

# Table des matières

2  
9  
14

- Chapitre 1  
Introduction
- Chapitre 2  
Intersections
- Chapitre 3  
Vers une première image

Chapitre

# 1

# Introduction

## Sommaire

- 1.1 Objectifs, 3
- 1.2 Travail attendu, 4
- 1.3 Prise en main du code, 4
  - 1.3.1 Ouverture/Modification et exécution d'un code Python, 5
  - 1.3.2 Création d'une image aléatoire, 5
  - 1.3.3 Vers un deuxième code s'adaptant au Lancer de Rayons, 7

# 1.1 Objectifs

L'objectif de ce projet est de vous faire écrire un algorithme de rendu "réaliste" appelé lancer de rayons (ray-tracing ou eye-tracing en anglais).

**Remarque :** Il existe de nombreuses descriptions de l'algorithme de lancer de rayon, il ne s'agit pas ici de faire un cours exhaustif, mais une introduction vous permettant de faire un algorithme simplifié. Pour ceux qui souhaiteraient aller plus loin, nous vous engageons à jeter un coup d'œil à l'excellente série de [cours](#) de Nicolas Bonneel

Le principe du ray-tracing s'appuie sur la perception visuelle de notre environnement : les rayons lumineux ont éclairé une scène observée, puis rebondi ou traversé certains objets pour enfin arriver dans notre œil et former une image sur notre rétine. Cependant, une méthode numérique utilisant cette approche directe se heuterait à une problème physique majeur : dans la réalité, la plupart des rayons lumineux ne sont pas allés sur notre rétine mais se sont dispersés ailleurs ! Ainsi, il faudrait que cette méthode envoie des milliards de rayons pour voir la scène se dessiner sur une image. Pour pallier cette difficulté, le ray-tracing adopte une approche inverse : la couleur de chaque pixel (un pixel étant vu comme un point sur la rétine) de l'image représentant la scène observée est calculée indépendamment en reconstruisant le trajet inverse de la lumière.

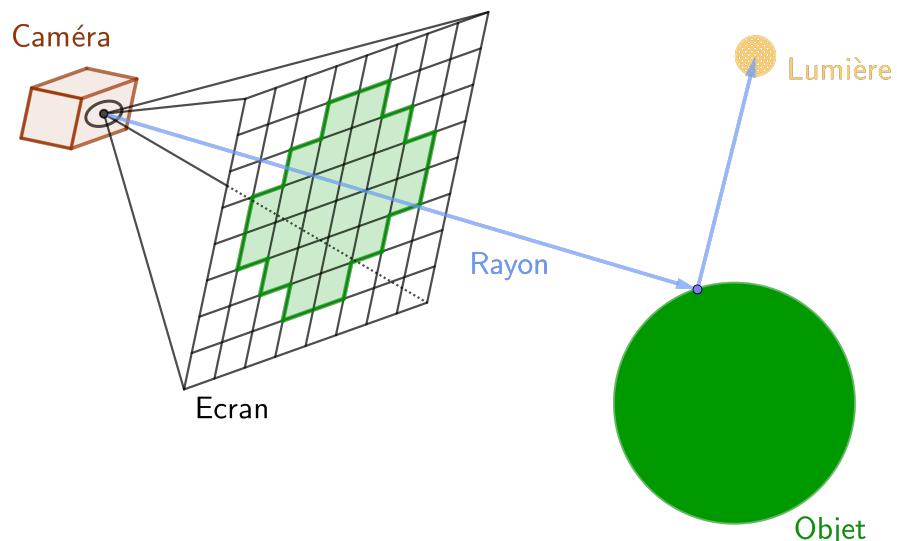


Fig. 1.1 – Illustration du principe de l'algorithme de lancer de Rayons

Autrement dit, le rayon est émis, non plus d'une source lumineuse, mais de l'œil, puis on calcule son intersection avec la scène observée :

- **diffusion** : si ce rayon intersecte un objet mat, la couleur correspondante est calculée en regardant où sont les sources de lumière puis le calcul s'arrête.
- **réflexion** : si ce rayon intersecte un objet réfléchissant, le rayon rebondit et le calcul continu jusqu'à tomber sur le cas d'arrêt.
- **réfraction** : si ce rayon intersecte un objet transparent, il le traverse et le calcul continue jusqu'à tomber sur le cas d'arrêt.

Cet algorithme permet d'obtenir des résultats intéressants avec des codes relativement simples. De plus, les nouveaux types d'objets (autres que cercles, plans ou triangles) ou de textures peuvent y être intégrer de façon très simple, sans changer la structure de l'algorithme. Cependant, la complexité du lancer de Rayons augmente directement en fonction du nombre d'objets et du nombre de sources lumineuses, et ce d'autant plus lorsque l'on rajoute de la transparence ou de la réflexion. Afin d'avoir

un code relativement rapide, on se limitera dans le cadre du projet à une seule source lumineuse. En outre, les réflexions/réfractions successives des rayons et leurs intersections avec les objets donnent lieu à des erreurs numériques. Pour corriger ces erreurs, il serait nécessaire d'envoyer plusieurs rayons par pixels à l'aide de méthodes d'échantillonnage non triviales (de type Montecarlo). mais nous ne traiterons pas cette approche dans ce projet.

A l'issue de ces séances, non seulement vous aurez codé en Python une méthode, que nous espérons intéressant, mais vous aurez aussi manipulé de la géométrie dans l'espace pour calculer les normales et les intersections des rayons avec différents objets 3D (cf module Traitement de modèle 1), ainsi que des notions physiques comme les différences entre des objets mats, brillants, transparents où les concepts de couleurs diffuses et spéculaires.

Pour conclure, l'objectif de ce projet est d'implémenter un algorithme assez complet de lancer de rayons comprenant plusieurs objets graphiques (sphères, plans) triangles avec plusieurs modèles de matériaux.

## 1.2 Travail attendu

Le travail attendu est mis en valeur de la manière suivante. Barème : **Si votre programme ne s'exécute pas, la note sera de 0.** **version basique 12pts maximum répartis comme suit :**

- rapport 2pts
- intersection sphère, plan et intersection scène 3pts
- modèles d'illumination 3pts
- ombres 2pts
- réflexions 2pt

**version avancée 12 points de la version basique auxquels s'ajoutent :**

- cylindre, triangle, cône ... 1pt par type d'objet
- modèles d'éclairage avancés (distribution de Beckmann, Terme de Fresnel, atténuation, refraction...) 4pts
- anti-aliasing 1pt
- texture 2pts
- faites nous rêver 2pts (par exemple avec une animation)

**Vous devrez impérativement respecter (sous peine de perdre des points) le formalisme de programmation imposé dans le sujet (nom des fonctions, variables d'entrées et de sorties...).** Vous devrez compter une vingtaine d'heures de travail pour ce projet, en dehors des séances de TPs. Vous devrez remettre un rapport avec votre projet, détaillant ce que vous avez implémenté, les problèmes rencontrés, et montrant les images que vous avez produites. Le code sera évalué en regard du rapport, par exemple, un programme très complet avec un rapport sibyllin sera évalué négativement. Si le rendu est en trop grand décalage avec les travaux effectués lors des TPs, l'équipe pédagogique pourra convoquer le binôme concerné pour un oral complémentaire. Enfin, vous rendrez impérativement (sous peine de perdre des points) le projet de votre binôme de TP dans un dossier compressé (zip ou tar.gz) au format suivant :

- le pdf portera le nom : NomPrenom1\_NomPrenom2\_GroupeTP.pdf
- le code (pas de fichier notebook avec l'extension .ipynb) portera le nom code\_raytracing\_NomPrenom1\_NomPrenom2

## 1.3 Prise en main du code

Afin de vous faciliter au mieux le travail et de vous concentrer sur la partie "graphique" du projet, un code basique permettant de fabriquer une image vous est fourni.

### 1.3.1 Ouverture/Modification et exécution d'un code Python

Tous les codes peuvent être exécutés via un Jupyter-Notebook (comme pour les TP précédents). Il existe, selon les salles où se déroulent vos TPs, d'autres moyens pour exécuter votre code.

➊ Si vous êtes dans les salles 3PN - SF1/SF2 : vous pouvez ouvrir/modifie votre code via éditeur de texte (p.e. Kate ou Nedit en tapant le nom correspondant dans un terminal) et l'executer dans un terminal. Pour cela, vous lancerez l'interpréteur Python3 en tapant python3 dans un terminal. Apparaît alors du texte ressemblant aux lignes suivantes :

```
Python 3.8.3 (default, Jul 2 2020, 11:26:31)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.16.1 -- An enhanced Interactive Python. Type '?' for help.
```

In [1]:

Pour exécuter le code contenu, par exemple, dans le fichier monSuperCode.py, il vous suffit alors de taper la commande run monSuperCode.py à la suite des deux points.



Attention, il ne faut pas utiliser Spyder car l'interpréteur Python associé est une version 2.7 qui provoquera des erreurs.

➋ Si vous êtes dans une autre salles que celles du 3PN - SF1/SF2 : vous pouvez directement utiliser Spyder pour ouvrir/modifier et exécuter votre code.

### 1.3.2 Crédation d'une image aléatoire

Voici un premier code qui permet de créer une image avec des couleurs aléatoires.



Un premier code permettant de générer une image :

```
1 import numpy as np
2 import random as rd
3 import matplotlib.pyplot as plt
4
5 # Taille de l'image
6 w = 800
7 h = 600
8
9 # image vide : que du noir
10 img = np.zeros((h, w, 3))
11 # Boucle sur l'ensemble des pixels
12 for i in range(w):
13     for j in range(h):
14         # la couleur est un tuple modélisant les trois canaux Red, Green, Blue
15         couleur = (rd.random(), rd.random(), rd.random())
16         img[h - j - 1, i] = couleur
17
18 plt.imsave('figInit.png', img)
```



## Analyse du code :

Les trois premières lignes permettent de charger des librairies standards que vous connaissez déjà. La librairie `matplotlib` (ligne 3) permet notamment d'avoir accès à toutes les méthodes de manipulation/sauvegarde d'images. La ligne 8 permet de créer une image, c'est-à-dire une matrice de `h` lignes, `w` colonnes et contenant des triplets RGB. Cette image est initialement remplie des zéros (du noir). Les deux boucles `for` imbriquées (lignes 11 et 12) permettent de parcourir l'image de lignes en lignes (la première ligne est remplie, puis on passe à la seconde...etc). La ligne 14 permet de créer un tuple de trois valeurs aléatoires, c'est ce qui représente la couleur qui est ensuite mise à la position (i.e. au pixel) `[h-j-1, i, :]` (les `:` signifie que les trois valeurs disponibles sont remplies d'un coup) de l'image `img`.

Un fois ce code exécuté, vous obtiendrez l'image suivante :

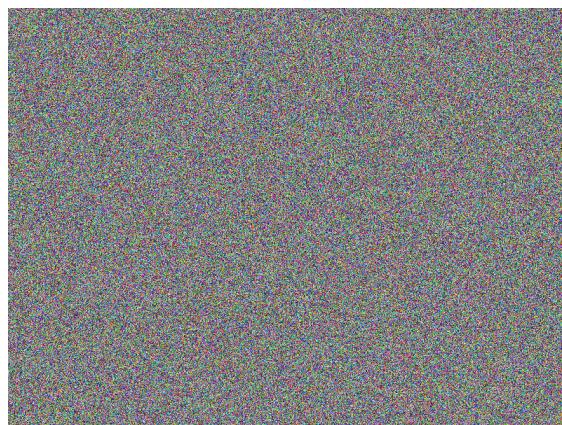


Fig. 1.2 – Une première image avec des couleurs aléatoires.



Le code qui vous est fourni implique certaines contraintes que vous devrait impérativement respecter :

- ▣ **L'origine du repère image est en haut à gauche**, c'est pour cela que l'on a `h-j-1` lorsqu'on remplit l'image dans les boucles `for`.
- ▣ Les valeurs de chacun des trois canaux RGB, c'est-à-dire les valeurs des tuples contenus dans les cases de la matrice `image`, doivent **impérativement être contenues dans l'intervalle [0, 1]**.

Les deux TP corrigés mis en ligne sur le moodle vous aideront à mieux appréhender la création et la manipulation des images à avec Python.

### 1.3.3 Vers un deuxième code s'adaptant au Lancer de Rayons

Il s'agit maintenant d'intégrer les différents éléments liés au principe de l'algorithme de Lancer de Rayons, à savoir **la caméra et l'écran** :

- ▶ **la caméra** est définie par une position dans l'espace (un vecteur), et une direction vers laquelle elle est dirigée (un autre vecteur).

- ▶ **l'écran** est défini par un tuple de quatre nombres représentant respectivement l'extrémité gauche, haute, droite et basse. Cependant, il faut que l'écran ait le même rapport d'aspect que l'image que nous voulons produire. Arbitrairement, les extrémités gauche et droite de l'écran sont -1 et 1. Afin d'ajuster l'écran à l'image il faut donc adapter les coins haut et bas. Ceci s'effectue à l'aide du ratio  $r$  défini comme le rapport entre la largeur  $w$  et la hauteur  $h$  de l'image. Ainsi l'extrémité basse vaut  $-\frac{1}{r}$  et l'extrémité haute  $\frac{1}{r}$ . En configurant l'écran de cette façon, on obtiendra un rapport d'aspect (largeur sur hauteur) :  $2/(2/r) = r$  qui est le rapport de l'image souhaitée (800x600).

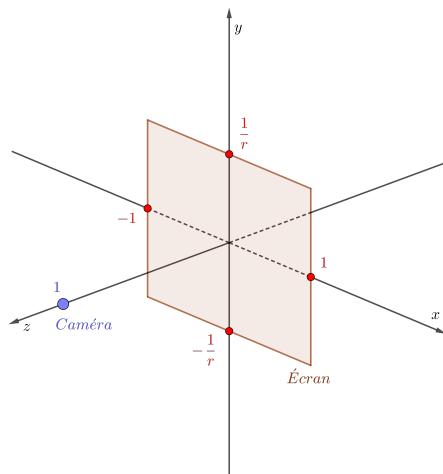


Fig. 1.3 – Taille de l'écran

Le code suivant intègre ces nouveaux éléments dans le code du paragraphe précédent.



#### Un premier code permettant de générer une image :

```

1 import numpy as np
2 import random as rd
3
4 import matplotlib.pyplot as plt
5 # Taille de l'image
6 w = 800
7 h = 600
8
9 img = np.zeros((h, w, 3)) # image vide : que du noir
10
11 C = np.array([0., 0.1, 1.1]) # Coordonée du centre de la camera.
12 Q = np.array([0,0.3,0]) # Orientation de la caméra
13 img = np.zeros((h, w, 3)) # image vide : que du noir
14
15 r = float(w) / h # rapport d'aspect
# coordonnées de l'écran : x0, y0, x1, y1.

```

```

17 S = (-1., -1. / r , 1., 1. / r )
18
19 # Boucle sur l'ensemble des pixels
20 for i, x in enumerate(np.linspace(S[0], S[2], w)):
21     for j, y in enumerate(np.linspace(S[1], S[3], h)):
22         couleur = (rd.random(), rd.random(), rd.random())
23         img[h - j - 1, i] = couleur
24
25 plt.imsave('figInit.png', img)

```



### Analyse du code :

Les lignes 11, 12, 15 et 17 définissent les nouveaux éléments. Les lignes 20 et 21 permettent de parcourir l'image en utilisant l'écran. L'instruction `np.linspace(...)` permet de diviser la largeur (respectivement hauteur) en `w` points (resp. `h`). La commande `enumerate` permet d'itérer sur les indices `i` (resp.`j`) et les valeurs `x` (resp. `y`) de cette subdivision de la largeur (resp. hauteur). Les indices `i` et `j` de ces subdivisions ont alors les même valeurs que les indices `i` et `j` des boucles du premier code (c'est normal, me direz-vous, c'est calculé pour...). Les variables `x` et `y` sont inutiles et ne sont pas utilisées.

L'image produit par ce deuxième code est la même que la précédente. Il faut maintenant s'attaquer à la partie la plus intéressante, la géométrie

Chapitre

# 2

# Intersections

## Sommaire

- 2.1 Structures de base, 10
- 2.1.1 Rayon, sphère et plan, 10
- 2.1.2 Fonctions additionnelles, 11
- 2.2 Calcul d'intersection, 11
- 2.2.1 Intersection rayon sphère, 11
- 2.2.2 Intersection rayon plan, 12
- 2.2.3 Tests unitaires, 12

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

## 2.1 Structures de base

### 2.1.1 Rayon, sphère et plan

➊ **Un rayon**, autrement dit une demi-droite, est constitué d'une origine  $\mathbf{O}$  (un point de  $\mathbb{R}^3$ ) et d'une direction  $\vec{\mathbf{d}}$  (un vecteur **normalisé** de  $\mathbb{R}^3$ ). Un rayon est donc défini paramétriquement par :

$$r_{\mathbf{O}, \vec{\mathbf{d}}} = \left\{ \mathbf{O} + t \vec{\mathbf{d}}, t \in [0, +\infty[ \right\} \quad (2.1)$$

Du point de vue de la programmation, nous considérerons qu'un rayon est un dictionnaire dont les clés sont son origine et sa direction. Afin d'utiliser un rayon dans le programme, vous devrez tout d'abord programmer une fonction `create_Ray` qui prend en entrée :

- un vecteur O représentant l'origine  $\mathbf{O}$ ,
- un vecteur D représentant la direction  $\vec{\mathbf{d}}$

et renvoie le rayon  $r_{\mathbf{O}, \vec{\mathbf{d}}}$ . Notez que O et D sont des `numpy.array`. Ensuite vous devrez programmer une fonction `rayAt` qui prend en entrée un rayon `ray` et un réel `t` et qui renvoie le point 3D du rayon repéré par le paramètre `t`.

➋ **Une sphère** est caractérisée par son centre  $\mathbf{C}$  et son rayon  $R$ . Les points appartenant à la sphère sont ceux qui sont à une distance  $R$  du centre, soit  $\|\mathbf{M} - \mathbf{C}\| = R$ . Une sphère est donc définie par son équation cartésienne avec la formule :

$$S_{\mathbf{C}, R} = \left\{ \mathbf{M} \in \mathbb{R}^3 \setminus \|\mathbf{M} - \mathbf{C}\| = R \right\} \quad (2.2)$$

Du point de vue de la programmation, nous considérerons, dans un premier temps qu'une sphère est un dictionnaire dont les clés sont :

- **son type** représenté par une chaîne de caractère
- **son centre  $\mathbf{C}$**  représenté par un `numpy.array`,
- **son rayon  $R$**  représenté par un réel,
- **son index  $i$**  (une entier) qui représente son étiquette (i.e. son numéro) dans la scene. Par exemple, l'objet 1 aura un index égal à 1, l'objet 2 un index égal à 2...

Ce modèle évoluera lorsque nous introduirons les paramètres de réflexion. Afin d'utiliser une sphère dans le programme, vous devrez tout d'abord programmer une fonction `create_Sphere` qui prend en entrée :

- un vecteur C représentant le centre  $\mathbf{C}$ ,
- un réel R représentant le rayon  $R$
- un index i

puis renvoie la sphère  $S_{\mathbf{C}, R}$ .

➌ **Un plan** est caractérisée par sa position  $\mathbf{P}$  (un point de  $\mathbb{R}^3$ ) dans l'espace et sa normale  $\vec{\mathbf{n}}$  (un vecteur **normalisé** de  $\mathbb{R}^3$ ). Un rayon est donc défini par son équation cartésienne avec la formule :

$$\Pi_{\mathbf{P}, \vec{\mathbf{n}}} = \left\{ \mathbf{M} \in \mathbb{R}^3 \setminus \langle (\mathbf{M} + \mathbf{P}) \cdot \vec{\mathbf{n}} \rangle = 0 \right\} \quad (2.3)$$

Du point de vue de la programmation, nous considérerons, dans un premier temps qu'un plan est un dictionnaire dont les clés sont :

- **son type** représenté par une chaîne de caractère

- sa position  $\mathbf{P}$  représenté par un `numpy.array`,
- son vecteur normal  $\vec{n}$  représenté par un `numpy.array`,
- son index  $i$  (une entier) qui représente son étiquette (i.e. son numéro) dans la scène.

Comme pour la sphère, ce modèle évoluera lorsque nous introduirons les paramètres de réflexion. Afin d'utiliser un plan dans le programme, vous devrez tout d'abord programmer une fonction `create_Plan` qui prend en entrée :

- un vecteur  $P$  représentant la position  $\mathbf{P}$ ,
- un vecteur  $n$  représentant la normale  $\vec{n}$
- un index  $i$

puis renvoie le plan  $\Pi_{\mathbf{P}, \vec{n}}$ .

## 2.1.2 Fonctions additionnelles

Outre les fonctions décrites ci-dessus, deux autres fonctions interviennent tout au long du programme. Il s'agit d'une méthode permettant de récupérer la normale d'un objet en point donné (pour l'instant les plans et les sphères) ainsi qu'une méthode permettant de normaliser un vecteur. Ainsi vous devrez, tout d'abord, programmer une fonction `get_Normal` qui prend en entrée :

- un objet  $obj$  (pour l'instant un plan ou une sphère)
- un point  $M$ , représenté par un `numpy.array`.

et qui renvoie le vecteur normal (un `numpy.array`) à l'objet  $obj$  au point  $M$ . Cette méthode évoluera avec les nouveaux objets que vous rajouterez (triangle, disque, cylindre...). Ensuite, vous programmerez une fonction `normalize` qui prend en entrée un vecteur  $x$  et renvoie le vecteur  $x$  normalisé.

## 2.2 Calcul d'intersection

Les scènes 3D basiques que vous aurez à programmer seront composées de deux types d'objets : des plans et des sphères. Il vous faudra donc écrire les fonctions capables de déterminer s'il y a intersection ou non avec un rayon, et si c'est le cas, calculer la position du point d'intersection.

### 2.2.1 Intersection rayon sphère

Comme nous l'avons expliqué, un rayon est caractérisé par son origine  $\mathbf{O}$  et sa direction  $\vec{d}$ . Une sphère est, quant à elle, caractérisée par son centre  $\mathbf{C}$  et son rayon  $R$ . Suivant les équations (2.1) et (2.2), il y a intersection s'il y a au moins un point du rayon qui se trouve sur la sphère, c'est à dire qui satisfait les deux équations :

$$\begin{cases} \mathbf{M} = \mathbf{O} + t \vec{d} \\ \|\mathbf{M} - \mathbf{C}\| = R \end{cases} \quad (2.4)$$

En remplaçant l'expression de  $\mathbf{M}$  dans la deuxième équation, on obtient :

$$\|(\mathbf{O} + t \vec{d}) - \mathbf{C}\| = R \quad (2.5)$$

En élevant au carré l'égalité précédente et en utilisant le fait que  $\|u\|^2 = \langle u, u \rangle \quad \forall u \in \mathbb{R}^3$ , vous résoudrez l'équation (2.5) et programmerez une fonction `intersect_Sphere` qui prend en entrée :

- un **dictionnaire**  $ray$  représentant le rayon  $r_{\mathbf{O}, \vec{d}}$

- un **dictionnaire** `sphere` représentant la sphère  $S_{C,R}$

et renvoie :

- le paramètre  $t$  tel que  $\mathbf{O} + t \vec{\mathbf{d}} \in S_{C,R}$ , s'il y a une intersection.
- $+\infty$  sinon



Nous porterons une attention particulière aux éléments suivants :

- ⌚ les variables d'entrées `ray` et `sphere` doivent impérativement être des dictionnaires.
- ⌚ la gestion des différents cas (i.e. selon les valeurs du discriminant) lors de la résolution de l'équation (2.5) ainsi que leur interprétations.

## 2.2.2 Intersection rayon plan

Un rayon est caractérisé par son origine  $\mathbf{O}$  et sa direction  $\vec{\mathbf{d}}$ . Une plan est, quant à lui, caractérisé par sa position  $\mathbf{P}$  et sa normale  $\vec{\mathbf{n}}$ . Suivant les équations (2.1) et (2.3), il y a intersection s'il y a au moins un point du rayon qui se trouve sur la plan, c'est à dire qui satisfait les deux équations :

$$\begin{cases} \mathbf{M} = \mathbf{O} + t \vec{\mathbf{d}} \\ \langle (\mathbf{M} - \mathbf{P}) . \vec{\mathbf{n}} \rangle = 0 \end{cases} \quad (2.6)$$

En remplaçant l'expression de  $\mathbf{M}$  dans la deuxième équation, on obtient :

$$\langle (\mathbf{O} + t \vec{\mathbf{d}} - \mathbf{P}) . \vec{\mathbf{n}} \rangle = 0 \quad (2.7)$$

En résolvant l'équation précédente, vous programmerez une fonction `intersect_Plan` qui prend comme paramètre :

- un **dictionnaire** `ray` représentant le rayon  $r_{\mathbf{O}, \vec{\mathbf{d}}}$
- un **dictionnaire** `plane` représentant le plan  $\Pi_{\mathbf{P}, \vec{\mathbf{n}}}$

et renvoie :

- le réel  $t$  tel que  $\mathbf{O} + t \vec{\mathbf{d}} \in \Pi_{\mathbf{P}, \vec{\mathbf{n}}}$ , s'il y a une intersection,
- $+\infty$  sinon



Nous porterons une attention particulière aux éléments suivants :

- ⌚ les variables d'entrées `ray` et `plane` doivent impérativement être des dictionnaires.
- ⌚ la gestion des différents cas possibles lors de la résolution de l'équation (2.7).

## 2.2.3 Tests unitaires

Les fonctions précédentes sont validées par des tests unitaires. Ces tests sont définis dans le fichier dans le code `raytracingTestsUnitaires.py`. Pour vérifier votre implémentation, vous remplissez les endroits indiqués avec votre code. Si votre implémentation est correcte, vous devriez obtenir l'affichage suivant lors de l'exécution :

```
Début des tests unitaires :  
r1 at t1 : [OK]  
r1 at t2 : [OK]  
r1 at t3 : [OK]  
r2 at t4 : [OK]  
r2 at t5 : [OK]  
r2 at t6 : [OK]  
r3 at t7 : [OK]  
r3 at t8 : [OK]  
r3 at t9 : [OK]  
--> tests de la fonction traceRay : OK  
r1 to plane1 : [OK]  
r1 to plane2 : [OK]  
r1 to plane3 : [OK]  
r2 to plane1 : [OK]  
r2 to plane2 : [OK]  
r2 to plane3 : [OK]  
r3 to plane1 : [OK]  
r3 to plane2 : [OK]  
r3 to plane3 : [OK]  
--> tests de la fonction intersection_Plane : OK  
r1 to sphere1 : [OK]  
r1 to sphere2 : [OK]  
r1 to sphere3 : [OK]  
r2 to sphere1 : [OK]  
r2 to sphere2 : [OK]  
r2 to sphere3 : [OK]  
r3 to sphere1 : [OK]  
r3 to sphere2 : [OK]  
r3 to sphere3 : [OK]  
--> tests de la fonction intersection_Sphere : OK
```

## Chapitre

# 3

# Vers une première image

## Sommaire

3.1	Un premier rendu minimal, 15
3.1.1	Rajout de la couleur “ambiante”, 15
3.1.2	Intersection entre un rayon et un objet de la scène, 15
3.1.3	Un premier calcul de la couleur d'un pixel, 15
3.1.4	Une première image, 16
3.2	Modèle d'éclairage et Ombres, 16
3.2.1	Rajout de la couleur “diffuse”, 17
3.2.2	Un premier modèle d'illumination, 17
3.2.3	Modèle d'illumination de Blinn-Phong, 18
3.2.4	Ombres, 20
3.2.5	Réflexions, 22
3.2.6	Comment tester votre code, 25

# 3.1 Un premier rendu minimal

Dans cette section, nous allons générer une image “primitive”, sans modèle d’illumination, prise en compte de l’ombre et des reflexions. Le but est uniquement de mettre en place et de valider les premiers éléments de l’algorithme général.

## 3.1.1 Rajout de la couleur “ambiante”

Comme nous l’avons décrit au début du document, il faut affecter à chaque intersection entre un rayon et une objet (pour l’instant sphère ou plan) une couleur. Il n’est pas question dans cette section de modèle d’illumination, la couleur de l’intersection sera affecté à la couleur **ambiante** de l’objet. Cette couleur (qui pourrait s’apparenter à une sorte de couleur d’émission) représente la couleur de l’objet qui interagit avec la lumière ambiante. Notez qu’en toute rigueur, il faudrait parler de la couleur ambiante du matériau dont est fait l’objet et non de celle de l’objet, cependant nous n’introduiront pas distinction entre les deux dans ce projet. Afin de pouvoir utiliser cette couleur dans les calculs, il faut définir une nouvelle clé `ambient` dans les objets `plan` et `sphère`. La valeur associée à cette clé est une couleur, c'est-à-dire un vecteur de  $\mathbb{R}^3$  (i.e. un `numpy.array`) dont les composantes sont comprise entre 0 et 1. Vous commencerez donc par modifier les fonctions `create_sphère` et `create_plan` afin d’intégrer cette nouvelle clé.

## 3.1.2 Intersection entre un rayon et un objet de la scène

Pour obtenir un premier rendu, il faut ensuite implémenter une première fonction `intersect_Scene` dont le but est de vérifier s’il y a une intersection entre un rayon et chaque objet de la scène. Ainsi, la fonction `intersect_Scene` doit prendre en entrée :

- un **dictionnaire** `ray` représentant un rayon
- un **dictionnaire** `obj` représentant l’objet de la scène considéré.

et renvoyer, si elle existe, l’intersection entre le rayon `ray` et l’objet `obj` et  $+\infty$  sinon. Notez que, pour l’instant, les seuls objets considérées sont les plans et les sphères. Cependant vous pourrez faire évoluer cette fonction selon les objets que vous rajouterez (triangle, disque, cylindre...).

## 3.1.3 Un premier calcul de la couleur d’un pixel

Une fois la fonction `intersect_Scene` programmée, vous devrez implémenter la fonction `trace_ray`. Cette fonction est au coeur du programme : c’est elle qui est chargée de calculer la couleur du pixel associée à un rayon. Vous commencerez par programmer une version simplifiée de `trace_ray`, puis vous la ferez évoluer lorsque vous améliorerez l’éclairage avec le rajout du modèle d’illumination et de la prise en compte de l’ombre. Pratiquement, cette fonction doit prendre en entrée un dictionnaire `ray` représentant le rayon et renvoyer :

- un **dictionnaire** `obj` représentant le **premier** objet de la scène intersecté par le rayon `ray`,
- un `numpy.array` `P` représentant le point d’intersection entre `obj` et `ray`,
- un `numpy.array` `N` représentant la normale à l’objet `obj` au point `P`,
- un `numpy.array` `col_ray`  $\in [0, 1]^3$  représentant la couleur du pixel correspondant au rayon `ray`.

Afin de programmer le corps de cette fonction, vous utiliserez le pseudo-code suivant :

---

```

1: pour tout objet obj_test de la scène faire
2:   calculer l'intersection entre ray et obj_test
3:   si cette distance est plus courte que la précédente alors
4:     la distance courante devient la distance de première intersection.
5:     l'objet obj_test courant devient le premier objet intersecté obj.
6:   si le rayon n'a pas intersecté d'objets de la scène alors
7:     retour None
8:   sinon
9:     on calcule P, l'intersection entre obj et ray
10:    on calcule N, la normale à obj en P
11:    on calcule col_ray, qui vaut pour l'instant la couleur de l'objet (la valeur de la clé ambient)
12:    retour obj, P, N et col_ray

```

---



**Indication :** La fonction trace\_ray utilise les fonctions intersect\_Scene, rayAt et get\_Normal

### 3.1.4 Une première image

La dernière étape de cette section consiste à mettre en place la boucle principale. Celle-ci s'écrit, sous forme de pseudo-code, de la manière suivante : Comme dans le code raytracing2.py,

---

```

1: pour tout pixel p de l'écran faire
2:   associer la couleur noire à p
3:   construire le rayon raytest qui part de la caméra et va vers p
4:   si le rayon raytest intersecte un objet de la scène alors
5:     récupérer la couleur du point d'intersection et l'ajouter à celle de p
6:   associer la couleur p au pixel correspondant dans l'image img
7: sauvegarder img

```

---

l'écran est subdivisé dans les directions  $x$  et  $y$ . Ainsi, la boucle qui parcourt les pixels de l'écran est, en réalité, décrite à l'aide de deux boucles imbriquées qui décrivent les indices et les coordonnées des pixels de l'écran dans les deux direction  $x$  et  $y$ .

```

1  for i, x in enumerate(np.linspace(S[0], S[2], w)):
2      for j, y in enumerate(np.linspace(S[1], S[3], h)):

```

Afin de suivre la progression des calculs, on introduit, entre les deux boucles, quelques lignes qui permettent d'afficher le pourcentage d'avancement du code :

```

1  for i, x in enumerate(np.linspace(S[0], S[2], w)):
2      if i % 10 == 0:
3          print(i / float(w) * 100, "%")
4      for j, y in enumerate(np.linspace(S[1], S[3], h)):

```

Si votre programme fonctionne correctement, vous devriez obtenir l'image suivante :

## 3.2 Modèle d'éclairage et Ombres

Evidemment, le rendu précédent est très grossier. Afin d'améliorer ce rendu, nous allons prendre en compte les sources de lumières, car on sait qu'elles vont changer la couleur perçue des objets ainsi



Fig. 3.1 – Une première image dite "primitive"

que le calcul de l'ombre qui en découle.

### 3.2.1 Rajout de la couleur "diffuse"

L'amélioration du modèle d'illumination passe par l'introduction d'une nouvelle couleur dite **diffuse**. Contrairement à la couleur ambiante précédemment utilisée, l'interprétation de la couleur **diffuse** est plus instinctive : il s'agit essentiellement de la couleur que l'objet révèle sous une lumière blanche pure. Elle est donc perçue comme la couleur de l'objet lui-même plutôt que comme un reflet de la lumière. Afin de pouvoir utiliser cette couleur dans les calculs, il faut définir une nouvelle clé `diffuse` dans les objets `plan` et `sphere`. La valeur associée à cette clé est une couleur, c'est-à-dire un vecteur de  $\mathbb{R}^3$  (i.e. un `numpy.array`) dont les composantes sont comprise entre 0 et 1. Vous commencerez donc par modifier les fonctions `create_sphere` et `create_plan` afin d'intégrer cette nouvelle clé. Ensuite vous définirez la source lumineuse de notre programme appelée `Light` comme un dictionnaire dont les clés sont :

- **sa position** représentée par `numpy.array`,
- **sa couleur ambiante** représenté par un `numpy.array`,
- **sa couleur diffuse** représenté par un `numpy.array`,

Vous positionnerez cette source lumineuse en  $(5, 5, 0)^\top$  avec pour couleur ambiante  $(\frac{5}{100}, \frac{5}{100}, \frac{5}{100})^\top$  et une couleur diffuse **blanche**.

### 3.2.2 Un premier modèle d'illumination

Les surfaces mates (neige, papier) présentent une couleur diffuse (on parle aussi de réflexion Lamber-tienne) et apparaissent brillante quelle que soit la direction d'observation. Comme nous allons le voir, ceci est dû au fait, qu'avec ce modèle, la couleur en un point de la surface **O** dépend uniquement de l'angle  $\theta$  entre la normale  $\vec{n}$  à la surface **O** et la direction du point à la source lumineuse **L**. En pratique, la couleur diffuse, notée  $c_d$  est calculée au point d'intersection **P** entre le rayon et l'objet considéré à l'aide de la formule suivante :

$$c_d = K_a \times l_a + K_d \times l_d \times \langle \vec{l}, \vec{n} \rangle \quad (3.1)$$

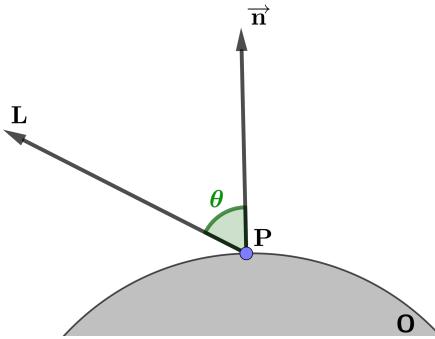


Fig. 3.2 – Modèle d'illumination lambertien simplifié

où :

- $\vec{I}$  est la direction d'éclairage du point  $\mathbf{P}$  éclairé par une lumière située en  $\mathbf{L}$ . Cette direction est définie comme le vecteur (normalisé) orienté du point  $\mathbf{P}$  vers la lumière  $\mathbf{L}$  :  $\vec{I} = \frac{\mathbf{L} - \mathbf{P}}{\|\mathbf{L} - \mathbf{P}\|}$ ,
- $K_a$  est la couleur ambiante de l'objet,
- $l_a$  la couleur ambiante de la lumière.
- $K_d$  est la couleur diffuse de l'objet,
- $l_d$  la couleur diffuse de la lumière.

A l'aide de la formule (3.1), vous programmerez la fonction `eclairage` dont le but est de calculer la couleur diffuse au point d'intersection entre un rayon et un objet. Vous commencerez par programmer une version simplifiée de `eclairage` qui évoluera avec le modèle d'éclairage. Pratiquement, cette fonction doit prendre en entrée :

- un **dictionnaire** `obj` représentant l'objet de la scène considéré,
- un **dictionnaire** `light` représentant la source lumineuse considérée,
- un `numpy.array` `P` représentant le point  $\mathbf{P} \in \text{obj}$  auquel on veut calculer la couleur diffuse.

et renvoie un `numpy.array` représentant la couleur diffuse au point  $\mathbf{P}$ .



Comme vous avez tout suivi dans les moindres détails, vous aurez noté que  $\vec{I}$  et  $\vec{n}$  sont des vecteurs normalisés. Ainsi, vous aurez remarqué que la couleur  $\mathbf{c}_d$  dépend directement du cosinus de l'angle  $\theta$  entre  $\vec{I}$  et  $\vec{n}$ . Cependant, le cosinus peut être négatif si  $\vec{I}$  et  $\vec{n}$  sont opposés : cela correspond au cas où la lumière  $\mathbf{L}$  se trouve derrière l'objet considéré. Dans ce cas, vous prendrez soin de renvoyer simplement du noir et non pas une couleur négative !

Lorsque vous aurez programmé cette fonction, vous l'intégrerez à la fonction `trace_ray`. Concrètement, vous remplacerez la couleur ambiante de l'objet (ligne 11 dans le pseudo-code fourni dans la sous-section 3.1.3) par l'appel à `eclairage`. Si votre programme fonctionne correctement, vous devriez obtenir l'image suivante :

### 3.2.3 Modèle d'illumination de Blinn-Phong

Quand on observe une scène éclairée par une forte source de lumière, on s'aperçoit qu'il y a des taches très brillantes qui apparaissent. Ces taches correspondent à la réflexion de la source de lumière sur l'objet, c'est ce que l'on appelle la couleur dite **spéculaire**. La spécularité est un effet causé par des imperfections microscopiques à la surface d'un objet. Cela se manifeste comme une

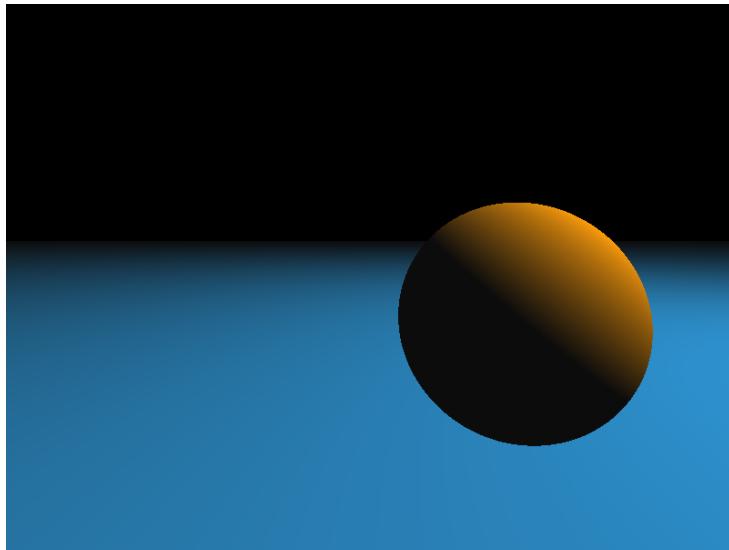


Fig. 3.3 – Une image avec un modèle d'illumination Lambertien simplifié

tâche lumineuse à sa surface. Afin de pouvoir utiliser cette couleur **spéculaire** dans les calculs, il faut définir une nouvelle clé `speculaire` dans les objets `plan` et `sphère`. La valeur associée à cette clé est une couleur, c'est-à-dire un vecteur de  $\mathbb{R}^3$  (i.e. un `numpy.array`) dont les composantes sont comprise entre 0 et 1. Vous commencerez donc par modifier les fonctions `create_sphère` et `create_plan` afin d'intégrer cette nouvelle clé. Ensuite vous ajouterez à la source lumineuse une nouvelle clé `speculaire` que vous affecterez à la couleur blanche.

Ce modèle introduit deux nouveaux vecteurs : le vecteur de réflexion  $\mathbf{R}$ , symétrique de  $\mathbf{L}$  par rapport à  $\vec{\mathbf{n}}$ , et le vecteur  $\mathbf{C}$  représentant la position de la caméra (cf Figure 3.4). Le modèle d'illumination de Blinn-Phong ajoute cette composante spéculaire modèle précédent.

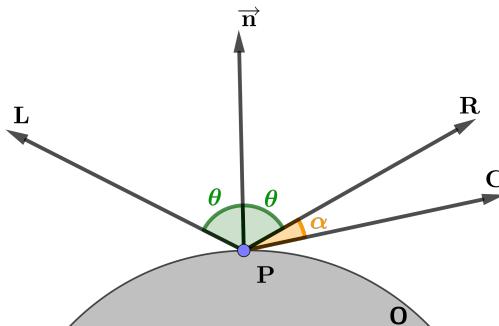


Fig. 3.4 – Exemple du modèle d'illumination complété de Blinn-Phong

Le modèle repose sur l'idée que la réflexion est importante lorsque l'angle  $\alpha$  entre la direction d'observation  $\mathbf{C}$  du point de la surface et la direction de réflexion  $\mathbf{R}$  est faible. En rajoutant le terme spéculaire, la couleur totale, notée  $\mathbf{c}_t$  est calculée au point d'intersection  $\mathbf{P}$  entre le rayon et l'objet considéré à l'aide de la formule suivante :

$$\mathbf{c}_t = \mathbf{c}_d + K_s \times l_s \times \left\langle \frac{\vec{\mathbf{l}} + \vec{\mathbf{c}}}{\|\vec{\mathbf{l}} + \vec{\mathbf{c}}\|}, \vec{\mathbf{n}} \right\rangle^{\frac{\beta}{4}} \quad (3.2)$$

où :

- $\vec{\mathbf{c}}$  le vecteur (normalisé) orienté du point  $\mathbf{P}$  vers la caméra  $\mathbf{C}$  :  $\vec{\mathbf{c}} = \frac{\mathbf{C} - \mathbf{P}}{\|\mathbf{C} - \mathbf{P}\|}$ ,

- $K_s$  est la couleur spéculaire de l'objet,
- $I_s$  la couleur spéculaire de la lumière.
- $\beta$  est la brillance de l'objet que vous définirez comme variable `materialShininess` du programme principal. Pour les obtenir les images suivantes, `materialShininess` est fixé à 50.

**💀** La nouvelle composante ajoutée à couleur  $c_d$  ne dépend que du cosinus  $\vec{I} \cdot \vec{C}$  et  $\vec{n}$ . Cependant, ce cosinus peut être négatif. Dans ce cas, vous prendrez donc soin de renvoyer du noir et non pas une couleur négative !

Modifiez la fonction `eclairage` afin de prendre en compte ce nouveau modèle. Si tout fonctionne correctement, vous devriez obtenir l'image suivante :

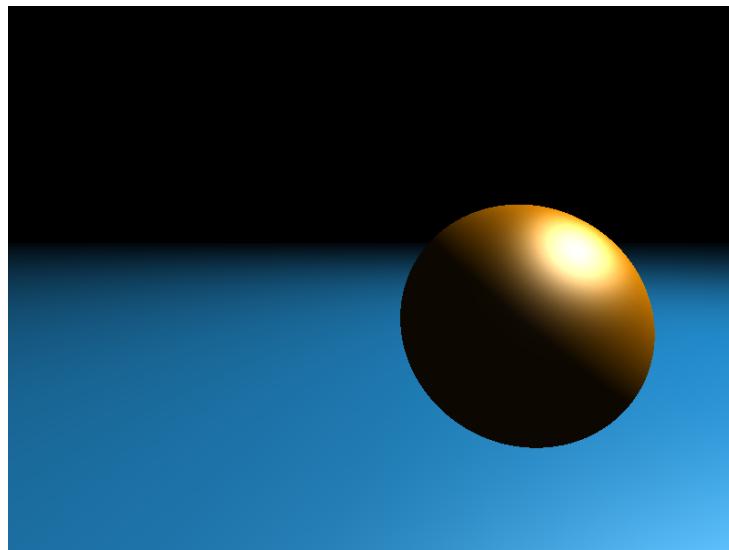


Fig. 3.5 – Une image avec un modèle d'illumination de Blinn-Phong

### 3.2.4 Ombres

Actuellement, pour calculer le rendu en un point  $\mathbf{P}$  d'un objet, on regarde la de lumière et on additionne les couleurs ambientes, diffuses et spéculaires correspondantes. Or, si un autre objet se situe entre  $\mathbf{P}$  et la source de lumière, il devrait y avoir occultation. Autrement dit, le point  $\mathbf{P}$  est dans l'ombre par rapport à la source de lumière (cf Figure 3.6). Dans ce paragraphe, nous allons donc rajouter la prise en compte des ombres portées sur objet générées par l'occultation de la lumière par un autre objet. Pour tenir compte de ces ombres, on teste l'éclairage d'un point en envoyant un rayon partant du point considéré vers la source lumineuse. S'il n'y a pas d'intersection entre le point et la lumière, alors le point est éclairé, sinon il est dans l'ombre.



#### Phénomène d'acné :

Si on ne prend pas de précautions pour initialiser le point de départ du rayon, les imprécisions de calcul numérique peuvent provoquer un phénomène d'**acné** (sorte de petites taches noires à la surface des objets). Pour éviter ce problème, il faut décaler le point de départ dans la direction de la lumière pour que la surface de départ ne soit pas en intersection avec le rayon lancé en direction de la lumière

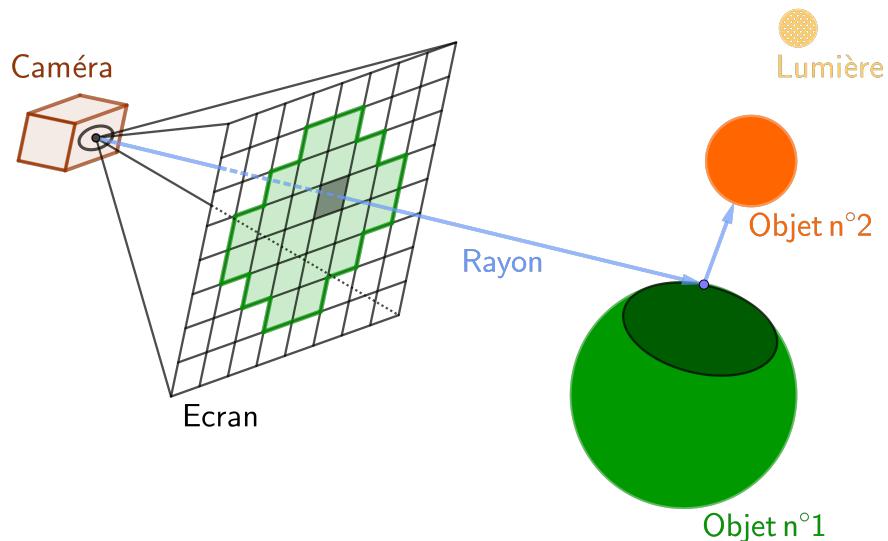


Fig. 3.6 – Illustration de la prise en compte de l'ombre

Maintenant que vous avez bien compris le phénomène géométrique lié aux ombres, vous devez programmer la fonction `Is_in_Shadow`. Le but de cette fonction est de déterminer si le point d'intersection  $\mathbf{P}$  entre l'objet  $\mathbf{O}$  et le rayon  $r$  est dans l'ombre ou non. En pratique, cette fonction doit prendre en entrée :

- un **dictionnaire** `obj` représentant l'objet  $\mathbf{O}$
- un `numpy.array` `P` représentant le point d'intersection entre l'objet `obj` et le rayon  $r$ ,
- un `numpy.array` `N` représentant la normale à l'objet `obj` au point `P`,

et renvoie `True` si l'objet est dans l'ombre, `False` sinon. Afin de programmer le corps de cette fonction, vous construirez une liste contenant tous les objets de la scène autres que `obj` se trouvant entre `obj` et la source lumineuse. Si cette liste est vide, le point n'est pas dans l'ombre, si elle n'est pas vide, c'est que l'objet s'y trouve. Ceci peut se mettre en œuvre à l'aide du pseudo-code suivant :

---

```

1: acne_eps ← 1e-4
2: L ← liste vide
3: créer  $\mathbf{l}_{PL}$ , une direction (normalisée) de  $\mathbf{P}$  vers la source lumineuse
4: créer un point  $\mathbf{P}_E$  en décalant  $\mathbf{P}$  de acne_eps dans la direction de  $\mathbf{N}$ 
5: créer un rayon  $\text{rayTest}$ , d'origine  $\mathbf{P}_E$  et de direction  $\mathbf{l}_{PL}$ 
6: pour tout objet obj_test de la scène faire
7:   si l'index de obj_test est différent de celui de obj alors
8:     calculer l'intersection entre rayTest et obj_test
9:     si il y a une intersection alors
10:      rajouter obj_test à L
11: si la liste est vide alors
12:   retour True
13: sinon
14:   retour False

```

---

Vous appellerez enfin `Is_in_Shadow` dans la fonction `trace_Ray`, avant d'appliquer l'éclairage. Cet appel se fait à l'aide de la condition positionnée aux lignes du pseudo-code de la fonction `trace_Ray` : Si tout fonctionne correctement, vous devriez obtenir l'image suivante :

---

```

1: pour tout objet obj_test de la scène faire
2:   calculer l'intersection entre ray et obj_test
3:   si cette distance est plus courte que la précédente alors
4:     la distance courante devient la distance de première intersection.
5:     l'objet obj_test courant devient le premier objet intersecté obj.
6:   si le rayon n'a pas intersecté d'objets de la scène alors
7:     retour None
8:   sinon
9:     on calcule P, l'intersection entre obj et ray
10:    on calcule N, la normale à obj en P
11:    shadow ← Is_in_Shadow(obj, P, N)
12:    si shadow alors
13:      retour None
14:    calculer col_ray à l'aide du modèle d'illumination de Blinn-Phong
15:  retour obj, P, N et col_ray

```

---

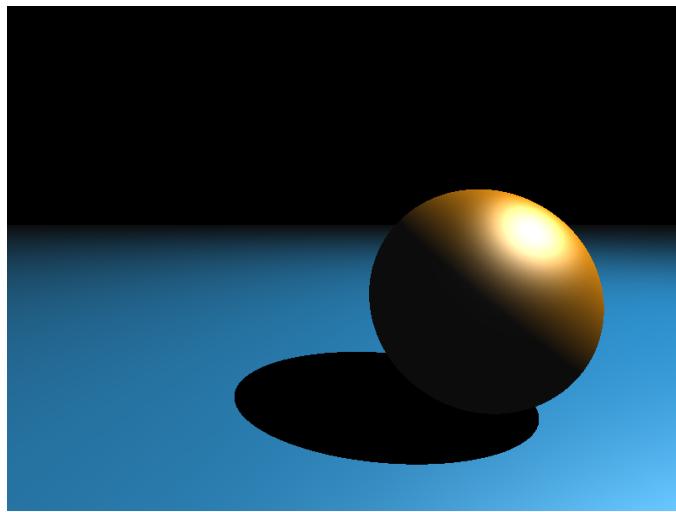


Fig. 3.7 – Une image avec un modèle d'illumination de Blinn-Phong et la prise en compte des ombres

### 3.2.5 Réflexions

Dans la version actuelle de notre code, les couleurs des pixels sont calculées uniquement à partir des contributions dues aux rayons partant de la caméra, touchant la surface d'un objet, et rebondissant directement vers la source lumineuse. Mais, que se passe-t-il si certains de ces rayons touchent plusieurs objets avant de toucher la caméra (cf Figure 3.8)? C'est le phénomène de **réflexion**. Dans la réalité, les couleurs proviennent toujours de la réflexion des rayons lumineux ou de leur transmission par transparence. Les notions de couleur diffuse et spéculaire que nous avons introduites via le modèle de Blinn-Phong permettent des approximations bien commodes pour un rendu rapide, mais elles ne suffisent pas à restituer toute la complexité liée aux milliards de rayons lumineux touchant le point observé. Dans ce projet, nous allons beaucoup simplifier le phénomène de réflexion en ne calculant qu'un rayon réfléchi et en ne le réfléchissant qu'une nombre fini de fois. Le rayon ainsi considéré accumulera différentes couleurs, faisant apparaître des reflets.

Afin de pouvoir programmer le phénomène de réflexion, nous aurons tout d'abord besoin d'introduire, pour chaque objet de la scène, un indice de réflexion qui représente la capacité de l'objet à refléter les rayons lumineux. Pour cela, il faut définir une nouvelle clé `reflection` dans les objets `plan` et `sphère`. La valeur associée à cette clé est un réel  $c_r \in [0, 1]$  représentant le coefficient de réflexion de l'objet. Vous commencerez donc par modifier les fonctions `create_sphere` et `create_plan`

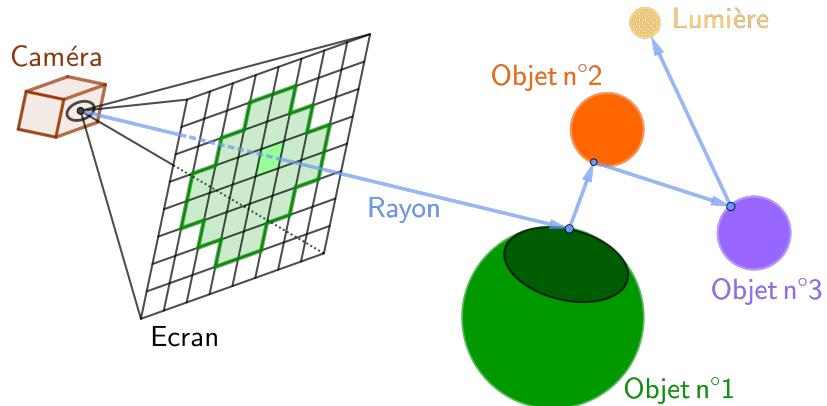


Fig. 3.8 – Illustration du phénomène de réflexion.

afin d'intégrer cette nouvelle clé.

Nous aurons ensuite besoin d'une fonction qui, à partir d'un rayon  $r_{\mathbf{O}, \vec{d}}$  et de la normale  $\vec{n}$  à l'objet qu'il intersecte, calcule la direction  $\vec{d}'$  du rayon réfléchi  $\tilde{r}_{\mathbf{P}, \vec{d}'}$  (voir Figure 3.9).

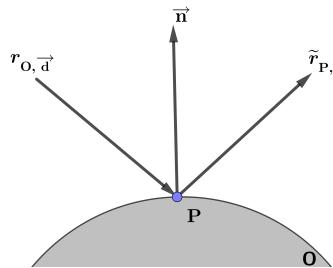


Fig. 3.9 – Illustration de la reflexion d'un rayon

La direction  $\vec{d}'$  du rayon réfléchi se calcule à l'aide de la formule :

$$\vec{d}' = \vec{d} - 2 \langle \vec{d}, \vec{n} \rangle \vec{n} \quad (3.3)$$

Ainsi, vous devez programmer la fonction `reflected_ray` qui prend en entrée :

- un `numpy.array` `dirRay` représentant la direction  $\vec{d}$  du rayon  $r_{\mathbf{O}, \vec{d}}$
- un `numpy.array` `N` représentant la normale  $\vec{n}$  à l'objet qu'intersecte le rayon  $r_{\mathbf{O}, \vec{d}}$ .

et renvoie un `numpy.array` représentant le vecteur  $\vec{d}'$ .

Maintenant que vous avez bien compris le phénomène géométrique lié aux ombres, vous devez programmer la fonction `Is_in_Shadow`. Le but de cette fonction est de déterminer si le point d'intersection  $\mathbf{P}$  entre l'objet  $\mathbf{O}$  et le rayon  $r$  est dans l'ombre ou non. En pratique, cette fonction doit prendre en entrée :

Pour terminer, vous devez programmer la fonction `compute_reflection` qui calcule la couleur au point d'intersection en ajoutant la contribution provenant du rayon réfléchi en ce point. Cette fonction suit le schéma suivant : après avoir calculé l'éclairage dû à la source de lumière, elle détermine le rayon réfléchi, puis calcule la contribution du rayon réfléchi en appelant `trace_ray` et mélange cette contribution avec la couleur obtenue par l'éclairage direct. **Attention**, pour ne pas avoir une infinité de rebonds, `compute_reflection` utilise un compteur qui s'incrémenté à chaque nouveau rebond et stoppe le calcul lorsque la limite autorisée `depth_max` est dépassée. En pratique, la fonction `compute_reflection` fonction doit prendre en entrée :

- un **dictionnaire** `rayTest` représentant le rayon associé au pixel dont on veut calculer la couleur
- un **scalaire** `depth_max` représentant le nombre maximum de réflexions,
- un **numpy.array** `col`  $\in [0, 1]^3$  représentant la couleur du pixel correspondant au rayon `rayTest` à laquelle on veut rajouter la contribution des réflexions.

et renvoie la couleur `col` à laquelle ont été rajoutées les contributions des réflexions. Afin de programmer le corps de la fonction `compute_reflection`, vous pourrez vous appuyer sur le pseudo-code suivant :

---

```

1:  $\vec{d} \leftarrow$  la direction de rayTest
2: initialiser le coefficient de réflexion c à 1
3: pour tout k dans la séquence {1, depth_max} faire
4:   calculer la première intersection de rayTest avec la scène.
5:   si pas d'intersection avec la scène alors
6:     on sort de la boucle
7:   récupérer obj, le premier objet de la scène rencontré par rayTest
8:   récupérer M, l'intersection de rayTest avec obj
9:   récupérer N, la normale à obj en M.
10:  récupérer col_ray, la couleur de la première intersection de rayTest avec la scène
11:  ajouter c  $\times$  col_ray à col
12:  créer un point M_E en décalant M de acne_eps dans la direction de N
13:  mettre à jour la direction  $\vec{d} \leftarrow$  la direction du rayon réfléchi issu de  $r_{M_E}, \vec{d}$ 
14:  mettre à jour rayTest  $\leftarrow$  le rayon réfléchi  $r_{M_E}, \vec{d}$ 
15:  mettre à jour le coefficient de réflexion c  $\leftarrow c  $\times$  le coefficient de réflexion c_r de obj
16: retour col$ 
```

---



Vous devrez impérativement utiliser la fonction `reflected_ray` dans `compute_reflection`

Si tout fonctionne correctement, vous devriez obtenir l'image suivante :

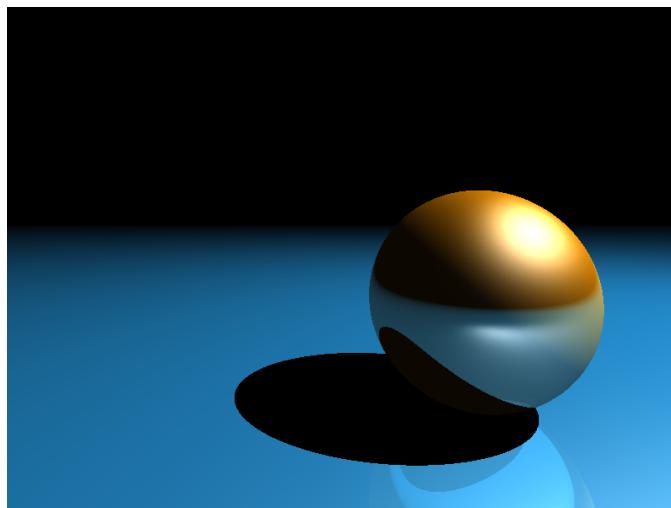


Fig. 3.10 – Une image avec un modèle d'illumination de Blinn-Phong, la prise en compte des ombres et des réflexions.

### 3.2.6 Comment tester votre code

Un programme appelé `raytracingTestScene.py` vous est fourni dans l'espace moodle dédié au projet. Ce programme contient l'ensemble des signatures des fonctions que vous devez coder : il vous reste donc à programmer le corps de celles-ci. Il contient aussi l'ensemble des éléments géométriques (définition de la scène, position et couleur de la lumière, position de la caméra...) et des constantes (`materialShininess`, `acne_eps...`) permettant de calculer les images présentées dans ce document. Vous y trouverez enfin la boucle principale dans laquelle vous devrez insérer votre code faisant appels aux différentes fonctions.



#### Attention :

Le scene est une liste contenant une sphere et un plan qui sont créés via l'appels des fonctions `create_Plan` et `create_Sphere`. **Vous ferez très attention à l'ordre des variables données en entrée : il faut qu'il soit le même que celui défini lors de la définition de vos fonctions `create_Plan` et `create_Sphere`.**