iNFO

IUT GRAND OUEST NORMANDIE

R 5.A.06

2023 – 2024

Sensibilisation à la programmation multimédia

CM n° 4
Raytracing

ANNE Jean-François

Sensibilisation à la programmation multimédia

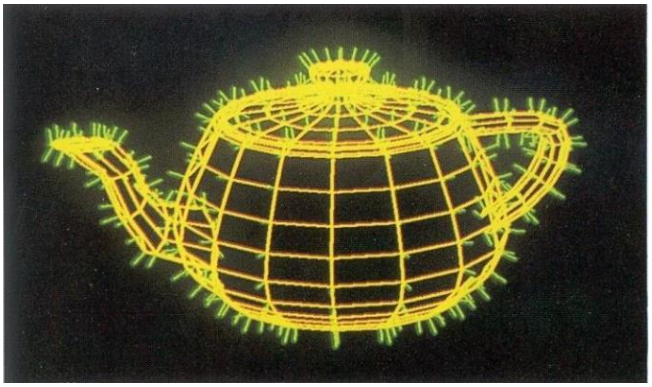
Programmation en 3D

Le but de ce TD est de se familiariser avec la programmation multimédia.

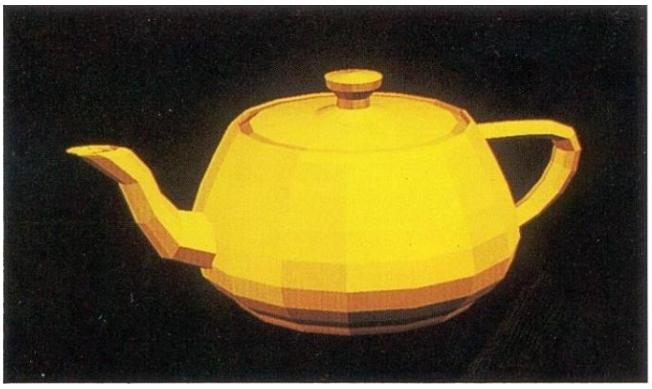
A. Historique :

Voici un résumé de l'historique de la création d'images en 3D jusqu'à l'avènement du ray tracing :

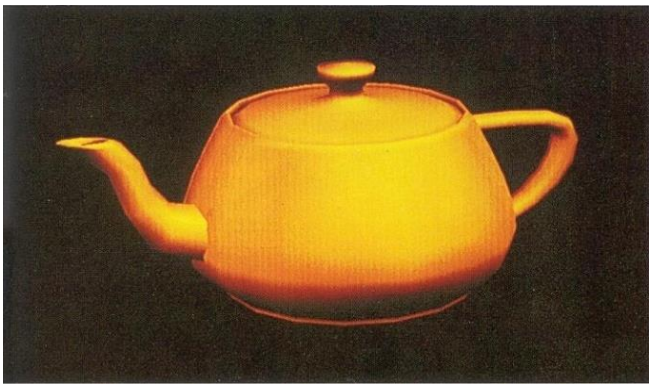
1. Modélisation filaire (Wireframe Modeling) : Dans les années 1960, les premières techniques de modélisation en 3D ont été développées. Elles se basaient sur des représentations filaires simples des objets, sans prendre en compte l'éclairage ou les surfaces réalistes.



2. Rendu par faces (Polygon Rendering) : Dans les années 1970, les graphistes ont commencé à utiliser des polygones pour modéliser les objets 3D. Les surfaces des polygones étaient colorées en fonction de leur position, mais les ombres et les effets de réflexion étaient encore absents.

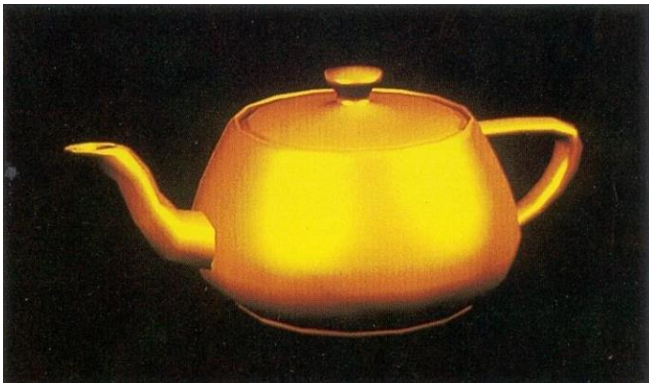


3. Rendu de Gouraud (Gouraud Shading) : Vers la fin des années 1970, le rendu de Gouraud a été introduit par Henri Gouraud. Cette technique a permis de calculer les intensités de couleur pour les sommets des polygones, puis de les interpoler sur les faces, créant ainsi une apparence plus lisse et réaliste.



4. Rendu de Phong (Phong Shading) : Dans les années 1980, Bui Tuong Phong a proposé une méthode plus avancée de rendu des surfaces en utilisant des équations d'éclairage pour calculer l'intensité de la lumière réfléchie par chaque point de surface. Le rendu de Phong a permis d'obtenir des surfaces plus

réalistes en prenant en compte la réflexion spéculaire.



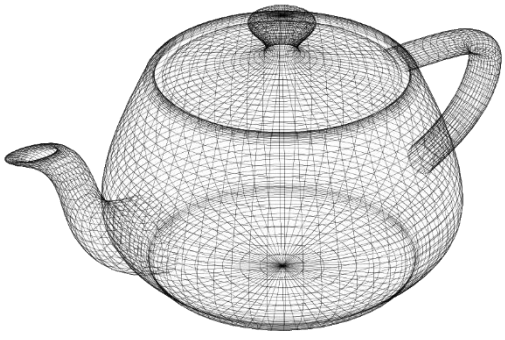
5. Rendu basé sur la texture (Texture Mapping) : Au cours des années 1980, les graphistes ont commencé à utiliser des textures pour appliquer des motifs ou des images sur les surfaces 3D. Cela a permis d'ajouter des détails réalistes aux objets en leur attribuant des caractéristiques visuelles plus complexes.
6. Rendu en temps réel (Real-Time Rendering) : À partir des années 1990, avec l'avènement des jeux vidéo, les techniques de rendu en temps réel ont connu un développement important. Les graphistes ont cherché des moyens d'accélérer les calculs tout en maintenant une apparence réaliste, en utilisant des méthodes d'approximation et d'optimisation.
7. Ray tracing : Dans les années 1980, le ray tracing est devenu une technique de rendu 3D avancée. Initialement, il était très coûteux en termes de puissance de calcul et de temps de rendu, mais avec les progrès de la technologie, il est devenu plus accessible. Le ray tracing simule le comportement de la lumière en traçant des rayons depuis la caméra virtuelle vers les objets de la scène, permettant ainsi de générer des images réalistes avec des ombres, des réflexions et des réfractions précises.

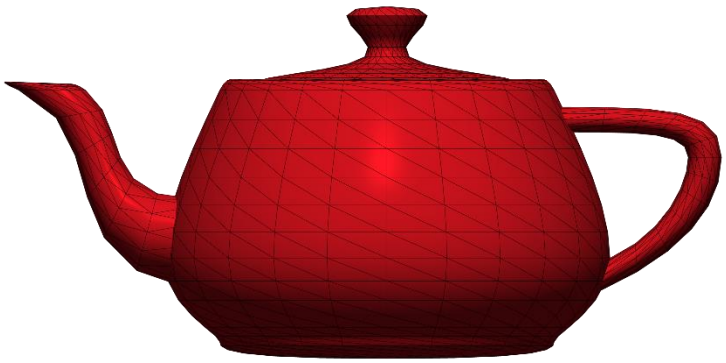
Depuis lors, le ray tracing a continué à évoluer et à s'améliorer, devenant une technique de rendu de plus en plus utilisée dans les jeux vidéo, les films d'animation et d'autres domaines de l'infographie. Il a été grandement favorisé par les avancées technologiques, telles que l'utilisation de processeurs graphiques (GPU) spécialisés dans le calcul parallèle et l'introduction de matériel dédié au ray tracing, tels que les cartes graphiques RTX de NVIDIA.

1°) Théière de l'Utah :

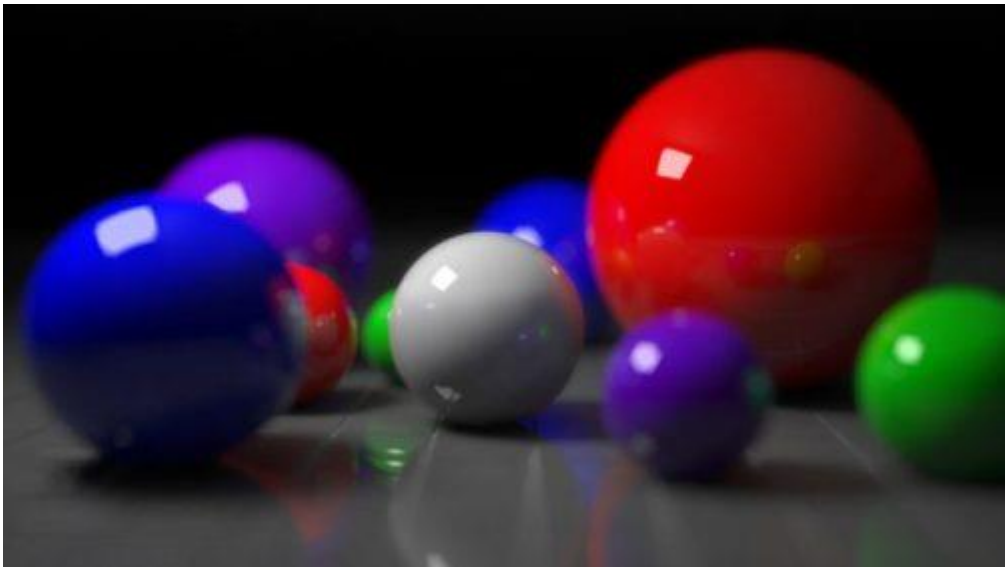
La théière de l'Utah (Utah teapot ou Newell teapot) est un modèle utilisé dans la synthèse d'image 3D qui est devenu un objet standard de référence dans la communauté de la synthèse d'image. Il s'agit du modèle mathématique d'une théière ordinaire qui est simple, ronde, solide, et partiellement convexe.

Le modèle de la théière fut créé en 1975 par un pionnier de la synthèse d'images, Martin Newell, à l'université de l'Utah.





B. Remis au gout du jour :



Nvidia présente le ray tracing comme une révolution graphique, bien que cette technologie de rendu d'image soit ancienne et repose sur la trajectoire des rayons lumineux. Voici pourquoi elle refait surface aujourd'hui.

Lors de la conférence du 20 août dernier, le géant de la carte graphique, Nvidia, a annoncé une nouvelle architecture baptisée Turing. Mais ce qui a vraiment captivé l'attention était le "R" de "RTX", qui annonçait la commercialisation à grande échelle d'une technologie capable de susciter l'émerveillement : le ray tracing, réputé pour apporter un réalisme inégalé dans les effets de lumière à l'écran.

Dans une démonstration assez spectaculaire, intitulée "Battlefield V: Official GeForce RTX Real-Time Ray Tracing Demo",:

- *Battlefield V: Official GeForce RTX Real-Time Ray Tracing Demo*

Nvidia plonge dans le décor du jeu de guerre *Battlefield V*. Le feu des canons se reflète autant sur la carrosserie des voitures que dans les flaques d'eau sur le bitume. Des éléments de la scène, qui sans *ray tracing* seraient sombres et ternes, se mettent à flamboyer. Pour comprendre ces techniques qui se veulent le *nec plus ultra* de l'image 3D, il faut se plonger tout au fond de son GPU, dans ce que l'on appelle *le pipeline graphique*.

1°) Au fond du pipeline graphique

Il est facile d'oublier une évidence : lorsque l'on joue à un jeu en "3D", l'image que l'on voit sur notre écran n'est pas en trois dimensions. Elle est en deux dimensions, car l'écran est plat. Même dans un casque de réalité virtuelle, ce que l'on voit ce sont deux scènes en 2D qui s'affichent devant nos deux yeux, et l'illusion de la stéréoscopie fait le reste. Pourtant, le jeu vidéo, tel qu'il est stocké dans l'ordinateur, est bien constitué de modèles en 3D, et il faut les reconstituer sur un écran en deux dimensions. C'est ce qu'on appelle le "rendu", et c'est le travail de la carte graphique.

Le fameux pipeline graphique est l'ensemble des processus nécessaires au rendu. On peut les regrouper en trois grandes étapes.

- Tout d'abord, il faut intégrer toutes les modifications qui ont pu se produire dans la scène, telles que les actions du joueur ou les mouvements des différents personnages. Cela s'appelle "l'application", et cela se déroule sur le processeur central (CPU), avant d'être transféré à la carte graphique.
- **Ensuite, la carte graphique prend le relais et se charge de la "géométrie", une série de tâches assez complexes consistant à ajuster les éléments de la scène, à ajouter une lumière ambiante ou à éliminer les objets qui se trouvent hors de la caméra.**

- La dernière étape, qui nous intéresse ici, est la rastérisation. Elle consiste à projeter les éléments 3D obtenus lors de l'étape de "géométrie" du pipeline sur la grille de pixels qui sera affichée à l'écran. Dans le langage courant, la rastérisation désigne une famille d'algorithmes qui rendent les éléments de la scène, appelés primitives, un par un. En 3D, les primitives sont généralement des polygones qui constituent chaque petite facette d'un objet en volume. On peut comparer cela à la peinture d'un tableau. L'algorithme le plus connu de cette famille est le rendu scanline, qui "peint le tableau" ligne par ligne en faisant ressortir les polygones les plus proches de la scène.

2°) Ray tracing et Ray casting

Cependant, la réalité du monde n'est pas comparable à un tableau flamand de la Renaissance. Les techniques de rendu scanline et autres algorithmes similaires ont du mal à prendre en compte les effets de la lumière. Dans la vie réelle, la lumière émise par le soleil ou d'autres sources d'éclairage rebondit sur les objets, se diffuse dans différentes directions, se réfléchit sur l'eau ou les surfaces métalliques, avant d'atteindre nos yeux. Tout cela crée des jeux d'ombres et de lumières, et affecte également la couleur perçue des éléments de notre environnement (une ombre, par exemple, n'est jamais entièrement noire).

Le ray tracing vise à reproduire le plus fidèlement possible ces phénomènes physiques. Les trajectoires des faisceaux de lumière sont calculées pour déterminer leurs rebonds. Au lieu de suivre le parcours des photons émis par la source de lumière, le ray tracing inverse cette logique : il n'est en effet pas nécessaire de consacrer des efforts aux photons qui se perdent dans l'environnement !

Seuls les rayons lumineux qui atteignent la caméra sont pris en compte, et leur trajectoire est calculée "à l'envers". Cette idée est similaire à celle des savants de l'Antiquité, qui pensaient que les yeux humains émettaient de la lumière.

Le ray casting a été implémenté dans les jeux vidéo dès 1992, et le ray tracing est un concept général qui existe depuis longtemps et a été mis en œuvre de différentes manières au fil du temps. L'algorithme original a été décrit en 1968 par le chercheur Arthur Appel d'IBM.

À l'époque, on parlait de ray casting, qui est l'implémentation la plus basique de ce principe. Des rayons virtuels partent de la caméra pour atteindre chaque pixel de l'écran ; lorsqu'ils rencontrent le premier objet, ils s'arrêtent. L'un des avantages de cette méthode était sa supériorité relative par rapport au rendu scanline pour les surfaces courbes.

La première implémentation en temps réel du *ray casting* a lieu en 1986, et son utilisation dans le jeu vidéo arrive assez vite.

- *Luxo Jr. (Pixar, 1986) John Lasseter*
- *Light and Heavy (Pixar, 1991) John Lasseter*
- *Surprise (Pixar, 1991) John Lasseter*
- *Up and Down (Pixar, 1991) John Lasseter*
- *Bande annonce de Pixar Animations Studios (2006)*

En 1991, le programmeur John Carmack s'est plongé dans le domaine de la tridimensionnalité sur ordinateur, qui était principalement réservé à des simulateurs de vol à l'époque. À cette époque, la 3D était trop exigeante par rapport à la puissance de calcul des ordinateurs personnels pour être utilisée dans des jeux d'action.

Carmack a utilisé le ray casting pour calculer uniquement les parties de la scène visibles par la caméra, plutôt que l'ensemble du décor. En seulement six semaines de travail, le programmeur a développé un moteur de ray casting et a sorti avec ses collègues le jeu de tir à la première personne Wolfenstein 3D en mai 1992 sur MS-DOS. D'autres jeux ont suivi au cours de la décennie.

***Wolfenstein 3D* (1992), ou comment le ray casting n'a rien à voir avec le réalisme des lumières**

3°) Pourquoi on n'en parle que maintenant ?

Cependant, la méthode de ray casting des années 90 est simpliste, car elle ne prend pas en compte les rayons qui sont diffractés plusieurs fois, rebondissant d'un objet à l'autre. Par conséquent, elle présente un intérêt très limité en termes d'effets de lumière et de réalisme visuel. Après les premiers travaux d'Arthur Appel sur le ray casting en 1968, Turner Whitted a été le premier à prendre en compte les rayons secondaires en 1979.

Whitted a proposé d'appliquer l'algorithme du ray casting de manière récursive en considérant trois types de rayons émis à chaque fois que la lumière touche un objet : la réflexion, la réfraction et l'ombre. La réflexion correspond à ce qui se produit dans un miroir ou une vitre, tandis que la réfraction entraîne des halos de lumière plus diffus. L'ombre est considérée comme un rayon "noir" distinct. Aujourd'hui, le terme de ray tracing désigne cette approche des rayons de lumière secondaires.

Cependant, le calcul des rayons secondaires est très lent. Très lent. Jusqu'à il y a une décennie, même les grandes productions hollywoodiennes hésitaient à l'utiliser pour leurs images de synthèse. Pixar, l'un des pionniers de l'animation 3D, n'a adopté le ray tracing qu'en 2013. Dans un environnement entièrement numérique, le calcul requis était tout simplement trop important auparavant. Par exemple, la réalisation du film Monsters University nécessitait l'allocation de 20 Go de RAM pour chaque image, sachant qu'il y a 24 images par seconde dans un film.

Si cela était le cas pour les studios disposant de supercalculateurs, capables de consacrer des heures, voire des jours, au rendu d'une scène, il est facile de comprendre que de telles ressources étaient inenvisageables en temps réel sur un simple PC. En 2008, Intel a démontré l'utilisation du ray tracing en temps réel dans le jeu de tir Enemy Territory: Quake Wars. Cependant, le jeu fonctionnait à une résolution de 1080 pixels par 720 pixels, avec une fréquence assez saccadée de 15 à 20 images par seconde, sur une configuration à quatre CPU haut de gamme (soit seize cœurs). Bien que les reflets dans l'eau étaient de bien meilleure qualité que le reste des graphismes, cela restait loin de ce que proposent les jeux de 2018, même en utilisant la méthode de rendu rasterisé. Quake Wars: Ray Traced est resté un projet de recherche et n'a jamais été commercialisé.

Même avec le *ray tracing*, les graphismes de *Quake Wars: Ray Traced* (2008) impressionnent peu aujourd'hui.

Le ray tracing utilisé seul ne peut pas accomplir des miracles. Il ne fait qu'approximer la façon dont la lumière se propage dans la réalité. Plus on cherche à rendre l'algorithme rapide, plus on le contraint à être approximatif. De plus, l'algorithme de ray tracing développé par Whitted a du mal à reproduire certains effets de lumière tels que la transluminance (lorsque vous mettez vos doigts devant une source de lumière et que vous voyez leurs bords briller en rouge, si vous avez la peau claire) ou la caustique (lorsque vous placez un verre sous le soleil et que vous observez les motifs lumineux projetés autour de lui).

C'est pourquoi, au cinéma, le ray tracing est généralement combiné à d'autres algorithmes tels que le photon mapping ou le path tracing, qui simulent de manière encore plus précise les phénomènes optiques naturels, bien que cela nécessite une puissance de calcul plus importante. D'autres techniques exploitent les textures visuelles des objets pour obtenir les effets désirés, mais nous n'entrerons pas dans les détails.

Alors, que dire des RTX de Nvidia ? Ce sont les premières cartes graphiques à apporter le ray tracing en temps réel au grand public.

C. Ray Tracing à partir de zéro en Python

1°) Créez une image générée par ordinateur à l'aide de l'algorithme de Ray Tracing (lancer de rayons) codé à partir de zéro en Python.

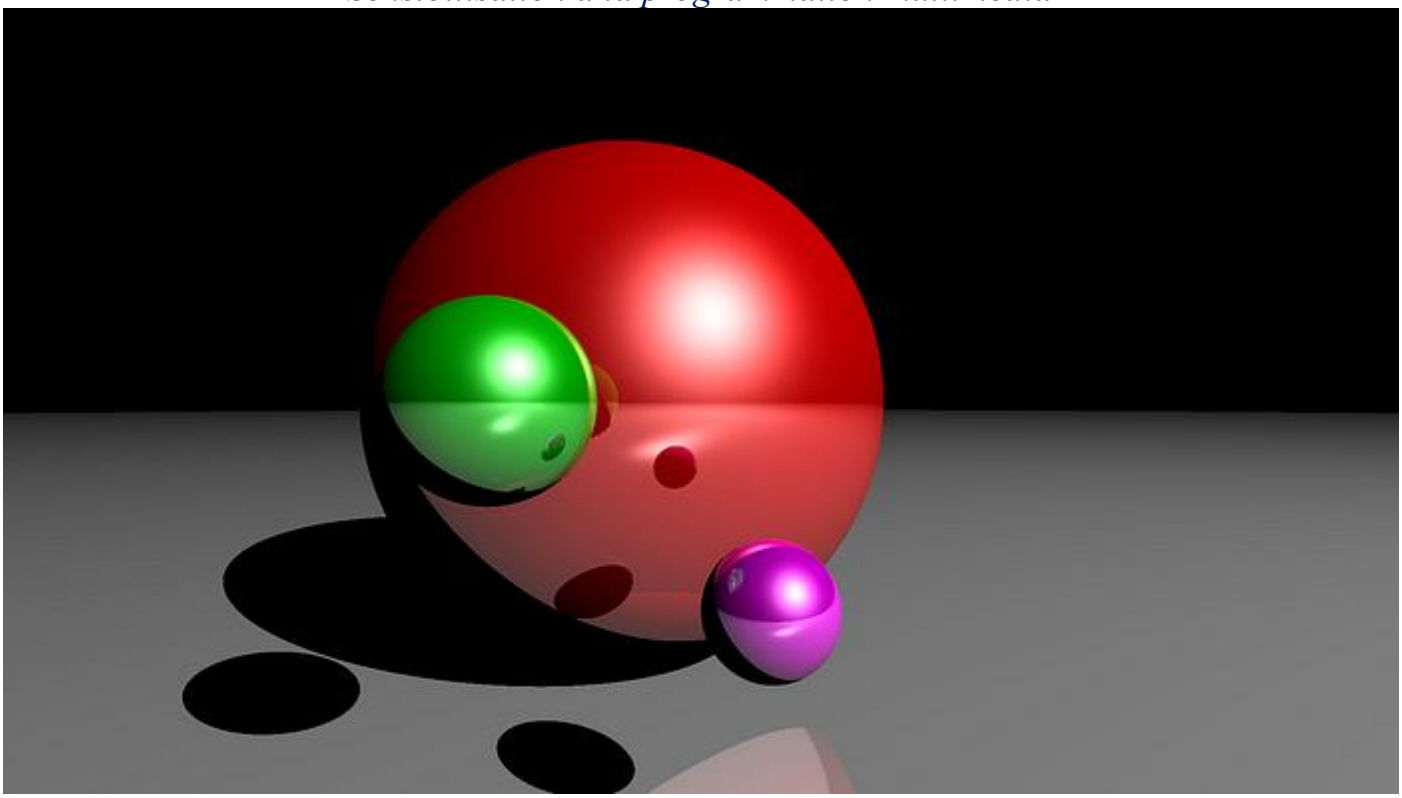


Figure. 1 — Image générée par ordinateur

Dans ce cours, nous allons avoir un aperçu de ce à quoi peuvent ressembler les algorithmes d'infographie. En particulier, avec un algorithme de *ray tracing* et montrer une implémentation simple en Python.

À la fin de ce cours, vous serez en mesure de créer un programme qui générera l'image ci-dessus, sans utiliser de bibliothèque graphique sophistiquée ! Seulement *NumPy*. N'est-ce pas fou ?! Allons-y !

P.S. Ce cours n'est en aucun cas un guide complet / explication du ray tracing, car il s'agit d'un sujet vaste, mathématique et complexe, mais plutôt une introduction pour les curieux :)

2°) Conditions préalables

Nous n'avons besoin que d'une géométrie vectorielle très basique.

1. Si vous avez 2 points A et B — quelle que soit la dimensionnalité : 1, 2, 3, ..., n — alors un vecteur qui va de A à B peut être trouvé en calculant `B - A` (élémentairement) ;
2. La longueur d'un vecteur — quelle que soit sa dimensionnalité — peut être trouvée en calculant la racine carrée de la somme des composantes carrées. La longueur d'un vecteur `v` est noté `||v||` ;
3. Un *vecteur unité* est un vecteur de longueur 1 : `||v|| = 1` ;
4. Étant donné un vecteur, un autre vecteur qui pointe dans la même direction mais avec une longueur de 1 peut être trouvé en divisant chaque composante du premier vecteur par sa longueur — c'est ce qu'on appelle la normalisation : `u = v / ||v||` ;
5. Produit de points pour les vecteurs. Plus précisément : $\langle v, v \rangle = ||v||^2$;
6. Résolution d'une équation quadratique ;
7. Un peu de patience et d'imagination ;

3°) Algorithme de lancer de rayons

En effet, le *ray tracing* est une technique de rendu **qui simule le** trajet de la lumière et les **intersections avec les objets** et **est** capable de produire des images avec un haut degré de réalisme. Des variantes plus optimisées de cet algorithme sont effectivement utilisées dans les jeux vidéo !

Pour expliquer l'algorithme, nous devons configurer une **scène** :

1. Nous avons besoin d'un espace **3D** (cela signifie simplement que nous allons utiliser **3** coordonnées (X, Y, Z) pour positionner les objets dans l'espace) ;
2. Nous avons besoin d'**objets** dans cet espace (puisque nous allons reproduire `la fig.1`, imaginez des sphères);
3. Nous avons besoin d'une source de **lumière** (ce sera un point unique émettant de la lumière dans toutes les directions, donc essentiellement une position unique) ;
4. Nous avons besoin d'un « œil » ou d'**une caméra** pour observer la scène (encore une fois, simplement une position) ;
5. Puisque la caméra pourrait regarder n'importe où vraiment, nous avons besoin d'**un écran** à travers lequel la caméra observera les objets (4 positions pour les quatre coins d'un écran rectangulaire) ;

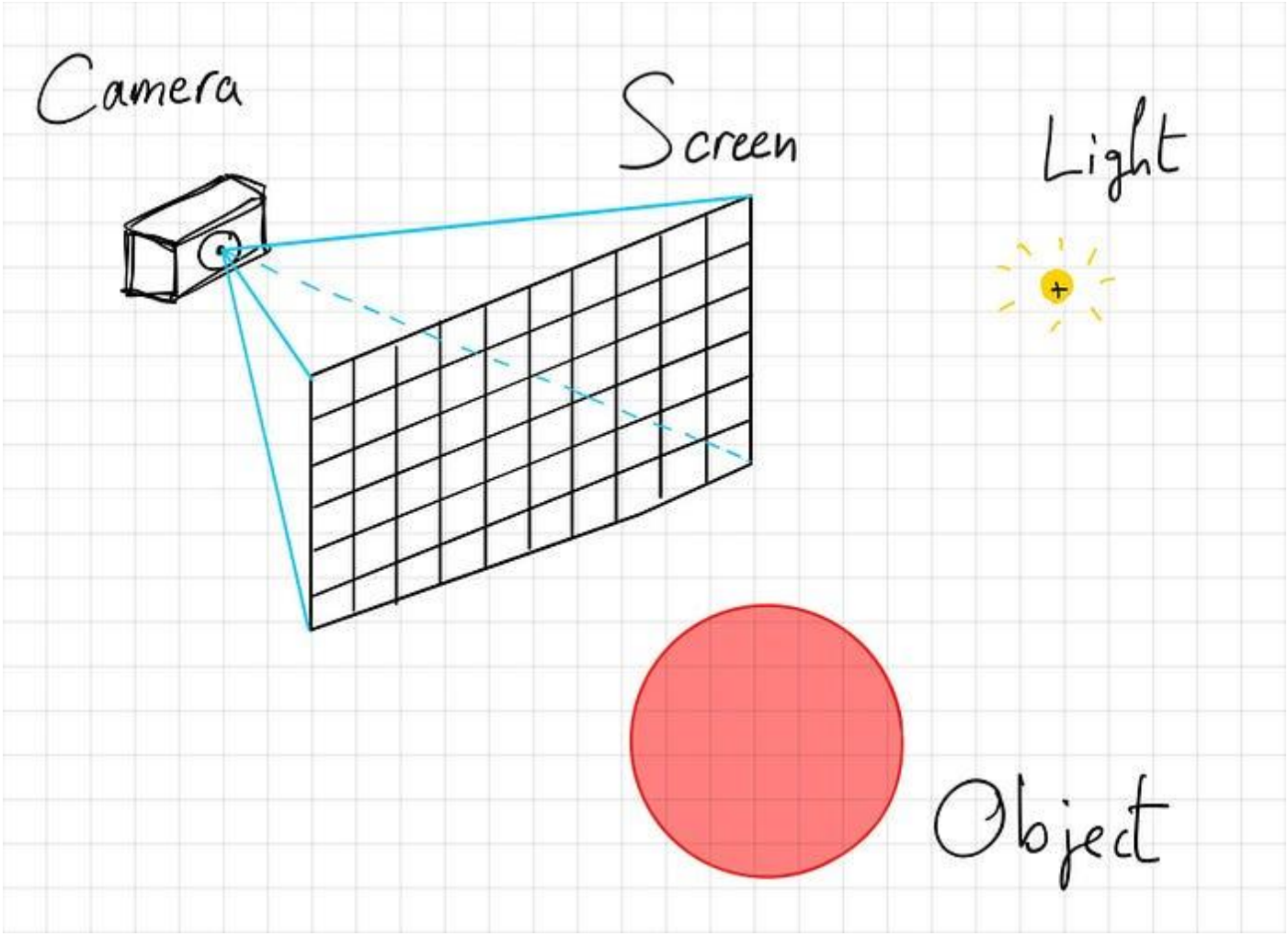


fig. 2

*Un mot sur **l'écran** :* l'écran va occuper un certain espace que vous allez définir (il peut s'agir d'un rectangle 3x2 par exemple). Mais 3 et 2 ne signifient pas vraiment quelque chose seuls. Ils signifient quelque chose quand vous les comparez à la taille des autres objets, ils sont relatifs. Ce qui est important ici, c'est comment vous allez diviser ce rectangle en carrés plus petits (pixels), comme la figure ci-dessus. Cela va déterminer la

taille de l'image finale. En d'autres termes, vous pouvez créer un rectangle 3x2 et le diviser en 300x200 pixels, cela fonctionnera très bien.

Compte tenu de la **scène**, voici l'algorithme de lancer de rayons :

```
Pour chaque pixel p(x,y,z) de l'écran :
    Associer une couleur noire à P
 Si le rayon (ligne) qui commence à la caméra et va vers p croise n'importe
quel objet de la scène, alors :
 Calculer le point d'intersection avec l'objet le plus proche
       S'il n'y a pas d'objet de la scène entre le point d'intersection
et la lumière, alors :
            Calculer la couleur du point d'intersection
            Associez la couleur du point d'intersection à p
```

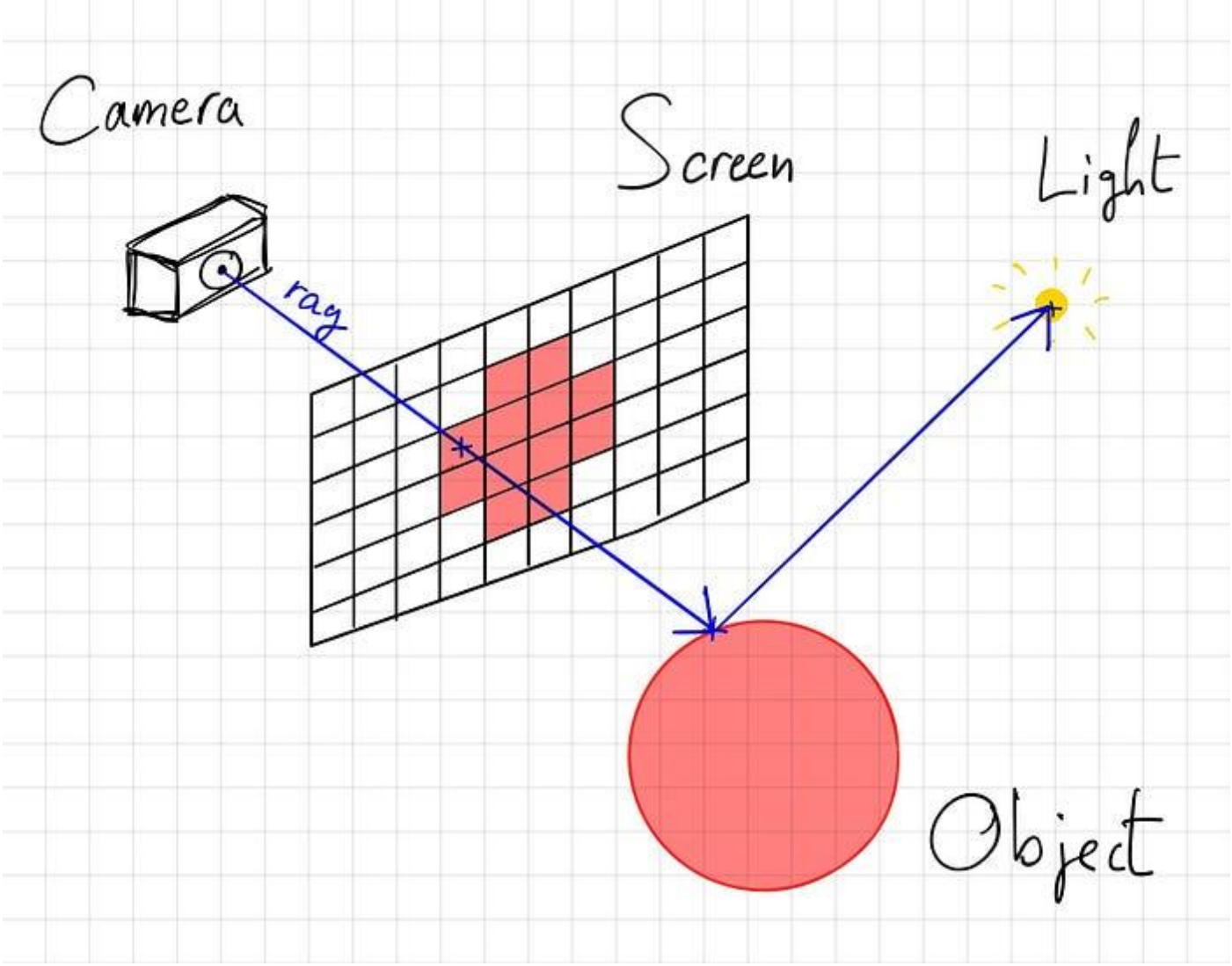


fig. 3

Notez que ce processus est en fait le processus inverse de l'éclairage réel. En réalité, la lumière sort de la source dans toutes les directions, rebondit sur les objets et frappe votre œil. Cependant, comme tous les rayons sortant de la source lumineuse ne se retrouveront pas dans votre œil, le lancer de rayons effectue le processus inverse pour économiser du temps de calcul (traces de rayons de l'œil à la source lumineuse).

Tout cela est purement géométrique, la seule chose que l'on n'a pas vu est comment `calculer la couleur du point d'intersection`. Ce n'est pas nécessaire pour le moment, alors nous le verrons plus tard. Sachez simplement qu'il existe des modèles physiques qui décrivent comment les objets sont éclairés lorsque la lumière les frappe avec un certain angle, intensité, etc.

À la fin de l'algorithme, nous aurons rempli l'***écran*** avec les couleurs correctes, et nous pourrons simplement l'enregistrer en tant qu'image.

4°) ***Configurer la scène***

Avant de commencer à coder, nous devons configurer une scène. Pour l'instant, nous allons décider où se trouvent la ***caméra*** et l'écran. Pour notre objectif, nous allons simplifier les choses en les alignant avec les axes de l'unité.

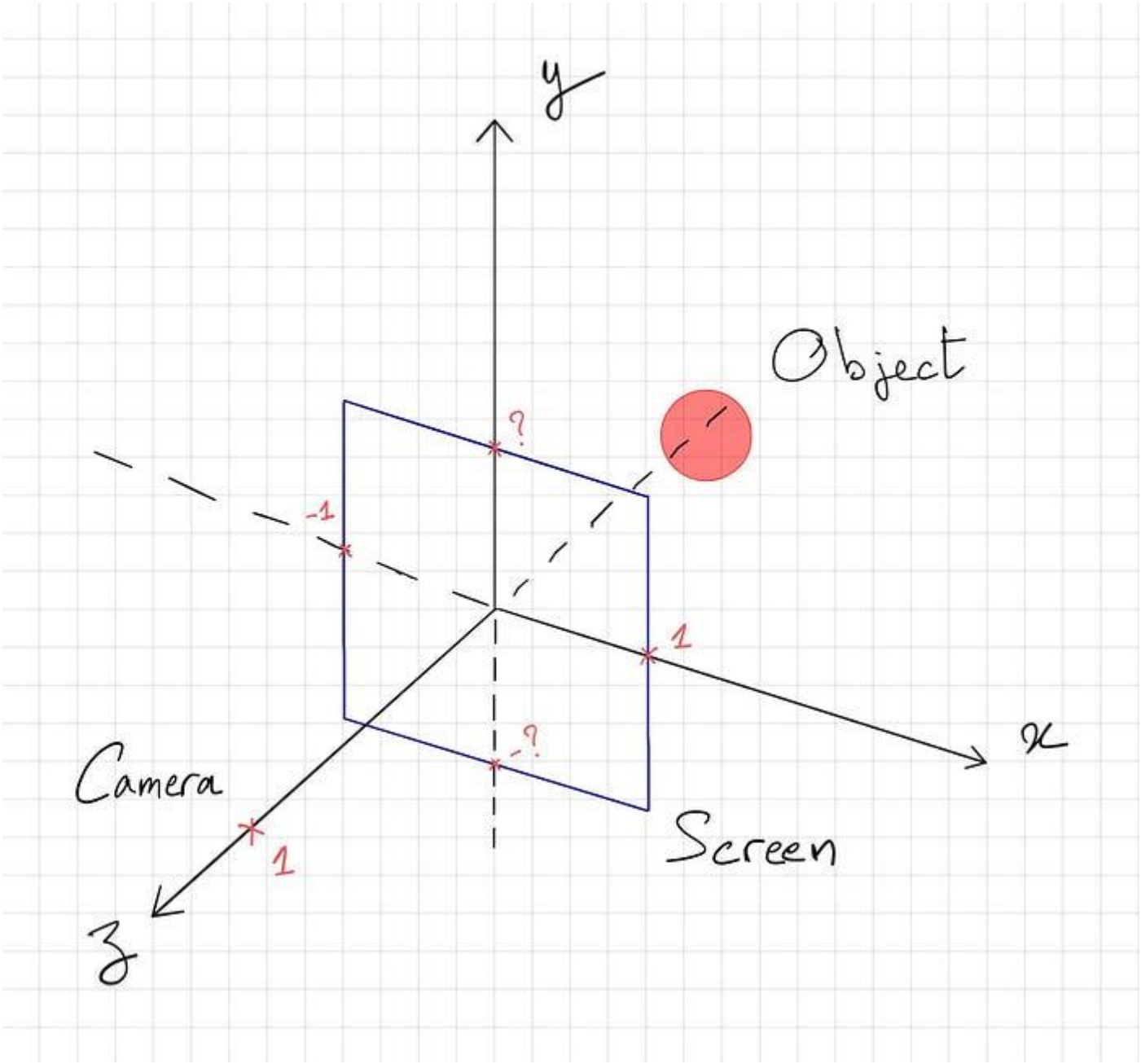


Fig.4 — Scène

Par conséquent, la ***caméra*** est située au point **(x = 0, y = 0, z = 1)** et l'***écran*** fait partie du plan formé par les axes ***x*** et ***y***. Avec cela étant mis en place, nous pouvons déjà écrire le squelette de notre code.

```
import numpy as np
import matplotlib.pyplot as plt

width = 300
height = 200

camera = np.array([0, 0, 1])
ratio = float(width) / height
screen = (-1, 1 / ratio, 1, -1 / ratio) # left, top, right, bottom

image = np.zeros((height, width, 3))
```

```
for i, y in enumerate(np.linspace(screen[1], screen[3], height)):
    for j, x in enumerate(np.linspace(screen[0], screen[2], width)):
        # image[i, j] = ...
        print("progress: %d/%d" % (i + 1, height))

plt.imsave('image.png', image)
```

Code. 1 — Squelette

1. La ***caméra*** est juste une position, 3 coordonnées ;

2. L'***écran***, quant à lui, est défini par quatre chiffres (ou deux points) : *gauche*, *haut*, *droite*, *bas*. Il varie de -1 à 1 dans la direction `x` (c'est arbitraire), et varie `de -1 / rapport à` 1 / rapport ***dans*** la direction `y`, où `le rapport` est ***la largeur de l'image / hauteur de l'image.*** La raison en est simple : nous voulons que l'***écran*** ait le même rapport hauteur / largeur que l'image réelle que nous voulons produire. La configuration ***de l'écran*** de cette façon produira un ratio d'aspect de (*largeur* sur *hauteur*) : `2 / (2 / ratio) = rapport` qui est le rapport de l'image souhaitée de `300x200` ;

3. Enfin, la boucle consiste à diviser l'écran en points `de largeur` et de `hauteur` dans les directions ***x*** et ***y*** respectivement, puis à calculer la couleur du pixel actuel ;

Vous pouvez réellement exécuter ce code et il produira – comme prévu pour l'instant – une image noire. Si vous regardez le pseudo-code, c'est ce que nous avons accompli.

```
✅ Pour chaque pixel p(x,y,z) de l'écran :
✅     Associer une couleur noire à P
 Si le rayon (ligne) qui commence à la caméra et va vers p croise n'importe quel objet de la scène, alors :
  Calculer le point d'intersection avec l'objet le plus proche
        S'il n'y a pas d'objet de la scène entre le point d'intersection et la lumière, alors :
            Calculer la couleur du point d'intersection
            Associez la couleur du point d'intersection à p
```

5°) Intersection de rayons

L'étape suivante de l'algorithme est la suivante :

```
Si le rayon (ligne) qui commence à la caméra et va vers p croise n'importe quel objet de la scène alors.
```

Décomposons-le en deux parties :

Tout d'abord, quel est `le rayon (ligne) qui commence à la` ***caméra*** `et va vers` ***p*** ?

a) Définition des rayons

Nous disons « *rayon* » mais ce n'est vraiment qu'un synonyme pour « *ligne* ». En général, chaque fois que vous codez quelque chose qui est géométrique, vous devriez préférer les vecteurs aux équations de ligne réelles, ils sont vraiment plus faciles pour travailler et sont beaucoup moins sujets aux erreurs telles que la division par zéro.

Ainsi, puisque le *rayon* commence à la ***caméra*** et va dans la direction du pixel actuellement ciblé, nous pouvons définir un vecteur unitaire qui pointe dans une direction similaire. Par conséquent, nous définissons un « *rayon qui commence à la **caméra** et va vers le **pixel*** » avec l'équation suivante :

$$ray(t) = camera + \frac{pixel - camera}{\|pixel - camera\|}t$$

Éq. 1 — rayon

N'oubliez pas que **Camera** et ***le pixel*** sont des points 3D. Pour `t = 0`, vous vous retrouvez à ***la position de la caméra, et plus vous augmentez*** t, `plus vous vous éloignez de` la caméra dans la direction du **pixel**. Il s'agit d'une équation paramétrique, qui donne **un point** le long de la droite pour un `t donné`.

Bien sûr, il n'y a rien de spécial à propos de l'***appareil photo*** ou du ***pixel***, nous pouvons également définir un rayon qui commence à l'origine (O) et va vers la destination (D) comme :

$$ray(t) = O + \frac{D - O}{\|D - O\|}t$$
$$= O + d \cdot t$$

Éq. 2 — rayon

Pour plus de commodité, nous définissons ***d*** comme le ***vecteur*** direction.

Nous pouvons maintenant compléter le code et ajouter le calcul du rayon.

```python
import numpy as np
import matplotlib.pyplot as plt

def normalize(vector):
    return vector / np.linalg.norm(vector)

width = 300
height = 200

camera = np.array([0, 0, 1])
ratio = float(width) / height
screen = (-1, 1 / ratio, 1, -1 / ratio) # left, top, right, bottom

image = np.zeros((height, width, 3))
for i, y in enumerate(np.linspace(screen[1], screen[3], height)):
    for j, x in enumerate(np.linspace(screen[0], screen[2], width)):
        pixel = np.array([x, y, 0])
        origin = camera
        direction = normalize(pixel - origin)

        # image[i, j] = ...
    print("progress: %d/%d" % (i + 1, height))

plt.imsave('image.png', image)
```

Code. 2 — Calcul des rayons

1. Nous avons ajouté la `fonction normalize(vector)` qui retourne un vecteur normalisé ;

2. Nous avons ajouté le calcul de l'`origine` et de la `direction` qui, ensemble, définissent un rayon. Notez que ***le pixel*** a un `z = 0` puisqu'il se trouve sur l'écran qui est contenu dans le plan formé par les ***axes x*** et ***y***;

Maintenant, nous arrivons à la deuxième partie qui croise `n'importe quel objet de la scène alors`. C'est essentiellement la partie « difficile ». Le calcul sera différent pour chaque type d'objets que nous traiterons (sphères, plans, triangles, etc.). Par souci de simplicité, nous ne rendrons que des sphères. Donc, pour la prochaine partie, nous allons voir :

1. Comment nous définissons une sphère ;

2. Comment nous calculons le *point d'intersection* entre un rayon et une sphère, s'il existe ;

6°) Définition de la sphère

Une sphère est en fait un objet mathématique assez simple à définir. *Une sphère est définie comme l'ensemble des points qui sont tous à la même distance r (rayon) d'un point donné (centre).*

Par conséquent, étant donné le centre ***C*** d'une sphère, et son rayon ***r***, un point arbitraire ***X*** se trouve sur la sphère si et seulement si :

$$\|X - C\| = r$$

Eq. 3— Equation de la sphère

Pour plus de commodité, nous mettons au carré les deux côtés pour nous débarrasser de la racine carrée causée par la magnitude de `X - C`.

$$\|X - C\|^2 = r^2$$

Eq. 4— intersection avec la sphère

Nous pouvons déjà définir certaines sphères justes après la déclaration d'écran.

```
objects = [
    { 'center': np.array([-0.2, 0, -1]), 'radius': 0.7 },
    { 'center': np.array([0.1, -0.3, 0]), 'radius': 0.1 },
    { 'center': np.array([-0.3, 0, 0]), 'radius': 0.15 }
]
```

Code. 4 — Définition des sphères

Calculons maintenant l'intersection entre un rayon et une sphère.

7°) Intersection de sphères

Nous connaissons l'équation des rayons, et nous savons à quelle condition un point doit satisfaire pour qu'il repose sur une sphère. Tout ce que nous avons à faire est de brancher `eq. 2` dans `eq. 4` et de résoudre pour `t`. Ce qui signifie, répondre à la question : *pour quel `t`, `rayon(t)` sera sur la sphère ?*

$$\begin{aligned}
\|ray(t) - C\|^2 &= r^2 \\
\|O + d \cdot t - C\|^2 &= r^2 \\
\langle d \cdot t + O - C, d \cdot t + O - C \rangle &= r^2 \\
\langle d, d \rangle t^2 + 2t\langle d, O - C \rangle + \langle O - C, O - C \rangle &= r^2 \\
\|d\|^2 t^2 + 2t\langle d, O - C \rangle + \|O - C\|^2 - r^2 &= 0
\end{aligned}$$

Équation 5 — intersection des sphères

C'est une équation quadratique ordinaire que nous pouvons résoudre pour `t`. Nous appellerons les coefficients associés respectivement à t^2, t^1, t^0 ***a***, ***b*** et ***c***. Calculons le discriminant de cette équation :

$$a = \|d\|^2 = 1$$
$$b = 2\langle d, O - C\rangle$$
$$c = \|O - C\|^2 - r^2$$
$$\Delta = b^2 - 4ac$$

Eq. 6 — discriminant

Puisque ***d*** (direction) est un vecteur unitaire, nous avons `a=1`. Une fois que nous avons calculé le discriminant de cette équation, il y a 3 possibilités :

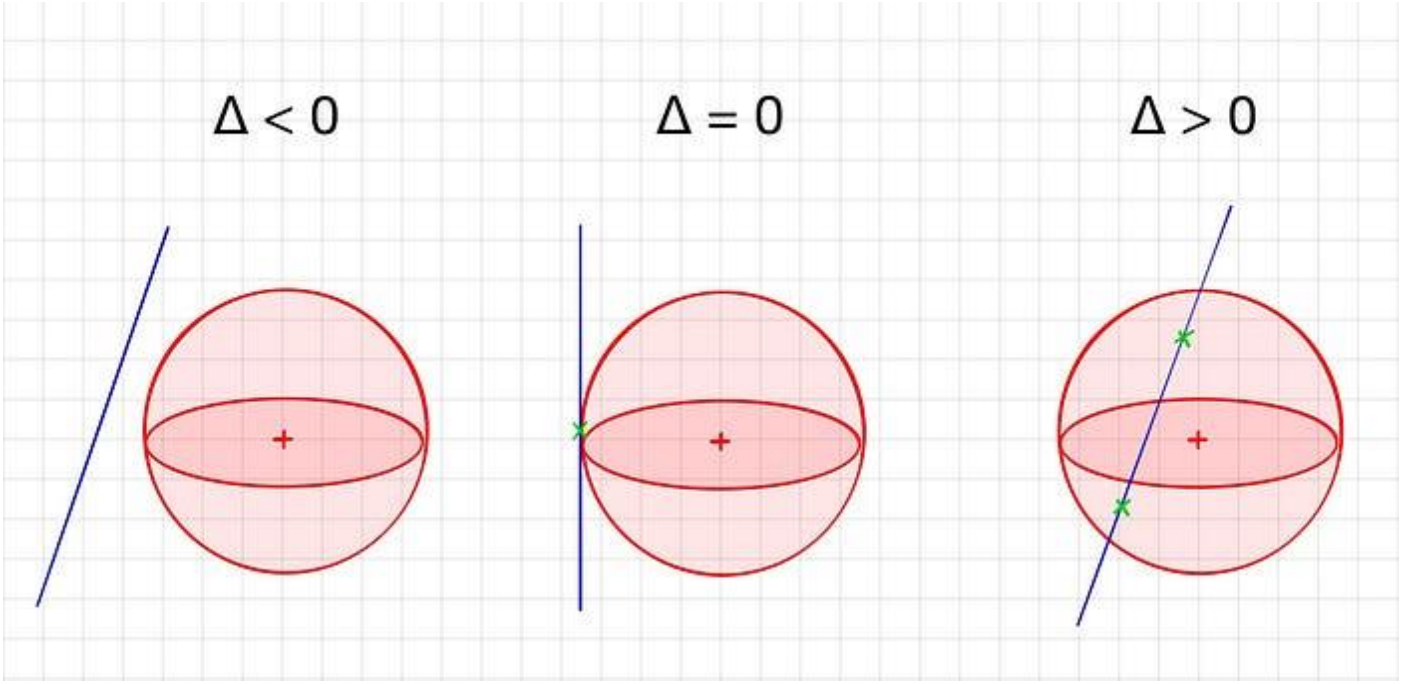


Figure. 5— discriminant sur la Sphère

Nous n'utiliserons le **troisième** cas que pour détecter les intersections. Voici une fonction qui peut détecter les intersections entre un rayon et une sphère. Il `retournera t` la distance de l'origine du rayon ***au*** point d'***intersection le plus proche*** si le rayon coupe réellement la sphère, et il retournera `None` autrement.

```
def sphere_intersect(center, radius, ray_origin, ray_direction):
    b = 2 * np.dot(ray_direction, ray_origin - center)
    c = np.linalg.norm(ray_origin - center) ** 2 - radius ** 2
    delta = b ** 2 - 4 * c
    if delta > 0:
        t1 = (-b + np.sqrt(delta)) / 2
        t2 = (-b - np.sqrt(delta)) / 2
        if t1 > 0 and t2 > 0:
            return min(t1, t2)
    return None
```

Code. 3— Intersection des sphères

Notez que nous ne retournons que l'intersection **la plus proche** (car il y en a 2) uniquement lorsque `t1` et `t2` sont positifs. En effet, un `t` qui résout l'équation pourrait être négatif, mais cela signifierait que le rayon qui coupe la sphère n'a pas d ***comme*** vecteur de direction, *mais* -d ***(par exemple si la sphère est derrière la*** caméra *et l'*écran*)*.

8°) ***Objet intersecté le plus proche***

D'accord, jusqu'ici tout va bien, mais nous n'avons toujours pas terminé l'instruction du pseudo-code qui était :

```
Si le rayon (ligne) qui commence à la caméra et va vers p coupe n'importe
quel objet de la scène alors[...] .
```

La bonne nouvelle est que nous pouvons faire cela et la prochaine instruction en une seule fois ! L'instruction suivante est :

```
Calculer le point d'intersection avec l'objet le plus proche.
```

Nous pouvons facilement créer une fonction qui utilise `sphere_intersect()` pour trouver l'objet le plus proche qu'un rayon croise, s'il existe. Nous faisons simplement une boucle sur toutes les sphères, recherchons des intersections et gardons la sphère la plus proche.

```
def nearest_intersected_object(objects, ray_origin, ray_direction):
    distances = [sphere_intersect(obj['center'], obj['radius'], ray_origin,
ray_direction) for obj in objects]
    nearest_object = None
    min_distance = np.inf
    for index, distance in enumerate(distances):
        if distance and distance < min_distance:
            min_distance = distance
            nearest_object = objects[index]
    return nearest_object, min_distance
```

Code. 5 — Vérifiez l'intersection avec l'objet le plus proche

Lors de l'appel de la fonction, si nearest_object `est Aucun`, il n'y a pas d'objet recoupé par le rayon, sinon sa valeur est `l` 'objet intersecté le plus proche **et nous obtenons** min_distance, `la distance de l'`origine du rayon *au* point d'*intersection*.

9°) ***Point d'intersection***

Afin de calculer le point d'intersection, nous utilisons la fonction précédente :

```
nearest_object, distance = nearest_intersected_object(objects, o, d)
if nearest_object:
    intersection_point = o + d * distance
```

Hourra! Nous avons terminé les deuxième et troisième instructions. Voici le code que nous avons jusqu'à présent :

```
def nearest_intersected_object(objects, ray_origin, ray_direction):
    distances = [sphere_intersect(obj['center'], obj['radius'], ray_origin,
ray_direction) for obj in objects]
    nearest_object = None
    min_distance = np.inf
    for index, distance in enumerate(distances):
        if distance and distance < min_distance:
            min_distance = distance
            nearest_object = objects[index]
    return nearest_object, min_distanceimport numpy as np
import matplotlib.pyplot as plt

def normalize(vector):
    return vector / np.linalg.norm(vector)

def sphere_intersect(center, radius, ray_origin, ray_direction):
    b = 2 * np.dot(ray_direction, ray_origin - center)
    c = np.linalg.norm(ray_origin - center) ** 2 - radius ** 2
    delta = b ** 2 - 4 * c
```

```
    if delta > 0:
        t1 = (-b + np.sqrt(delta)) / 2
        t2 = (-b - np.sqrt(delta)) / 2
        if t1 > 0 and t2 > 0:
            return min(t1, t2)
    return None

def nearest_intersected_object(objects, ray_origin, ray_direction):
    distances = [sphere_intersect(obj['center'], obj['radius'], ray_origin,
ray_direction) for obj in objects]
    nearest_object = None
    min_distance = np.inf
    for index, distance in enumerate(distances):
        if distance and distance < min_distance:
            min_distance = distance
            nearest_object = objects[index]
    return nearest_object, min_distance

width = 300
height = 200

camera = np.array([0, 0, 1])
ratio = float(width) / height
screen = (-1, 1 / ratio, 1, -1 / ratio) # left, top, right, bottom

objects = [
    { 'center': np.array([-0.2, 0, -1]), 'radius': 0.7 },
    { 'center': np.array([0.1, -0.3, 0]), 'radius': 0.1 },
    { 'center': np.array([-0.3, 0, 0]), 'radius': 0.15 }
]

image = np.zeros((height, width, 3))
for i, y in enumerate(np.linspace(screen[1], screen[3], height)):
    for j, x in enumerate(np.linspace(screen[0], screen[2], width)):
        pixel = np.array([x, y, 0])
        origin = camera
        direction = normalize(pixel - origin)

        # check for intersections
        nearest_object, min_distance = nearest_intersected_object(objects, origin,
direction)
        if nearest_object is None:
            continue

        # compute intersection point between ray and nearest object
        intersection = origin + min_distance * direction

        # image[i, j] = ...
        print("%d/%d" % (i + 1, height))

plt.imsave('image.png', image)
```

Code. 6 — Résumé

✅ for each pixel ***p***(x,y,z) of the screen:
✅ associate a black color to ***p***
✅ if the ray (line) that starts at ***camera*** and goes towards ***p*** intersects any object of the scene then:
✅ calculate the ***intersection point*** to the nearest object
if there is no object of the scene in-between the ***intersection point*** and the ***light*** then:
calculate the color of the ***intersection point***
associate the color of the ***intersection point*** to ***p***

10°) Intersection lumineuse

Jusqu'à présent, nous savons s'il y a une ligne droite qui va de la caméra / œil à un objet, et nous savons de quel objet il s'agit, ainsi qu'exactement quelle partie de l'objet nous regardons. Ce que nous ne savons pas encore, c'est si ce point spécifique est éclairé du tout ! Peut-être que la lumière n'est pas frappante sur ce point particulier, et qu'il n'est donc pas nécessaire d'aller plus loin parce que nous ne pouvons pas le voir. Par conséquent, l'étape suivante consiste à vérifier `s'il n'y a pas d'objet de la scène entre le point d'`***`intersection`*** `et la` ***`lumière`***.

Heureusement, nous avons déjà une fonction pour nous aider : `nearest_intersected_object()`. En effet, nous voulons savoir si le rayon qui part du point d'***intersection*** et va vers ***la lumière croise un objet*** de la scène avant de traverser la lumière. C'est pratiquement la même tâche que précédemment, il suffit de changer l'origine et la direction du rayon. Tout d'abord, nous devons définir une lumière. Vous pouvez ajouter ceci près de la déclaration d'objets :

```
light = { 'position': np.array([5, 5, 5]) }
```

Pour vérifier si un objet fait de l'ombre au point d'intersection, nous devons passer le rayon qui commence au point d'***intersection et se dirige vers*** la lumière, ***et voir si l'objet le plus proche renvoyé est réellement plus proche que la lumière du point d'intersection (en d'autres termes, entre les deux).***

Ça a l'air chouette, n'est-ce pas ? Eh bien, cela ne fonctionnera pas... Nous devons faire un léger ajustement. Si nous utilisons ***le point d'intersection comme origine du nouveau rayon, nous pourrions finir par détecter la sphère où nous nous trouvons actuellement comme objet entre le point*** d'intersection et la lumière. Une solution rapide et largement utilisée pour ce problème est de faire un petit pas qui nous éloigne de la surface de la sphère. Nous utilisons généralement un vecteur normal à la surface et faisons un petit pas dans cette direction.

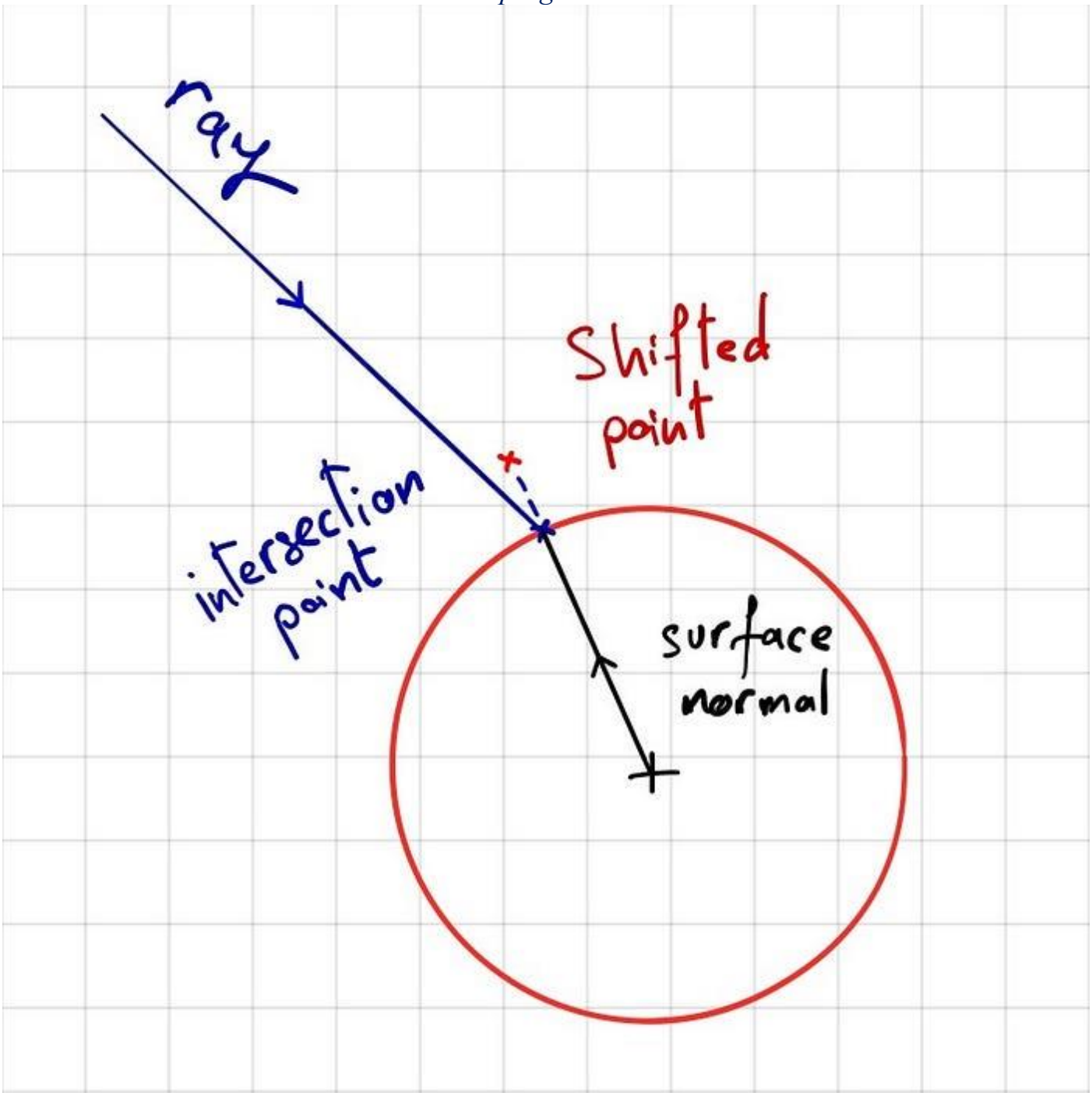


Figure. 6 — Étape normale de la sphère

Cette astuce n'est pas utilisée seulement pour les sphères, mais pour n'importe quel objet type.

Par conséquent, le code correct est :

```
# ...
intersection = origin + min_distance * direction

normal_to_surface = normalize(intersection - nearest_object['center'])
shifted_point = intersection + 1e-5 * normal_to_surface
intersection_to_light = normalize(light['position'] - shifted_point)

_, min_distance = nearest_intersected_object(objects, shifted_point,
intersection_to_light)
intersection_to_light_distance = np.linalg.norm(light['position'] - intersection)
is_shadowed = min_distance < intersection_to_light_distance

if is_shadowed:
    continue
```

Code. 7 — light intersection

```
✅ for each pixel p(x,y,z) of the screen:
✅    associate a black color to p
✅    if the ray (line) that starts at camera and goes towards p intersects any object of the scene then:
✅        calculate the intersection point to the nearest object
✅        if there is no object of the scene in-between the intersection point and the light then:
            calculate the color of the intersection point
            associate the color of the intersection point to p
```

1. Modèle de réflexion Blinn-Phong

C'est tout, la dernière partie. Nous savons qu'un faisceau lumineux a caressé l'objet, et que la réflexion du faisceau est entrée directement dans la ***caméra***. La question est : Que voit la caméra ? C'est ce à quoi le modèle Blinn-Phong tente de répondre.

FYI: Le modèle **Blinn-Phong** est une approximation du modèle **Phong** qui est moins intensive en calcul.

Selon ce modèle, tout matériau possède 4 propriétés :

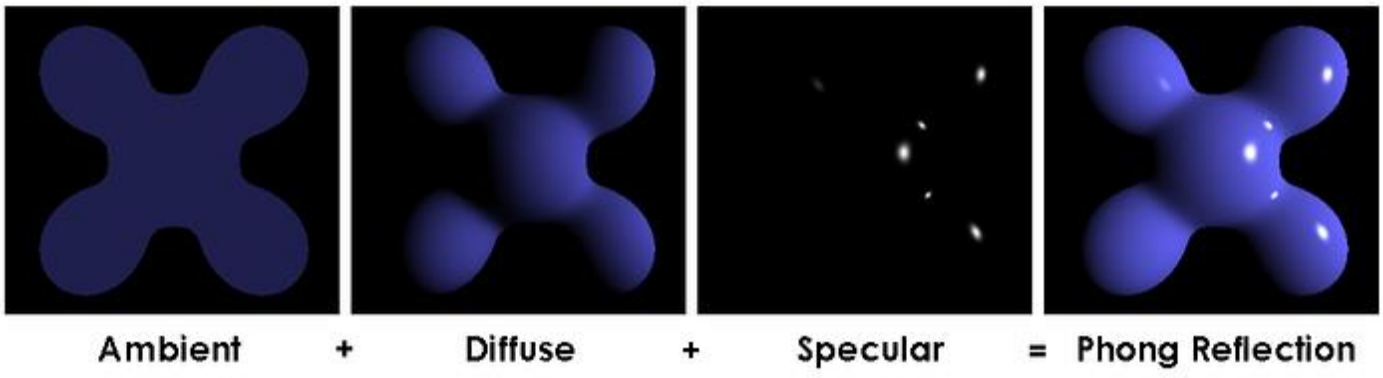
1. **Couleur ambiante** : couleur qu'un objet est censé avoir en l'absence de lumière. C'est difficile à imaginer, car nous ne voyons des objets que lorsque la lumière les frappe, mais généralement c'est une couleur pâle teintée de la couleur réelle que vous imaginez ;

2. **Couleur** diffuse : couleur qui se rapproche le plus de ce à quoi nous pensons lorsque nous disons « couleur » ;

3. **Couleur spéculaire** : couleur de la partie brillante d'un objet lorsque la lumière a un trait sur celui-ci. La plupart du temps, c'est blanc ;

4. **Brillance** : un coefficient représentant la brillance d'un objet ;

Remarque : *Toutes les couleurs sont* ***des représentations RVB*** *comprises entre* ***0 et 1****.*



Phong reflection model — Wikipedia

Donc, chaque objet de la scène doit avoir ces 4 propriétés. Ajoutons-les aux sphères.

```
objects = [
    { 'center': np.array([-0.2, 0, -1]), 'radius': 0.7, 'ambient': np.array([0.1, 0, 0]), 'diffuse': np.array([0.7, 0, 0]), 'specular': np.array([1, 1, 1]), 'shininess': 100 },
    { 'center': np.array([0.1, -0.3, 0]), 'radius': 0.1, 'ambient': np.array([0.1, 0, 0.1]), 'diffuse': np.array([0.7, 0, 0.7]), 'specular': np.array([1, 1, 1]), 'shininess': 100 },
    { 'center': np.array([-0.3, 0, 0]), 'radius': 0.15, 'ambient': np.array([0, 0.1, 0]), 'diffuse': np.array([0, 0.6, 0]), 'specular': np.array([1, 1, 1]), 'shininess': 100 }
]
```

Code. 8 — Propriétés de couleur de la sphère

Dans cet exemple, les sphères sont respectivement rouges, magenta et vertes.

Le modèle de Blinn-Phong indique que la lumière possède également les trois propriétés de couleur : ambiante, diffuse et spéculaire. Ajoutons-les aussi.

```
light = { 'position': np.array([5, 5, 5]), 'ambient': np.array([1, 1, 1]), 'diffuse': np.array([1, 1, 1]), 'specular': np.array([1, 1, 1]) }
```

Code. 9 — Propriétés de la couleur de la lumière

Compte tenu de ces propriétés, le modèle de Blinn-Phong calcule l'éclairement d'un point comme suit:

$$I_p = k_a * i_a + k_d * i_d * L \cdot N + k_s * i_s * (N \cdot \frac{L+V}{\|L+V\|})^{\frac{\alpha}{4}}$$

Eq. 7 — Blinn-Phong model

où

1. `ka`, `kd`, `ks` sont les propriétés *ambiantes, diffuses et spéculaires* de l'**objet**;
2. `ia`, id, `sont` les propriétés `ambiantes, diffuses, spéculaires` de la *lumière*;
3. `L` est un **vecteur unité** de direction depuis le point d'intersection vers la ***lumière*** ;
4. `N` est le **vecteur normal unitaire** à la surface de l'objet au point d'intersection ;
5. `V` est un **vecteur unité** de direction du point d'intersection vers la ***caméra*** ;
6. α est la **brillance** de l'objet;

```
# ...
if is_shadowed:
    break

# RGB
illumination = np.zeros((3))

# ambiant
illumination += nearest_object['ambient'] * light['ambient']

# diffuse
illumination += nearest_object['diffuse'] * light['diffuse'] *
np.dot(intersection_to_light, normal_to_surface)

# specular
intersection_to_camera = normalize(camera - intersection)
H = normalize(intersection_to_light + intersection_to_camera)
illumination += nearest_object['specular'] * light['specular'] *
np.dot(normal_to_surface, H) ** (nearest_object['shininess'] / 4)

image[i, j] = np.clip(illumination, 0, 1)
```

Code. 10 — Blinn-Phong

Notez qu'à la fin, nous avons lié la couleur entre ***0*** et ***1*** pour nous assurer qu'elle est dans la plage correcte.

- ✅ `for each pixel p(x,y,z) of the screen:`
- ✅ `  associate a black color to p`

✅ if the ray (line) that starts at ***camera*** and goes towards ***p*** intersects any object of the scene then:
✅ calculate the ***intersection point*** to the nearest object
✅ if there is no object of the scene in-between the ***intersection point*** and the ***light*** then:
✅ calculate the color of the ***intersection point***
✅ associate the color of the ***intersection point*** to ***p***

1. Exécutez le code !

Augmentez la ***largeur*** et la ***hauteur*** pour une résolution plus élevée (au prix de votre temps).

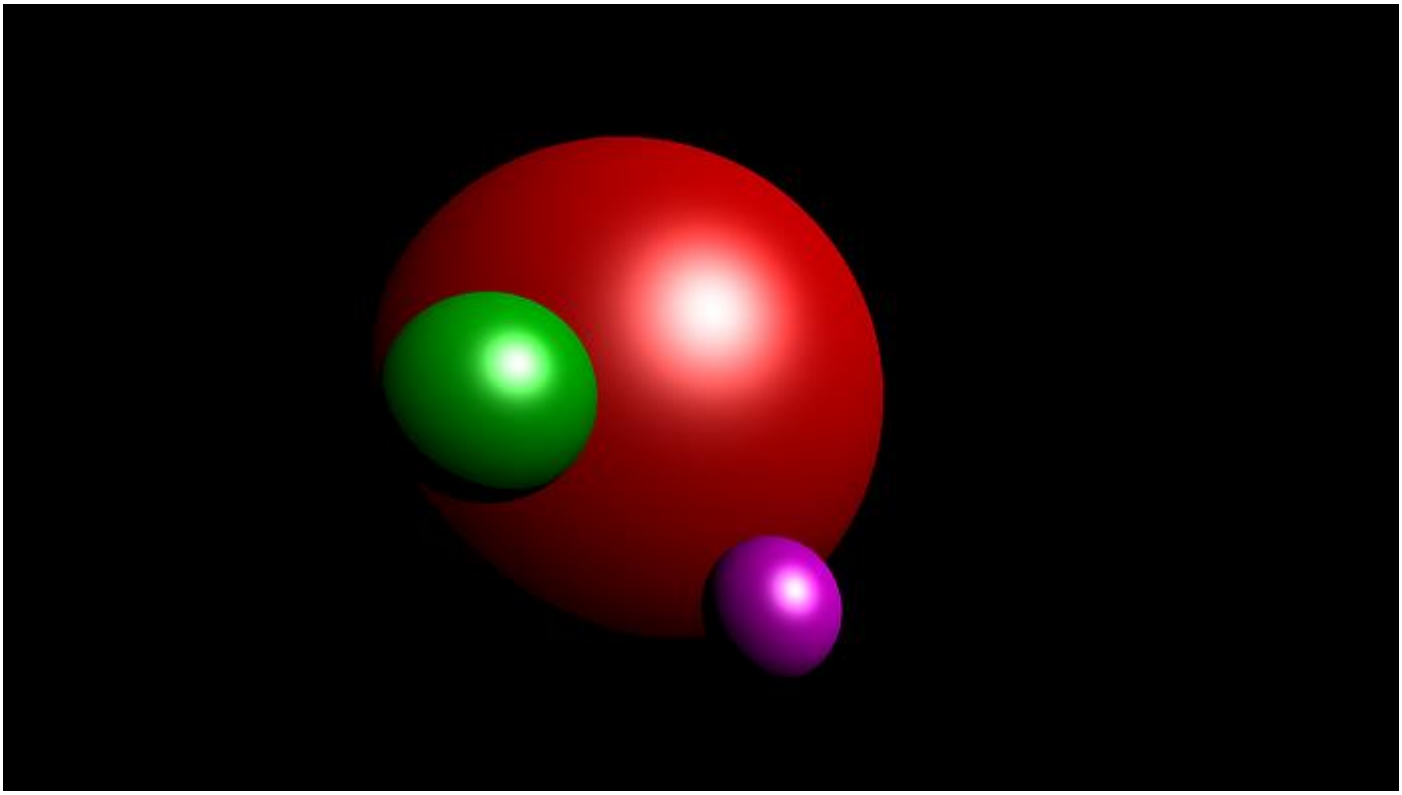


Figure. 5 — Premier résultat

Wow c'est cool ! Cependant, vous remarquerez peut-être 2 choses qui diffèrent de la première image que j'ai montrée au début. Allez-y, jetez un coup d'œil en arrière.

1. Le sol gris est manquant ;

2. Il n'y a pas de reflets (effet miroir) dans cette image ;

Abordons ces deux points.

11°) Faux plancher

Idéalement, nous créerions un autre type d'objet, un plancher, mais parce que nous sommes paresseux, nous pouvons simplement utiliser une autre sphère. *Comment ?* Eh bien, si vous vous tenez sur une sphère qui a un rayon infiniment grand (par rapport à votre taille), alors vous aurez l'impression de vous tenir sur une surface plane. Tout comme la terre :)

Ajoutez cette sphère à votre liste d'objets, et effectuez à nouveau le rendu !

```
{ 'center': np.array([0, -9000, 0]), 'radius': 9000 - 0.7, 'ambient': np.array([0.1, 0.1, 0.1]), 'diffuse': np.array([0.6, 0.6, 0.6]), 'specular': np.array([1, 1, 1]), 'shininess': 100 }
```

a) Réflexion

À l'heure actuelle, nous rendons des rayons qui : sortent de la source de lumière, frappent la surface d'un objet, puis rebondissent directement vers la caméra. Que se passe-t-il si le rayon frappe plusieurs objets avant de frapper la caméra ? C'est de la réflexion. Le rayon accumulera différentes couleurs et lorsqu'il frappera l'appareil photo, vous verrez des reflets. Allons-y.

Chaque objet a un coefficient de **réflexion** compris entre ***0 et 1***. « 0 » signifie que l'objet est mat, « 1 » signifie que l'objet est comme un miroir. Ajoutons une propriété réflexion à toutes les sphères :

```
{ 'center': np.array([-0.2, 0, -1]), ..., 'reflection': 0.5 }
{ 'center': np.array([0.1, -0.3, 0]), ..., 'reflection': 0.5 }
{ 'center': np.array([-0.3, 0, 0]), ..., 'reflection': 0.5 }
{ 'center': np.array([0, -9000, 0]), ..., 'reflection': 0.5 }
```

b) Algorithme

Actuellement, nous calculons un rayon qui commence à ***la caméra*** et se dirige vers un ***pixel,*** *puis nous traçons ce rayon dans la scène, vérifions l'intersection la plus proche et calculons la couleur du point d'intersection.*

Afin d'inclure des réflexions, nous devons tracer le **rayon réfléchi** après une intersection et inclure la contribution de couleur de chaque point d'intersection. Nous répétons ce processus un certain nombre de fois (pour définir).

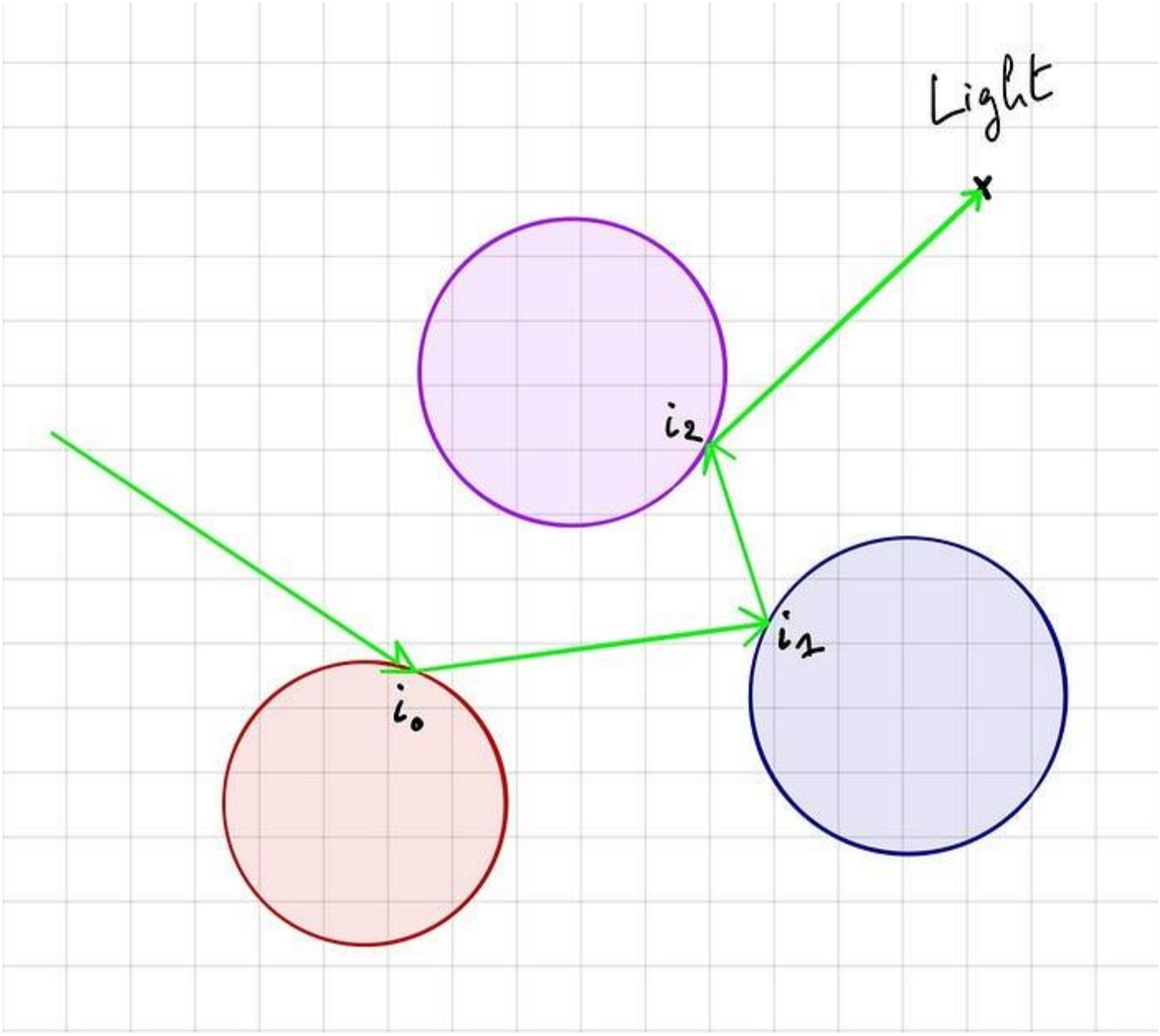


Figure 6 — réflexion

1. Calcul des couleurs

Afin d'obtenir la couleur d'un pixel, nous devons additionner la contribution de chaque point intersecté par le rayon.

$$c_p = i_0 + r_0 i_1 + r_0 r_1 i_2 + r_0 r_1 r_2 i_3 + \dots$$

Équation 8 — calcul des couleurs

où

1. `c`'est la couleur (finale) d'un pixel;
2. `I` est l'illumination calculée par le modèle de Blinn-Phong du point d'intersection #index ;
3. `r` est le reflet de l' *objet #index* intersecté;

Ensuite, c'est à vous de décider quand arrêter de calculer cette somme (c'est-à-dire quand arrêter de tracer les rayons réfléchis).

1. Rayon réfléchi

Avant de pouvoir coder cela, nous devons trouver la direction du rayon réfléchi. Nous pouvons calculer un rayon réfléchi de la manière suivante :

$$R = V - 2(V \cdot N)N$$

Eq. 9 — Réflexion

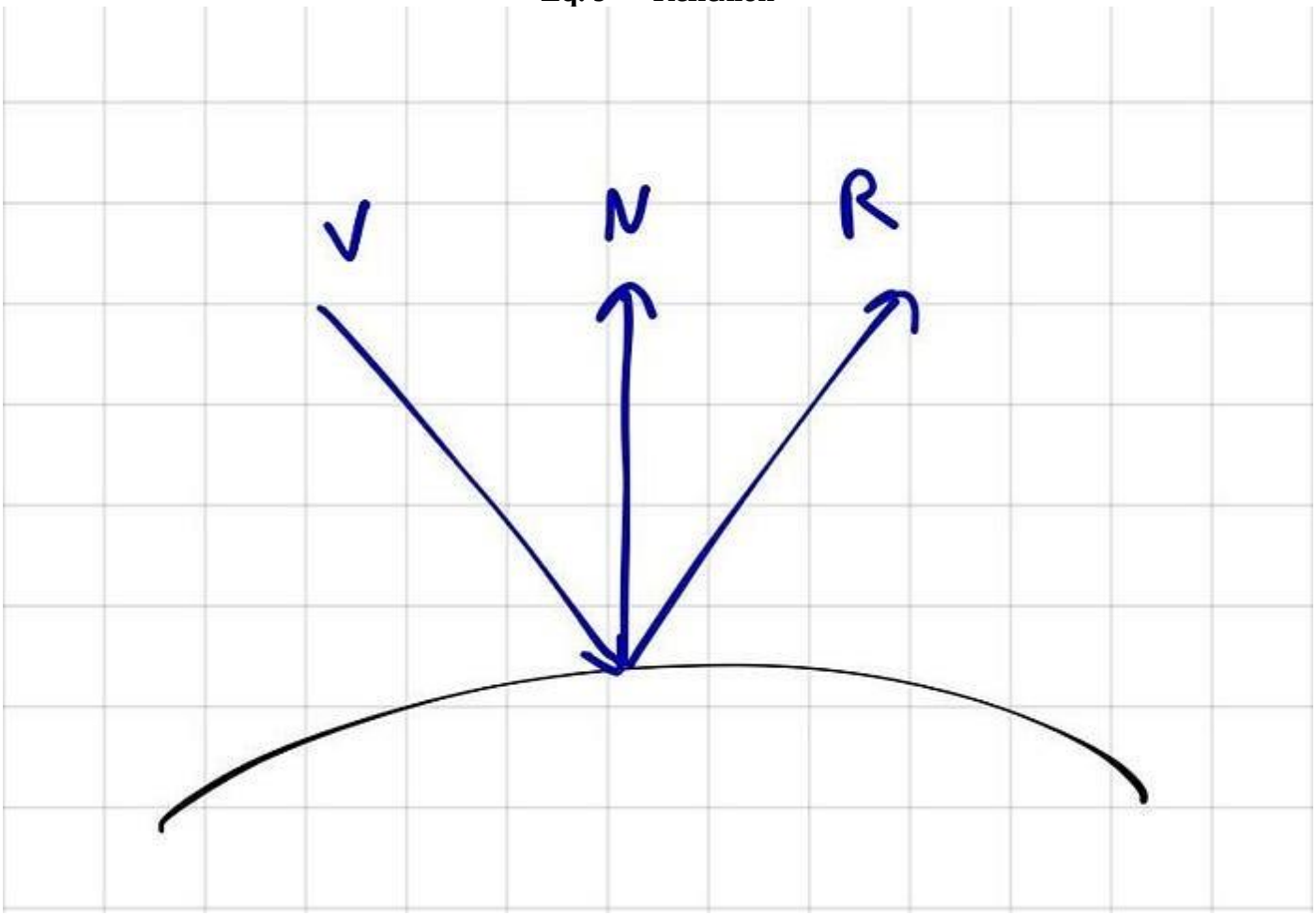


Figure. 7— Diagramme de réflexion

Où

1. R est le rayon réfléchi normalisé ;
2. V est un **vecteur unité** de direction du rayon à réfléchir ;
3. N est le **vecteur unité de direction** *perpendiculaire* à la surface de la course du rayon ;

Ajoutez cette méthode en haut du fichier avec la fonction `normalize()` :

```
def reflected(vector, axis):
    return vector - 2 * np.dot(vector, axis) * axis
```

Code. 11 — Rayon réfléchi

12°) Code

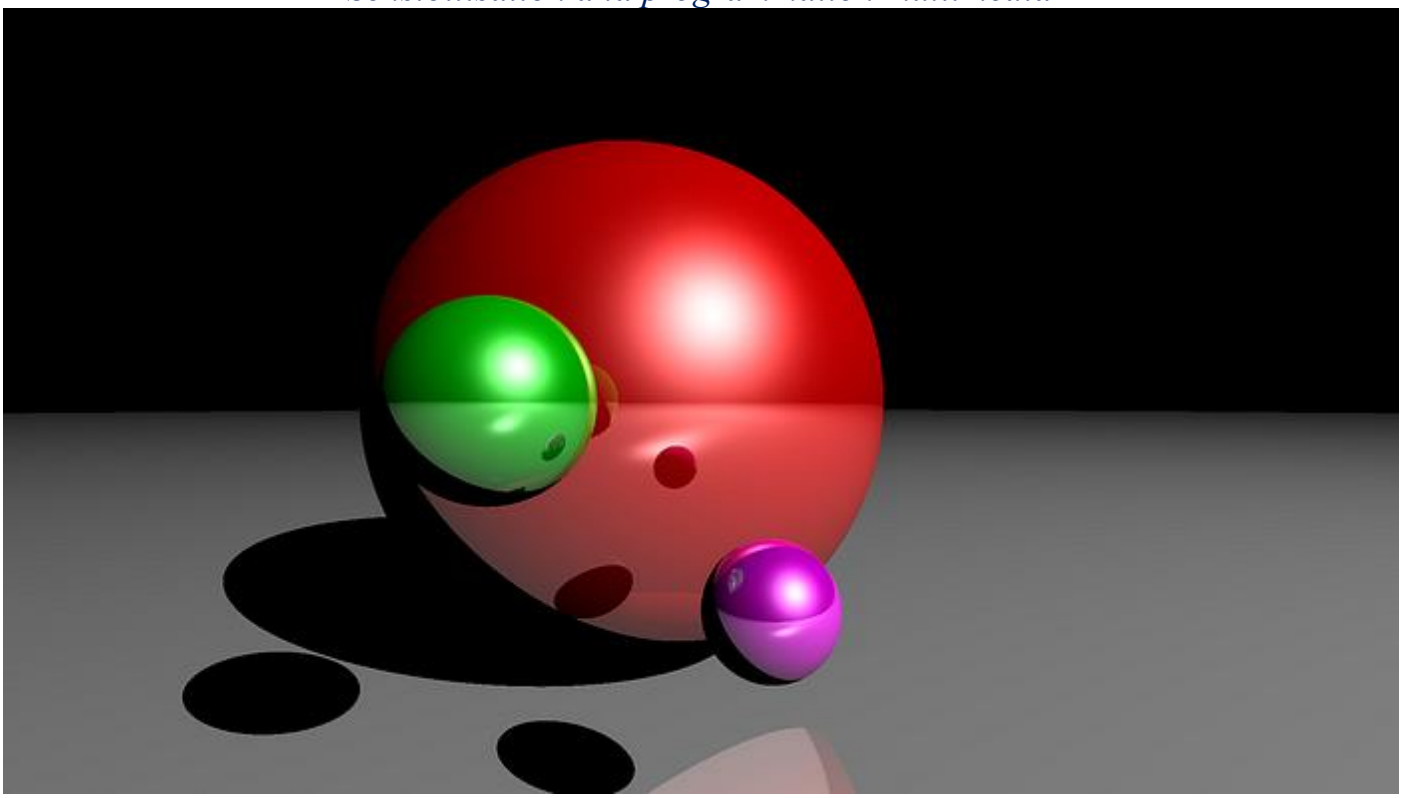
Il est temps de coder cela. C'est en fait un petit changement à la fin. Apportez simplement les modifications suivantes :

```
# global variable along with width, height, etc.
max_depth = 3

# everything that follows is inside the double for loop
color = np.zeros((3))
reflection = 1

for k in range(max_depth):
    nearest_object, min_distance = # ...
    # ...
    illumination += # ...

    # reflection
    color += reflection * illumination
    reflection *= nearest_object['reflection']

    # new ray origin and direction
    origin = shifted_point
    direction = reflected(direction, normal_to_surface)

image[i, j] = np.clip(color, 0, 1)
```

Code. 12 — Insertion du Rayon réfléchi

Important : *Maintenant que nous avons placé le code d'intersection dans une autre boucle de réflexion, nous devrions utiliser* `des instructions break` *là où nous utilisions auparavant des* `instructions continues, afin d'éviter des calculs inutiles.`

Voilà ! Exécutez le code et observez le beau résultat !



1. Final Code

Le code final est étonnamment petit, environ une centaine de lignes de code !

```
import numpy as np
import matplotlib.pyplot as plt

def normalize(vector):
    return vector / np.linalg.norm(vector)

def reflected(vector, axis):
    return vector - 2 * np.dot(vector, axis) * axis

def sphere_intersect(center, radius, ray_origin, ray_direction):
    b = 2 * np.dot(ray_direction, ray_origin - center)
    c = np.linalg.norm(ray_origin - center) ** 2 - radius ** 2
    delta = b ** 2 - 4 * c
    if delta > 0:
        t1 = (-b + np.sqrt(delta)) / 2
        t2 = (-b - np.sqrt(delta)) / 2
        if t1 > 0 and t2 > 0:
            return min(t1, t2)
    return None

def nearest_intersected_object(objects, ray_origin, ray_direction):
    distances = [sphere_intersect(obj['center'], obj['radius'], ray_origin,
ray_direction) for obj in objects]
    nearest_object = None
    min_distance = np.inf
    for index, distance in enumerate(distances):
        if distance and distance < min_distance:
            min_distance = distance
            nearest_object = objects[index]
    return nearest_object, min_distance

width = 300
height = 200

max_depth = 3
```

```
camera = np.array([0, 0, 1])
ratio = float(width) / height
screen = (-1, 1 / ratio, 1, -1 / ratio) # left, top, right, bottom

light = { 'position': np.array([5, 5, 5]), 'ambient': np.array([1, 1, 1]), 'diffuse':
np.array([1, 1, 1]), 'specular': np.array([1, 1, 1]) }

objects = [
    { 'center': np.array([-0.2, 0, -1]), 'radius': 0.7, 'ambient': np.array([0.1, 0,
0]), 'diffuse': np.array([0.7, 0, 0]), 'specular': np.array([1, 1, 1]), 'shininess':
100, 'reflection': 0.5 },
    { 'center': np.array([0.1, -0.3, 0]), 'radius': 0.1, 'ambient': np.array([0.1, 0,
0.1]), 'diffuse': np.array([0.7, 0, 0.7]), 'specular': np.array([1, 1, 1]),
'shininess': 100, 'reflection': 0.5 },
    { 'center': np.array([-0.3, 0, 0]), 'radius': 0.15, 'ambient': np.array([0, 0.1,
0]), 'diffuse': np.array([0, 0.6, 0]), 'specular': np.array([1, 1, 1]), 'shininess':
100, 'reflection': 0.5 },
    { 'center': np.array([0, -9000, 0]), 'radius': 9000 - 0.7, 'ambient':
np.array([0.1, 0.1, 0.1]), 'diffuse': np.array([0.6, 0.6, 0.6]), 'specular':
np.array([1, 1, 1]), 'shininess': 100, 'reflection': 0.5 }
]

image = np.zeros((height, width, 3))
for i, y in enumerate(np.linspace(screen[1], screen[3], height)):
    for j, x in enumerate(np.linspace(screen[0], screen[2], width)):
        # screen is on origin
        pixel = np.array([x, y, 0])
        origin = camera
        direction = normalize(pixel - origin)

        color = np.zeros((3))
        reflection = 1

        for k in range(max_depth):
            # check for intersections
            nearest_object, min_distance = nearest_intersected_object(objects, origin,
direction)
            if nearest_object is None:
                break

            intersection = origin + min_distance * direction
            normal_to_surface = normalize(intersection - nearest_object['center'])
            shifted_point = intersection + 1e-5 * normal_to_surface
            intersection_to_light = normalize(light['position'] - shifted_point)

            _, min_distance = nearest_intersected_object(objects, shifted_point,
intersection_to_light)
            intersection_to_light_distance = np.linalg.norm(light['position'] -
intersection)
            is_shadowed = min_distance < intersection_to_light_distance

            if is_shadowed:
                break

            illumination = np.zeros((3))

            # ambiant
            illumination += nearest_object['ambient'] * light['ambient']

            # diffuse
            illumination += nearest_object['diffuse'] * light['diffuse'] *
np.dot(intersection_to_light, normal_to_surface)

            # specular
            intersection_to_camera = normalize(camera - intersection)
            H = normalize(intersection_to_light + intersection_to_camera)
```

```
            illumination += nearest_object['specular'] * light['specular'] *
np.dot(normal_to_surface, H) ** (nearest_object['shininess'] / 4)

            # reflection
            color += reflection * illumination
            reflection *= nearest_object['reflection']

            origin = shifted_point
            direction = reflected(direction, normal_to_surface)

        image[i, j] = np.clip(color, 0, 1)
    print("%d/%d" % (i + 1, height))

plt.imsave('image.png', image)
```

1. Quelle est la prochaine étape ?

C'était un programme très simpliste qui était destiné à éduquer sur le sujet. Il y a tellement de façons d'améliorer cela et de mettre en œuvre d'autres fonctionnalités fascinantes. En voici quelques-unes :

1. Programmation orientée objet ! Pour l'instant, nous avons mis tous les objets dans un dict, mais vous pouvez créer des classes, déterminer ce qui est spécifique aux sphères et ce qui ne l'est pas, créer une classe de base et implémenter d'autres objets tels que des plans ou des triangles ;

2. Il en va de même pour la lumière. Ajoutez du POO ici et faites en sorte que vous puissiez ajouter plusieurs lumières dans la scène ;

3. Séparer les propriétés du matériau des propriétés géométriques, pour pouvoir appliquer un matériau (par exemple, bleu mat) à n'importe quel type d'objets ;

4. Trouvez un moyen de positionner correctement l'écran en fonction de la position de la caméra et d'une direction à regarder ;

5. Modélisez la lumière différemment. Actuellement, il s'agit d'un point unique, c'est pourquoi les ombres des objets sont « dures » ou bien définies. Afin d'obtenir des ombres « douces » (avec un dégradé en gros), vous devez modéliser une lumière comme un objet 2D ou 3D : disque ou sphère ?

13°) Bonus

Voici une animation que j'ai faite avec le ray tracing. J'ai simplement rendu la scène plusieurs fois avec la caméra à différentes positions.

Le code est en Kotlin (vous remarquerez alors à quel point python est lent...) et disponible sur GitHub si vous êtes intéressé.

OmarAflak/RayTracer-Kotlin

- *Simple traceur de rayons. Contribuez au développement d'OmarAflak/RayTracer-Kotlin en créant un compte sur GitHub.*
- *github.com*

14°) Conclusion

Félicitations si vous avez été jusque-là ! J'espère que vous avez apprécié ce sujet fascinant. Pour d'autres lectures à ce sujet, je conseillerais fortement le site Web suivant :

15°) Scratchapixel

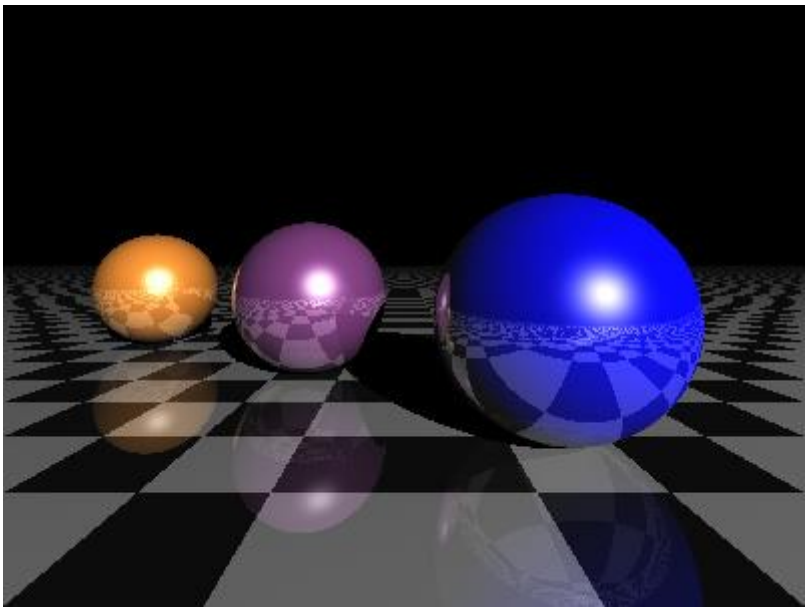
Apprenez l'infographie à partir de zéro ! « Vous n'avez aucune idée à quel point ils sont utiles. Les explications détaillées m'aident à ralentir...

- www.scratchapixel.com

À votre santé !

D. Traceur de rayons Python raisonnablement rapide

Le petit ray-tracer de Cyrille Rossant est un joli programme Python autonome (utilisant NumPy) qui rend cette image 400 × 300 en environ 15 secondes sur un PC rapide :



Vous pourriez en conclure que Python est un langage inapproprié pour un traceur de rayons. Mais cette version est légèrement plus petite et rend la même image en environ 115 *millisecondes*. C'est plus de 130 fois plus rapide que l'original.

Les deux versions utilisent NumPy, les deux fonctionnent sur un seul cœur. La différence réside dans la façon dont ils organisent leur calcul.

Le code original ressemble un peu à ceci :

```
for x in range(400):
    for y in range(300):
        pixel = raytrace(x, y)
write pixels to file
```

Et la version rapide ressemble plus à ceci :

```
x = <every pixel's x-coordinate from 0 to 400>
y = <every pixel's y-coordinate from 0 to 300>
pixels = raytrace(x, y)
write pixels to file
```

Dans la première version, x et y sont des valeurs scalaires, et `raytrace()` s'exécute 120 000 fois. Mais dans la deuxième version, x et y sont des tableaux NumPy, chacun de 120 000 valeurs de long, et `raytrace()` s'exécute une fois. Parce que l'exécution du code Python est assez lente et que les opérations de tableau de NumPy sont *très* rapides, l'accélération est énorme.

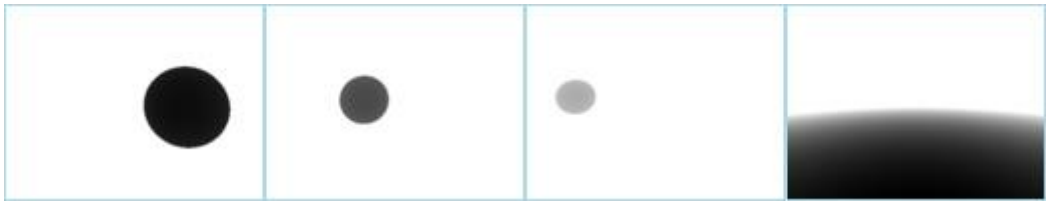
NumPy vous permet d'opérer sur des tableaux sans syntaxe spéciale, de sorte que la définition réelle de `raytrace()` n'a pas besoin d'être beaucoup préoccupée par le fait qu'il fonctionne sur des tableaux au lieu d'un scalaire. Par exemple, le code pour calculer une intersection de sphère ressemble à la version scalaire :

```
def intersect(self, O, D):
    a = 1
```

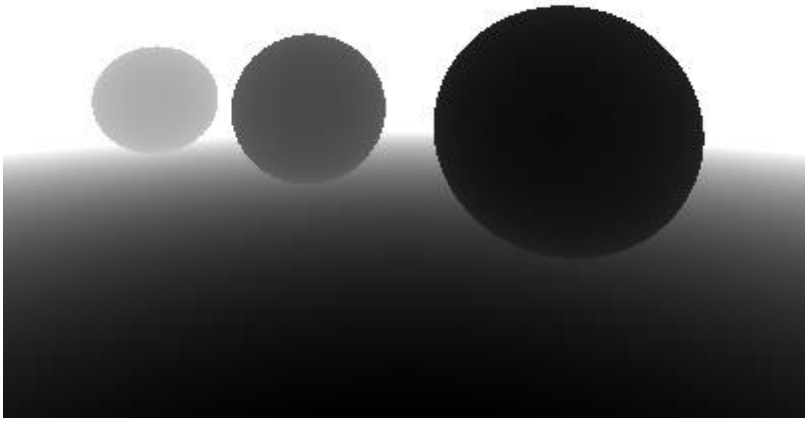
```
    b = 2 * D.dot(O - self.c)
    c = abs(self.c) + abs(O) - 2 * self.c.dot(O) - (self.r * self.r)
    disc = (b ** 2) - (4 * a * c)
    sq = np.sqrt(np.maximum(0, disc))
    h0 = (-b - sq) / 2
    h1 = (-b + sq) / 2
    h = np.where((h0 > 0) & (h0 < h1), h0, h1)

    hit = (disc > 0) & (h > 0)
    return np.where(hit, h, FARAWAY)
```

La seule différence entre ce code et le code scalaire est qu'il utilise `np.where()` au lieu des instructions `if`. En effet, chaque expression est en fait un tableau, de sorte qu'une instruction `if` n'effectuerait pas la sélection *par élément* requise.

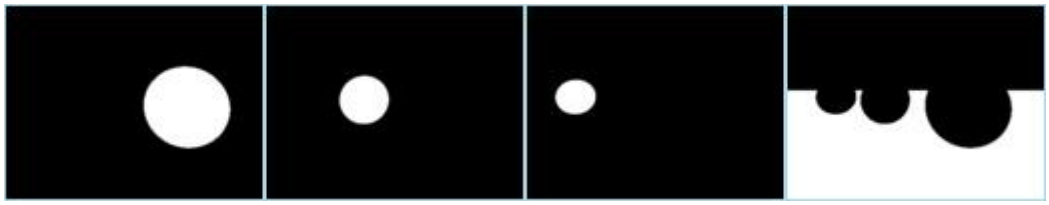
En commençant par les rayons générés, le code exécute d'abord `intersect()` sur chacun des objets de la scène, pour connaître leur distance par rapport à la caméra. Ici, plus clair signifie le plus loin :



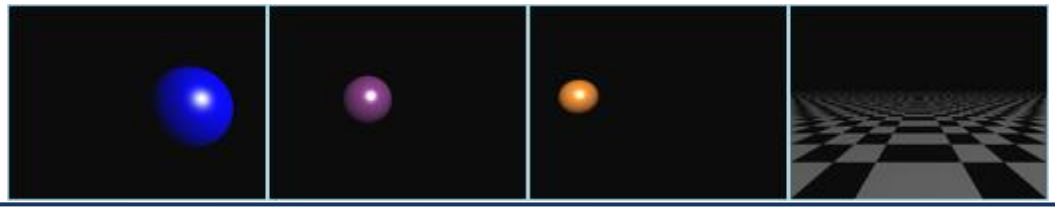
En prenant le minimum de ces distances, on obtient le « hit » le plus proche pour chaque pixel :



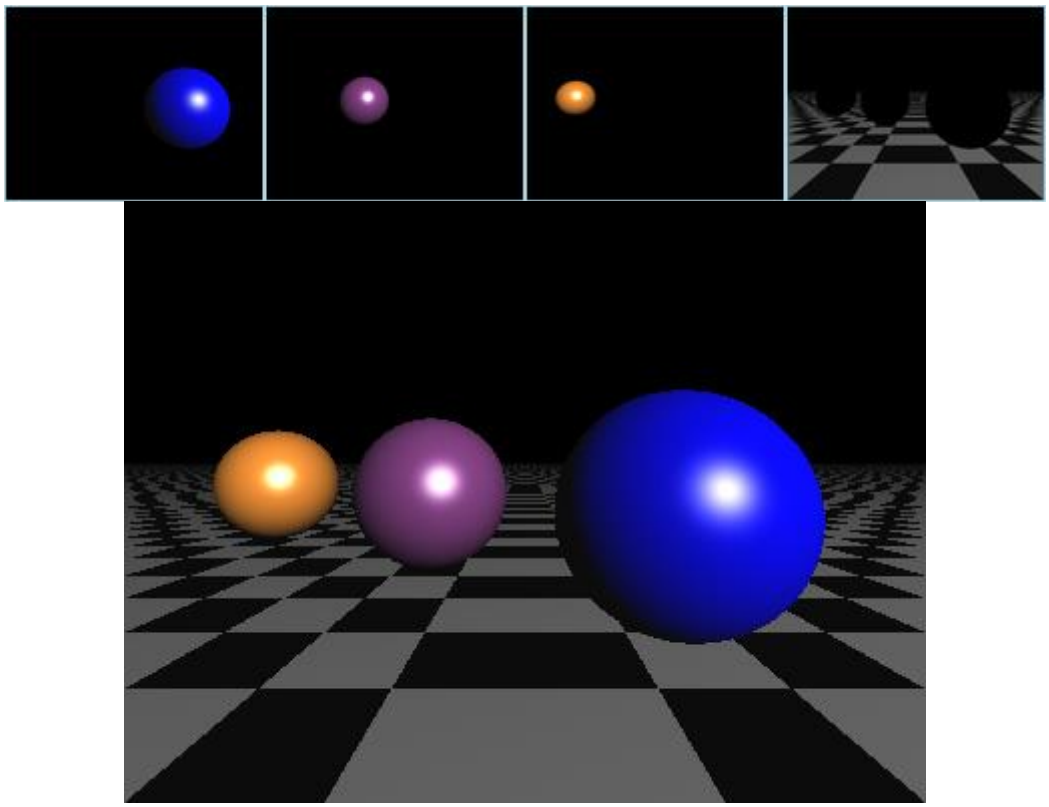
La comparaison de la distance de chaque objet par rapport à la valeur « la plus proche » donne quatre masques. Un pixel blanc dans le masque signifie que l'objet a « gagné » le concours de visibilité.



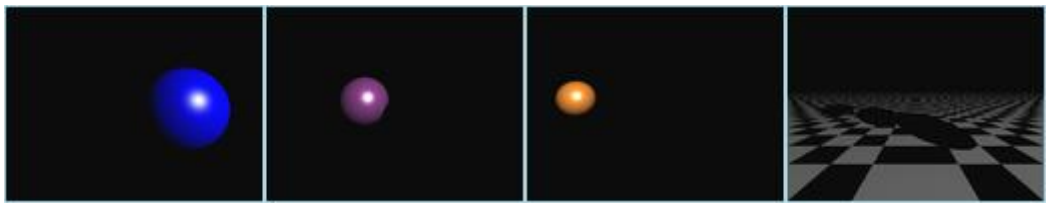
L'étape suivante consiste à exécuter la fonction shader de chaque objet, pour savoir quelle est sa couleur. Ceci est fait pour chaque pixel à l'écran, rendant efficacement chaque objet comme une image séparée.



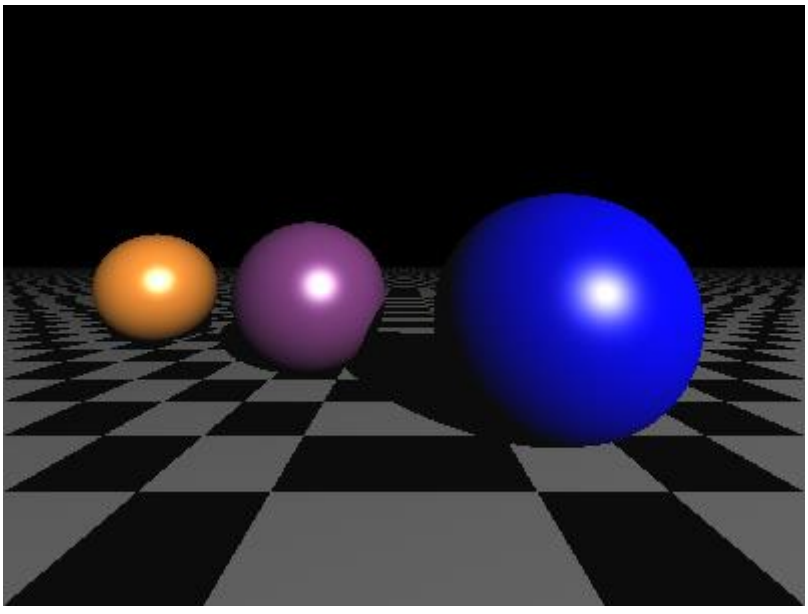
Maintenant, `raytrace()` multiplie chaque image couleur avec son masque et additionne les quatre images résultantes. Étant donné que les masques déterminent quelle image est la plus « proche » pour un pixel particulier, l'ordre n'a pas d'importance. Cela donne à ce composite, un trace-rayon de premier rebond très basique :



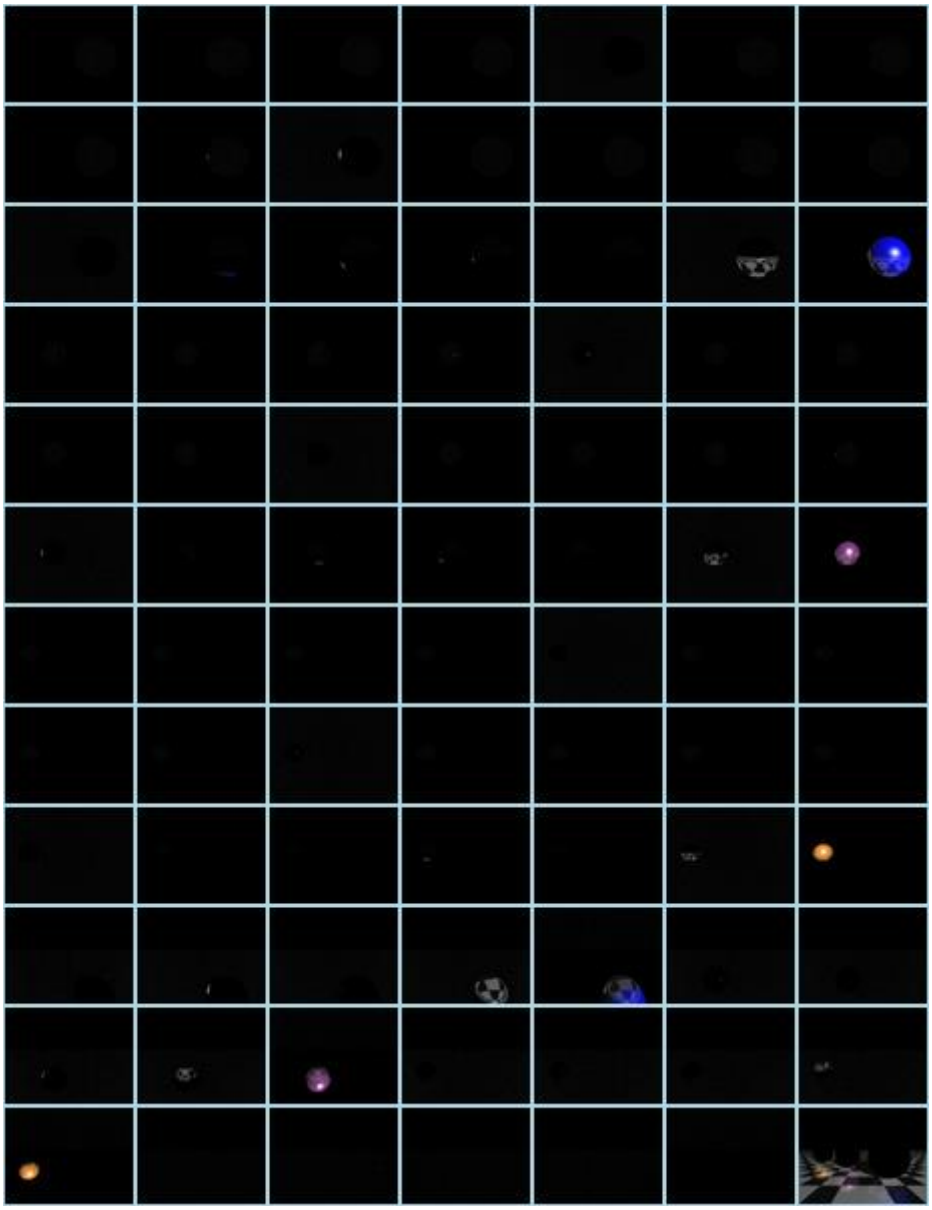
À partir de là, il y a quelques améliorations, toutes impliquant une fonction d'éclairage légèrement plus sophistiquée. Les ombres sont un repère visuel important, et la mise en œuvre consiste simplement à tester chaque pixel pour voir s'il peut « voir » la lumière. Si le pixel peut voir la lumière, il obtient une luminosité maximale, sinon il n'en obtient aucune :



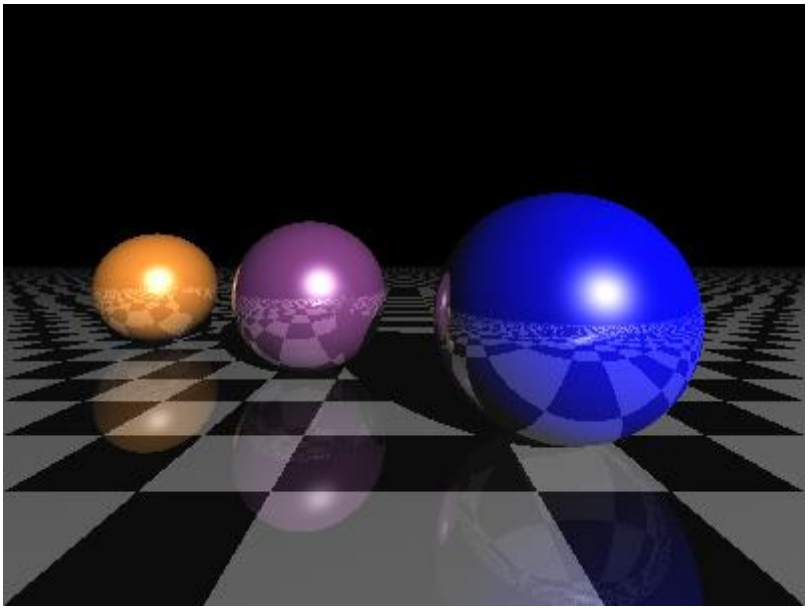
Ensemble, les choses semblent légèrement plus convaincantes :



L'étape suivante est les réflexions, qui sont un raytrace récursif utilisant le rayon rebondi de chaque pixel. Cela implique beaucoup de passes supplémentaires ; La fonction `raytrace()` finit par être appelée 84 fois avec différentes combinaisons de rebonds secondaires :

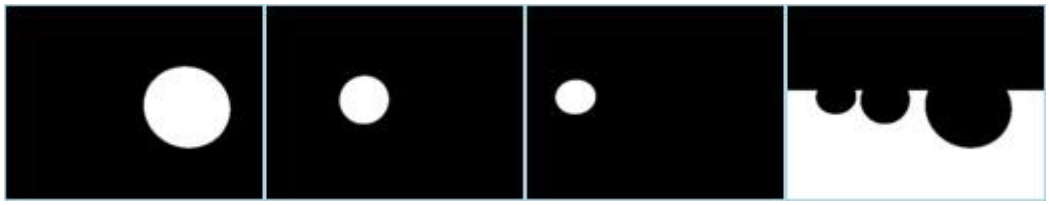


Le résultat correspond assez bien à l'original :



Malgré le calcul et la composition de 84 images en lancer de rayons, cette version fonctionne en environ 4 secondes, soit un quart du temps de l'original. Cette version est rt2.py.

Une petite optimisation de `raytrace()` est encore plus rapide, exploitant le fait que le shader de n'importe quel objet n'a besoin de fonctionner que sur les pixels visibles de cet objet. En utilisant les masques calculés pour les tests d'intersection :



et les fonctions `Numpy extract`() et `place(), raytrace()` extrait tous les pixels noirs inutilisés du masque de chaque objet avant d'exécuter le shader de l'objet. Donc, si l'objet ne couvre que 3000 pixels à l'écran, alors les tableaux NumPy introduits dans le shader ne font que 3000 pixels, au lieu de 120 000. Parce que la plupart des objets ne couvrent pas beaucoup de surface d'écran, l'accélération est énorme, environ 40X.

Cette version est rt3.py.

E. Webographie :

- https://excamera.com/sphinx/article-ray.html
- https://github.com/jamesbowman/raytrace
- https://github.com/rafael-fuente/Python-Raytracer
- https://github.com/jrmiranda/raytracer
- https://medium.com/swlh/ray-tracing-from-scratch-in-python-41670e6a96f9
- https://xboxsquad.fr/le-ray-tracing-que-la-lumiere-soit/
- https://www.numerama.com/tech/412041-tout-comprendre-au-ray-tracing-les-moteurs-de-jeu-video-ont-ils-trouve-leur-graal.html
- https://en.wikipedia.org/wiki/Luxo_Jr._(character)
- https://www.frandroid.com/dossiers/660693_dossier-ray-tracing-explication
- https://physique.cmaisonneuve.qc.ca/svezina/projet/ray_tracer.html
- https://www.cs.ucy.ac.cy/courses/EPL426/courses/Lectures/12_Polygon_Rendering_Methods_EN.pdf