantités de rouge, de vert et de bleu. Ainsi, $(0, 0, 0)$ représente la couleur noire, $(1, 1, 1)$ la couleur blanche, et, par exemple, $(1, 0, 0)$ la couleur rouge.

[49]
```
def lancer0(scene, camera):
    t = ecran(camera.ny, camera.nx)
    for i in range(camera.nx):
        for j in range(camera.ny):
            M = camera.point(i, j)
            r = Rayon(camera.pos, vecteur(camera.pos, M))
            s = []
            for objet in scene.objets:
                s += objet.intersection(r)
            if s != []: t[camera.ny - 1 - j][i] = (1., 1., 1.)
    plt.axis('off')
    plt.imshow(t)
    plt.show()
```

[50]
```
lancer0(scene, cameraSD)
```

Évidemment, tout ceci est pour l'instant *très* décevant. C'est tout à fait normal, puisque nous n'avons pas encore mis en place de modèle d'éclairement. Nous allons faire cela progressivement.

5. Illumination

5.1 La classe `Lumiere`

Qu'est-ce qu'une source de lumière ? Nous nous considérerons ici que des sources ponctuelles. Une telle source est caractérisée par sa position, et par l'intensité de rouge, de vert, et de bleu qu'elle émet (sa couleur, si vous préférez). D'où la classe `Lumiere`.

[51]
```
class Lumiere:

    def __init__(self, org, I):
        self.org = org
        self.I = I
```

5.2 Éclairement diffus

Lorsque nous regardons un objet, nous percevons la lumière émise par cet objet.

Un objet qui n'est pas une source de lumière est visible parce qu'il réfléchit la lumière qui lui est envoyée par d'autres sources. Il existe divers types de réflexion. Le premier type est la **réflexion diffuse**, étudiée par Lambert au dix-huitième siècle : la quantité d'énergie reçue par une surface dépend de l'angle d'arrivée de la lumière sur cette surface. Si la lumière arrive perpendiculairement à la surface, cette énergie est maximale. Si la lumière arrive tangentiellement, l'énergie reçue est nulle. Ceci mène au **modèle d'éclairement de Lambert** :

$$L = k_d I \max(0, < \overrightarrow{u}, \overrightarrow{v} >)$$

où

- k_d est le coefficient diffus, propre à l'objet illuminé. Plus k_d est grand, plus l'objet réfléchit la lumière.
- I est l'intensité de la source lumineuse (en fait un triplet (r, g, b) d'intensités)
- $\overrightarrow{u}$ est le vecteur unitaire normal à la surface
- $\overrightarrow{v}$ est le vecteur unitaire dirigé vers la source lumineuse.

Pourquoi le max ? Un produit scalaire négatif correspond à une lumière arrivant de l'arrière (l'angle est supérieur à $\frac{\pi}{2}$). L'objet n'est alors pas éclairé.

Laissons tomber pour l'instant le coefficient k_d (prenons-le égal à 1) et ruons nous sur un test.

[52]
```
def diffuse(A, objet, scene):
    I = (0, 0, 0)
    for L in scene.lumieres:
        v = vecteur(A, L.org)
        u = objet.vecteur_normal(A)
        I = add(I, prod(max(0, cosangle(u, v)), L.I))
    return I
```

Prenons une scène composée de trois sphères, d'un plan et de deux sources lumineuses blanches (changez les paramètres, testez, testez !).

[53]
```
sphere1 = Sphere(centre=(2, 0, -5), rayon=2)
sphere2 = Sphere(centre=(-1, -1, -3), rayon=1)
```

```
sphere3 = Sphere(centre=(0, 1, -20), rayon=3)
plan = Plan(org=(0, -2, 0), normal=(0, 1, 0))
lum1 = Lumiere(org=(-100, 20, 100), I=(0.7, 0.7, 0.7))
lum2 = Lumiere(org=(100, 100, 100), I=(0.7, 0.7, 0.7))
scene = Scene([sphere1, sphere2, sphere3, plan], [lum1, lum2])
```

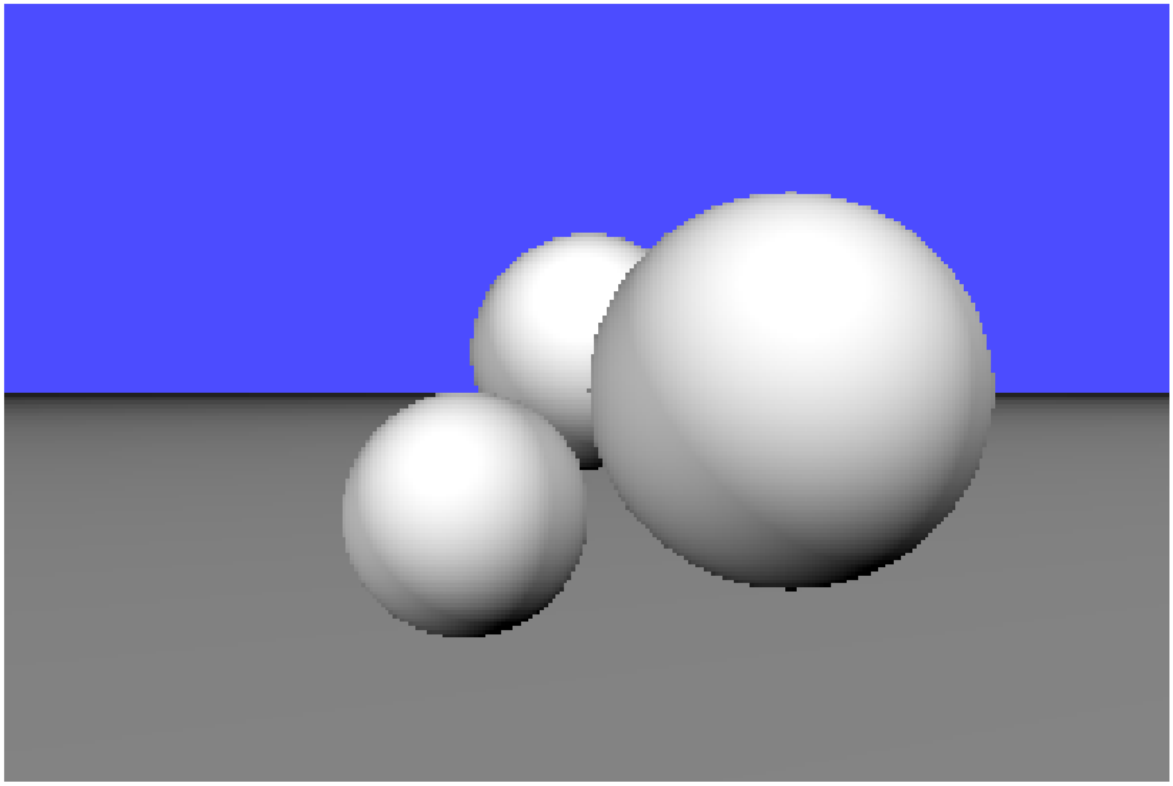
Réécrivons la fonction de lancer de rayons. Pour chaque rayon lancé de la caméra, on sélectionne, s'il y en a un, l'objet le plus proche frappé par le rayon. On calcule l'éclairement du point frappé au moyen du modèle de Lambert et on allume le pixel correspondant de l'écran à la couleur adéquate.

[54]
```
def lancer1(scene, camera):
    t = ecran(camera.ny, camera.nx)
    for i in range(camera.nx):
        for j in range(camera.ny):
            M = camera.point(i, j)
            r = Rayon(camera.pos, vecteur(camera.pos, M))
            s = []
            for objet in scene.objets:
                s += objet.intersection(r)
            if s != []:
                (objet, A) = plusProche(camera.pos, s)
                Ix, Iy, Iz = diffuse(A, objet, scene)
                t[camera.ny - 1 - j][i] = (min(Ix, 1), min(Iy,
1), min(Iz, 1))
    plt.axis('off')
    plt.imshow(t)
    plt.show()
```

La fonction `plusProche` renvoie, parmi une liste de couples (objet, point), le couple dont le point est le plus proche d'un point B donné.

[55]
```
def plusProche(B, objets):
    obj0, A0 = objets[0]
    for obj, A in objets:
        if distance(B, A) < distance(B, A0):
            obj0 = obj
            A0 = A
    return obj0, A0
```

[56]
```
lancer1(scene, cameraSD)
```



Ah oui, c'est quand même bien mieux que notre première fonction de lancer !

5.3 Éclairement spéculaire

L'éclairement diffus est indépendant de la position de la caméra. Cependant, beaucoup d'objets réels brillent plus ou moins et produisent des **réflexions spéculaires** qui, elles, dépendent de l'endroit d'où vous regardez l'objet. Alors que le modèle de Lambert est adapté aux surfaces mates (le caoutchouc par exemple), le modèle que nous allons considérer est adapté aux surfaces qui brillent (l'acier poli par exemple). C'est le **modèle de Blinn-Phong**.

$$L = k_s I \max(0, < \overrightarrow{v}, \overrightarrow{h} >)^p$$

où

- k_s est un coefficient propre à l'objet, le coefficient spéculaire
- I est l'intensité de la source lumineuse
- $\overrightarrow{v}$ est le vecteur unitaire dirigé vers la source lumineuse
- p est un paramètre de brillance

Quant au vecteur $\overrightarrow{h}$, il est défini comme suit : soit $\overrightarrow{u}$ le vecteur normal à la surface. Soit $\overrightarrow{w}$ le vecteur unitaire dirigé vers l'observateur. Alors

$$\overrightarrow{h} = \frac{\overrightarrow{u} + \overrightarrow{w}}{||\overrightarrow{u} + \overrightarrow{w}||}$$

Quelles valeurs prendre pour p ? Plus p est grand plus ça brille. Voici quelques valeurs typiques de p :

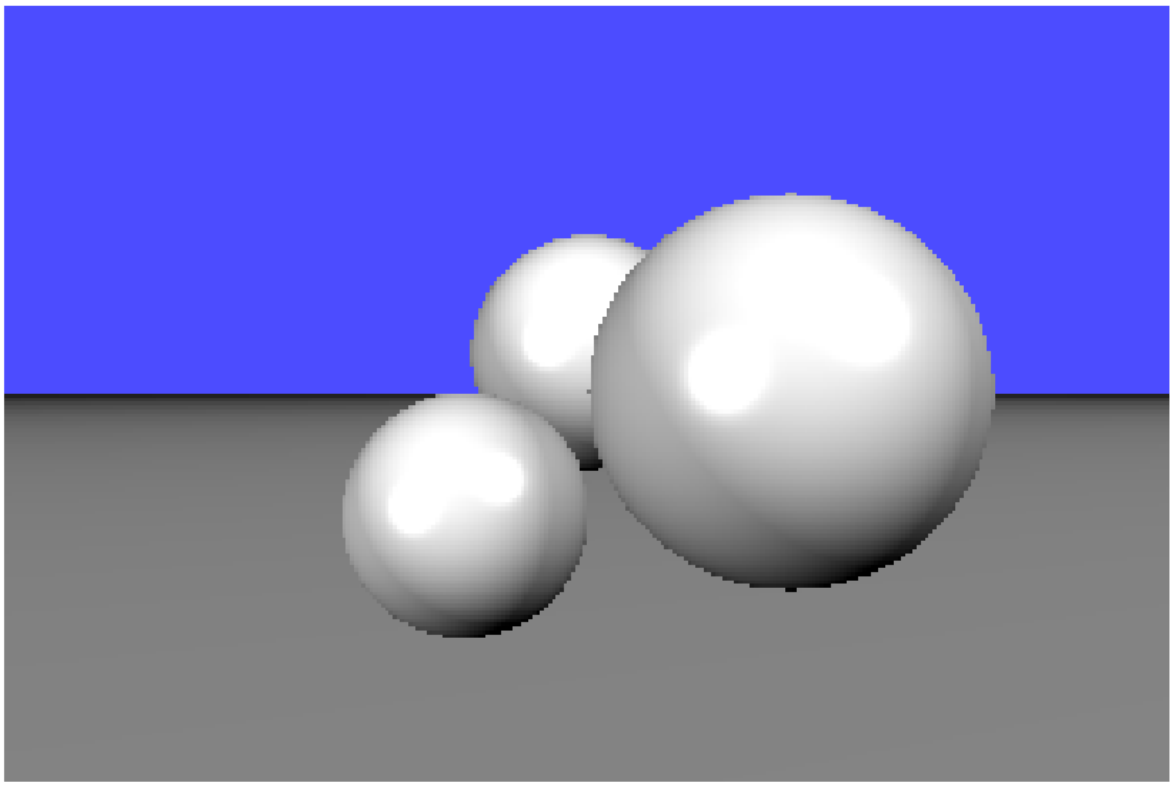
- 10 : coquille d'oeuf
- 100 : légèrement brillant
- 1000 : vraiment brillant
- 10000 : quasi-miroir

Je prendrai 100 dans ce qui suit.

[57]
```
def phong(A, obs, objet, scene):
    I = (0, 0, 0)
    for L in scene.lumieres:
        v = vecteur(A, L.org)
        v = prod(1 / norme(v), v)
        u = objet.vecteur_normal(A)
        w = vecteur(A, obs)
        w = prod(1 / norme(w), w)
        h = add(v, w)
        I = add(I, prod(max(0, cosangle(u, h)) ** 100, L.I))
    return I
```

[58]
```
def lancer2(scene, camera):
    t = ecran(camera.ny, camera.nx)
    for i in range(camera.nx):
        for j in range(camera.ny):
            M = camera.point(i, j)
            r = Rayon(camera.pos, vecteur(camera.pos, M))
            s = []
            for objet in scene.objets:
                s += objet.intersection(r)
            if s != []:
                (objet, A) = plusProche(camera.pos, s)
                Ix, Iy, Iz = add(diffuse(A, objet, scene),
phong(A, camera.pos, objet, scene))
                t[camera.ny - 1 - j][i] = (min(Ix, 1), min(Iy,
1), min(Iz, 1))
    plt.axis('off')
    plt.imshow(t)
    plt.show()
```

[59]
```
lancer2(scene, cameraSD)
```



Remarque : Je profite du dessin qui vient d'apparaître pour parler d'un phénomène dont je ne discuterai plus dans la suite ...

Les bords des sphères sont crénelés. C'est le phénomène d'**aliasage**. Ceci peut être résolu en découpant chaque pixel en 4 (par exemple) et en lançant des rayons sur chacun des quarts de pixels puis en faisant les moyennes des éclairements obtenus. Cette méthode d'anti-aliasage multiplie évidemment le temps de calcul par 4.

5.4 Éclairement ambiant

Il existe enfin un dernier type d'éclairement : **l'éclairement ambiant**. Une surface qui ne reçoit aucune lumière est normalement complètement noire. Dans la réalité ce n'est pas le cas, les réflexions indirectes d'autres surfaces (entre autres) produisent un léger éclairement. Cette "lumière ambiante" peut être ajoutée à notre modèle d'éclairement :

$$L = k_a I_a$$

où

- k_a est un coefficient ambiant propre à l'objet
- I_a est l'intensité de la lumière ambiante causée par la source lumineuse.

Remarquons que nous avons maintenant trois paramètres, k_a, k_d et k_s, propres à chaque objet, et deux paramètres, I_a et I propres à chaque source lumineuse. Il est temps de

réécrire nos classes `Sphere`, `Plan` et `Lumiere` pour tenir compte de ces nouveaux paramètres.

[60]
```
class Sphere:

    def __init__(self, centre, rayon, color, ka, kd, ks):
        self.centre = centre
        self.rayon = rayon
        self.color = color
        self.ka = ka
        self.kd = kd
        self.ks = ks

    def intersection(self, rayon):
        A, u = rayon.org, rayon.dir
        B, R = self.centre, self.rayon
        v = vecteur(B, A)
        a = norme(u) ** 2
        b = 2 * prodscal(u, v)
        c = norme(v) ** 2 - R ** 2
        Delta = b ** 2 - 4 * a * c
        if Delta < 0: return []
        elif Delta == 0:
            mu = - b / (2 * a)
            if mu > 1e-5:
                return [(self, add(A, prod(mu, u)))]
            else:
                return []
        elif Delta > 0:
            delta = sqrt(Delta)
            mu1 = (-b - delta) / (2 * a)
            mu2 = (-b + delta) / (2 * a)
            s = []
            if mu1 > 1e-5: s.append((self, add(A, prod(mu1, u))))
            if mu2 > 1e-5: s.append((self, add(A, prod(mu2, u))))
            return s

    def vecteur_normal(self, A):
        return vecteur(self.centre, A)
```

[61]
```
class Plan:

    def __init__(self, org, normal, color, ka, kd, ks):
        self.org = org
        self.normal = normal
        self.color = color
        self.ka = ka
        self.kd = kd
        self.ks = ks
```

```
    def intersection(self, rayon):
        A, u = rayon.org, rayon.dir
        B, n = self.org, self.normal
        v = vecteur(B, A)
        a = prodscal(u, n)
        b = prodscal(v, n)
        if a == 0: return []
        else:
            mu = - b / a
            if mu > 1e-5:
                M = add(A, prod(mu, u))
                return [(self, M)]
            else:
                return []

    def vecteur_normal(self, A):
        return self.normal
```

[62]
```
class Lumiere:

    def __init__(self, org, I, Ia):
        self.org = org
        self.I = I
        self.Ia = Ia
```

Les modèles déclairement doivent eux-aussi être réécrits pour tenir compte des nouvelles caractéristiques des objets et des lumières.

[63]
```
def diffuse(A, objet, scene):
    I = (0, 0, 0)
    for L in scene.lumieres:
        v = vecteur(A, L.org)
        u = objet.vecteur_normal(A)
        I1 = prod(objet.kd, L.I)
        I = add(I, prod(max(0, cosangle(u, v)), I1))
    return I
```

[64]
```
def phong(A, obs, objet, scene):
    I = (0, 0, 0)
    for L in scene.lumieres:
        v = vecteur(A, L.org)
        v = prod(1 / norme(v), v)
        u = objet.vecteur_normal(A)
        w = vecteur(A, obs)
        w = prod(1 / norme(w), w)
```

```
            h = add(v, w)
            I1 = prod(objet.ks, L.I)
            I = add(I, prod(max(0, cosangle(u, h)) ** 100, I1))
        return I
```

[65]
```
def ambient(A, objet, scene):
    I = (0, 0, 0)
    for L in scene.lumieres:
        I = add(I, prod(objet.ka, L.Ia))
    return I
```

La fonction `illumination` additionne les trois modèles.

Remarque : si nos sources lumineuses sont trop intenses, des phénomènes de saturation peuvent apparaître dans le rendu. Une intensité lumineuse strictement supérieure à 1 est ramenée à 1 : on a un phénomène de **surexposition**.

[66]
```
def illumination(A, obs, objet, scene):
    I = ambient(A, objet, scene)
    I = add(I, diffuse(A, objet, scene))
    I = add(I, phong(A, obs, objet, scene))
    Ix, Iy, Iz = I
    cx, cy, cz = objet.color
    return (Ix * cx, Iy * cy, Iz * cz)
```

Maintenant, notre nouvelle fonction de lancer. L'occasion est trop belle, je scinde la fonction en deux parties pour la rendre plus lisible (et préparer le terrain pour la suite :-)).

[67]
```
def lancer3(scene, camera):
    t = ecran(camera.ny, camera.nx)
    for i in range(camera.nx):
        for j in range(camera.ny):
            M = camera.point(i, j)
            r = Rayon(camera.pos, vecteur(camera.pos, M))
            Ix, Iy, Iz = lancerRayon(r, camera)
            t[camera.ny - 1 - j][i] = (min(Ix, 1), min(Iy, 1),
min(Iz, 1))
    plt.axis('off')
    plt.imshow(t)
    plt.show()
```

La fonction de lancer est donc extrêmenent courte et lisible. Reste à écrire `lancerRayon`.

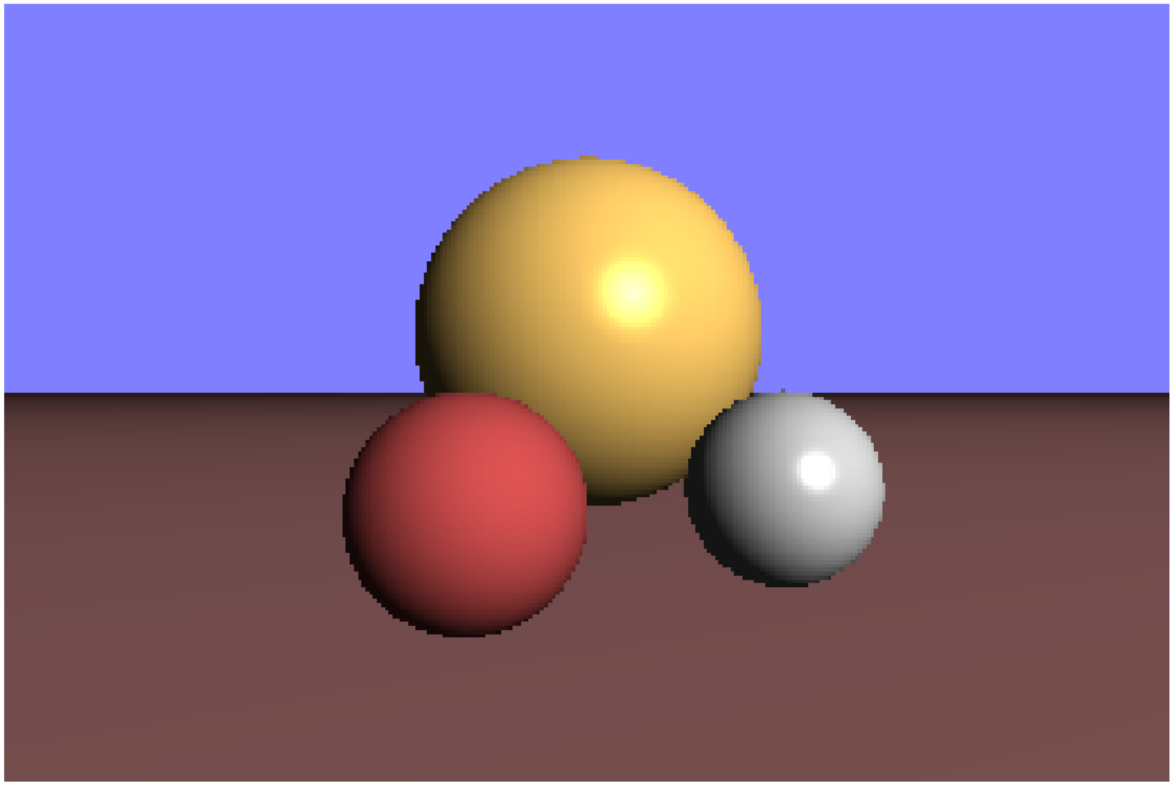
[68]
```
def lancerRayon(r, camera):
    s = []
    for objet in scene.objets:
        s += objet.intersection(r)
    if s != []:
        (objet, A) = plusProche(camera.pos, s)
        return illumination(A, camera.pos, objet, scene)
    else: return (0.5, 0.5, 1.)
```

Voici notre scène.

[69]
```
sphere1 = Sphere(centre=(2, -1, -5), rayon=1, color=(0.8, 0.8,
0.8), ka=0.1, kd=1, ks=1)
sphere2 = Sphere(centre=(-1, -1, -3), rayon=1, color=(0.8, 0.3,
0.3), ka=0.1, kd=1, ks=0.)
sphere3 = Sphere(centre=(0, 1, -12), rayon=3, color=(1, 0.8,
0.4), ka=0.1, kd=1, ks=1)
plan = Plan(org=(0, -2, 0), normal=(0, 1, 0), color=(0.9, 0.6,
0.6), ka=0.1, kd=1, ks=0)
lum1 = Lumiere(org=(60, 50, 100), I=(1, 1, 1), Ia=(1, 1, 1))
scene = Scene([sphere1, sphere2, sphere3, plan], [lum1])
```

Lançons des rayons !

[70]
```
lancer3(scene, cameraSD)
```



Il y a un petit souci : les sphères sont soit-disant posées sur le plan mais on a l'impression qu'elles flottent dans les airs. Que manque-t-il ? Il manque les **ombres**.

6. Ombres

[71]
```
def diffuse(A, objet, scene):
    I = (0, 0, 0)
    for L in scene.lumieres:
        r = Rayon(A, L.org)
        black = False
        for obj in scene.objets:
            if obj.intersection(r) != []:
                black = True
                break
        if not black:
            v = vecteur(A, L.org)
            u = objet.vecteur_normal(A)
            I1 = prod(objet.kd, L.I)
            I = add(I, prod(max(0, cosangle(u, v)), I1))
    return I
```

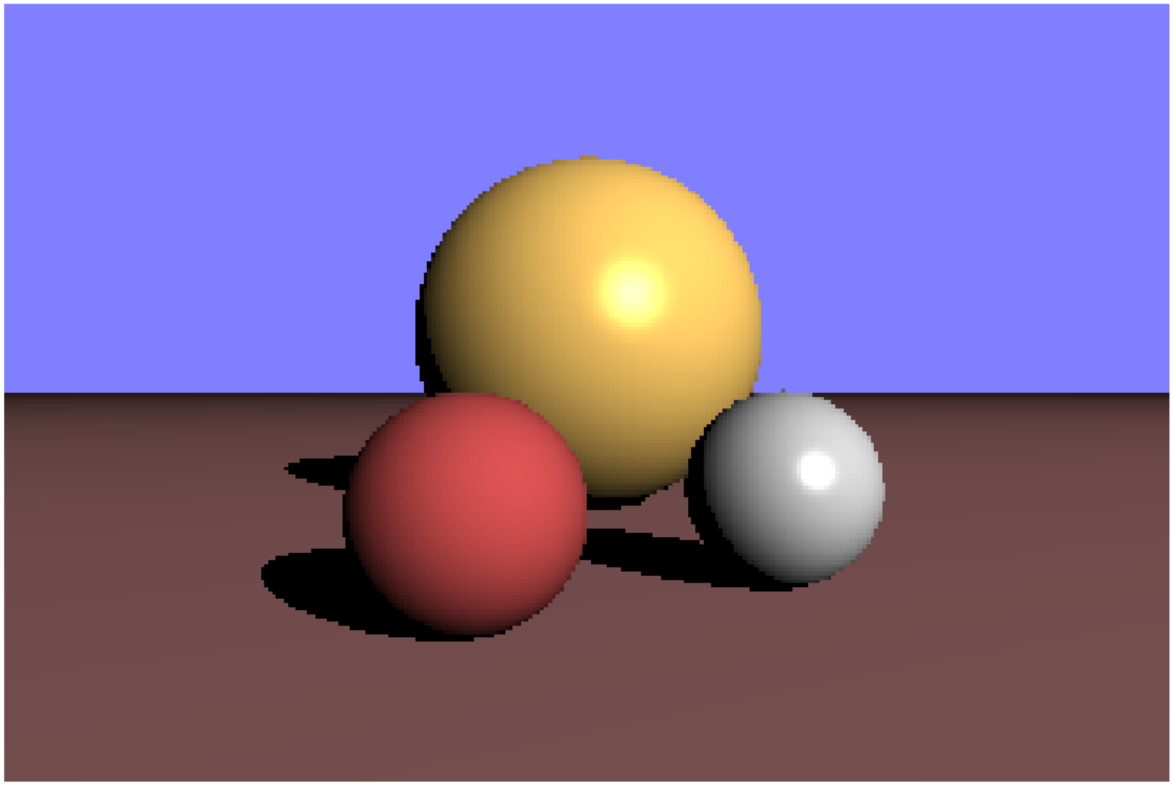
[72]
```
def phong(A, obs, objet, scene):
```

```
        I = (0, 0, 0)
        for L in scene.lumieres:
            r = Rayon(A, L.org)
            black = False
            for obj in scene.objets:
                if obj.intersection(r) != []:
                    black = True
                    break
            if not black:
                v = vecteur(A, L.org)
                v = prod(1 / norme(v), v)
                u = objet.vecteur_normal(A)
                w = vecteur(A, obs)
                w = prod(1 / norme(w), w)
                h = add(v, w)
                h = prod(1 / norme(h), h)
                I1 = prod(objet.ks, L.I)
                I = add(I, prod(max(0, cosangle(u, h)) ** 100, I1))
        return I
```

[73]
```
    def ambient(A, objet, scene):
        I = (0, 0, 0)
        for L in scene.lumieres:
            r = Rayon(A, L.org)
            black = False
            for obj in scene.objets:
                if obj.intersection(r) != []:
                    black = True
                    break
            if not black:
                I = add(I, prod(objet.ka, L.Ia))
        return I
```

[74]
```
    def illumination(A, obs, objet, scene):
        I = ambient(A, objet, scene)
        I = add(I, diffuse(A, objet, scene))
        I = add(I, phong(A, obs, objet, scene))
        Ix, Iy, Iz = I
        cx, cy, cz = objet.color
        return (Ix * cx, Iy * cy, Iz * cz)
```

[75]
```
    lancer3(scene, cameraSD)
```



Et voilà, les sphères ont atterri sur le plan.

Quelle est l'étape suivante ? Rappelez-vous le modèle de Blin-Phong pour l'éclairement spéculaire. Une surface brillante est souvent aussi **réfléchissante** : elle renvoie l'image des objets situés dans son voisinage. Nous allons nous attaquer au problème de la réflexion, qui sera le dernier problème évoqué dans ce notebook.

7. Réflexion

Comment prendre en compte les phénomènes de réflexion ? Lançons un rayon : ce rayon frappe un objet, puis se réfléchit selon les lois de Descartes. Le rayon réfléchi peut à nouveau se réfléchir sur un second objet, et ainsi de suite. La fonction `lancer4` est quasiment analogue à la fonction `lancer3`, à ceci près qu'elle prend également en paramètre un entier `prof` permettant d'éviter un nombre infini de réflexions.

[76]
```
def lancer4(scene, camera, prof, prof0):
    t = ecran(camera.ny, camera.nx)
    for i in range(camera.nx):
        for j in range(camera.ny):
            M = camera.point(i, j)
            u = vecteur(camera.pos, M)
            r = Rayon(camera.pos, u)
```

```
                Ix, Iy, Iz = lancerRayon(r, 1, prof, prof0)
                t[camera.ny - 1 - j][i] = (min(Ix, 1), min(Iy, 1),
min(Iz, 1))
    plt.axis('off')
    plt.imshow(t)
    plt.imsave('scene.png', t)
    plt.show()
```

Tout se passe dans la fonction `lancerRayon`. Celle-ci prend 4 paramètres :

- un rayon `r`
- un flottant `att` qui est un facteur d'atténuation : chaque réflexion diminue l'intensité du rayon lumineux.
- le fameux paramètre `prof` qui contrôle le nombre de réflexions.

On lance comme on en a l'habitude le rayon r et on frappe un objet. Si `prof` est non nul, on réfléchit ce rayon et on rappelle récursivement `lancerRayon` avec une prof(ondeur) diminuée de 1 et un facteur d'atténuation modifié en conséquence.

Rappelons à toutes fins utiles la loi de la réflexion : si le rayon a pour direction $\overrightarrow{u}$ et frappe une surface en un point d'une surface de vecteur normal $\overrightarrow{n}$, ce rayon se réfléchit selon une direction $\overrightarrow{v}$, où

$$\overrightarrow{v} = \overrightarrow{u} - 2 < \overrightarrow{u}, \overrightarrow{n} > \overrightarrow{n}$$

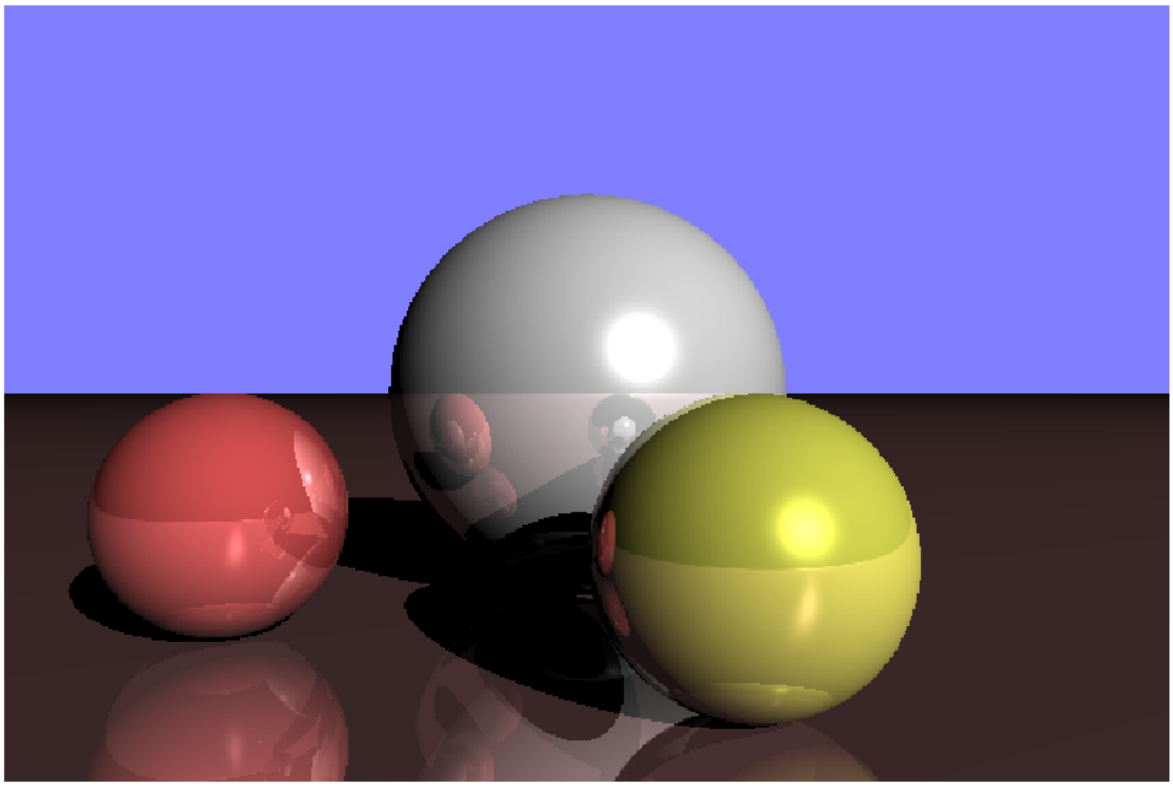
[77]
```
def lancerRayon(r, att, prof, prof0):
    s = []
    for objet in scene.objets:
        s += objet.intersection(r)
    if s != []:
        (objet, A) = plusProche(r.org, s)
        I =  prod(att, illumination(A, r.org, objet, scene))
        if prof > 0:
            u = r.dir
            u = normaliser(u)
            n = objet.vecteur_normal(A)
            n = normaliser(n)
            v = sub(u, prod(2 * prodscal(u, n), n))
            r1 = Rayon(A, v)
            I1 = lancerRayon(r1, 0.5 * att, prof - 1, prof0)
            I = add(I, I1)
        return I
    elif prof == prof0: return (0.5, 0.5, 1.)
    else: return (0., 0., 0.)
```

[78]
```
sphere1 = Sphere(centre=(1, -1, -1), rayon=1, color=(0.8, 0.8,
0.3), ka=0.1, kd=1, ks=1)
```

```
sphere2 = Sphere(centre=(-3, -1, -3), rayon=1, color=(0.8, 0.3,
0.3), ka=0.1, kd=1, ks=0.)
sphere3 = Sphere(centre=(0, 0, -5), rayon=2, color=(0.8, 0.8,
0.8), ka=0.1, kd=1, ks=1)
plan = Plan(org=(0, -2, 0), normal=(0, 1, 0), color=(0.9, 0.6,
0.6), ka=0.1, kd=0.5, ks=0.6)
lum1 = Lumiere(org=(60, 50, 100), I=(1, 1, 1), Ia=(0.5, 0.5,
0.5))
scene = Scene([sphere1, sphere2, sphere3, plan], [lum1])
```

Ce sera notre dernière expérience pour ce notebook, alors prenons notre caméra Haute Définition. Soyez patients (une vingtaine de secondes sur ma machine).

[79]
```
lancer4(scene, cameraHD, 3, 3)
```



Et maintenant ? Le notebook suivant abordera la question de la **transparence**. Certains objets se laissent partiellement traverser par la lumière : un rayon arrivant sur l'objet se subdivise en deux rayons.

- L'un des deux est un rayon **réfléchi**.
- L'autre est un rayon **réfracté**.

Les lois de Snell-Descartes seront nos amies :-).

```
[ ]
```