iNFO

IUT GRAND OUEST NORMANDIE

R 5.A.06

2025 – 2026

Sensibilisation à la programmation multimédia

TP n° 4
Raytracing

ANNE Jean-François

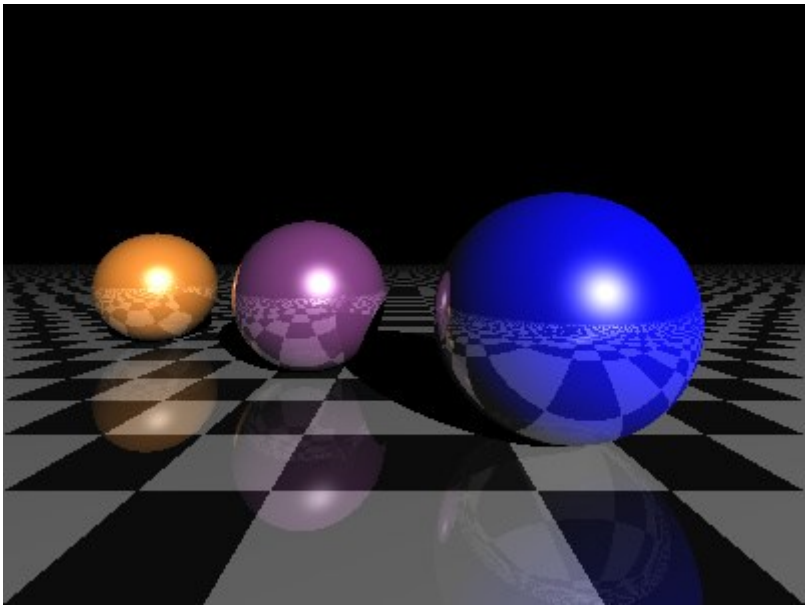
Sensibilisation à la programmation multimédia

Programmation en 3D

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

A. *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.

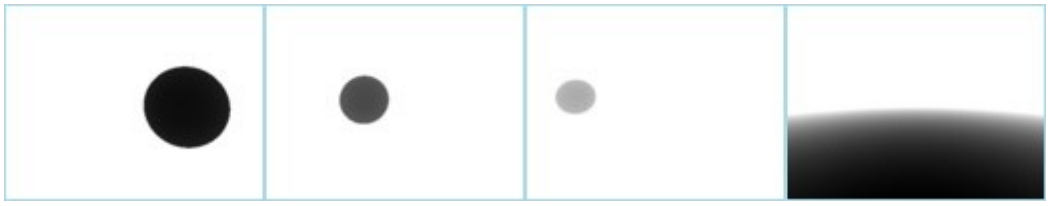
NumPy vous permet d'opérer sur des tableaux sans syntaxe spéciale, de sorte que la définition réelle de `raytrace()` n'a pas besoin d'être beaucoup préoccupée par le fait qu'il fonctionne sur des tableaux au lieu d'un scalaire. Par exemple, le code pour calculer une intersection de sphère ressemble à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

```
def intersect(self, O, D):
    a = 1
    b = 2 * D.dot(O - self.c)
```

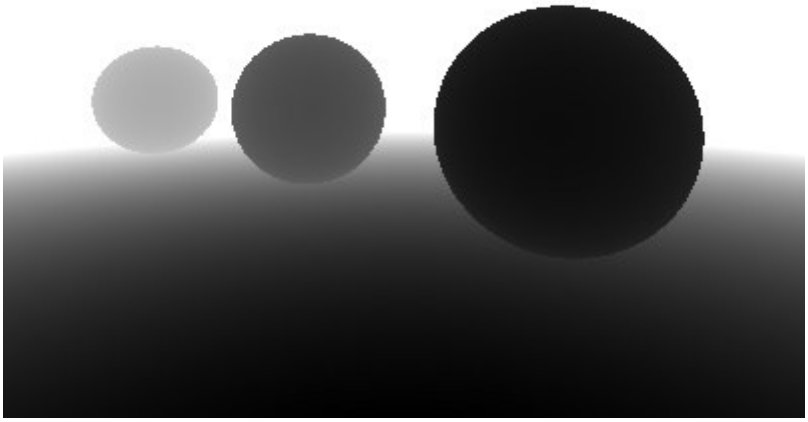
```
        c = abs(self.c) + abs(O) - 2 * self.c.dot(O) - (self.r * self.r)
        disc = (b ** 2) - (4 * a * c)
        sq = np.sqrt(np.maximum(0, disc))
        h0 = (-b - sq) / 2
        h1 = (-b + sq) / 2
        h = np.where((h0 > 0) & (h0 < h1), h0, h1)

        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 :

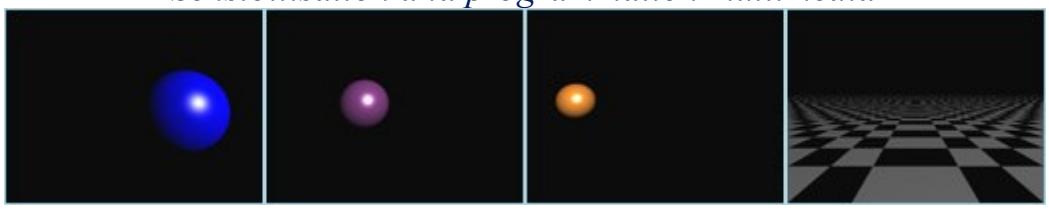


En prenant le minimum de ces distances, on obtient le « hit » le plus proche pour chaque pixel :

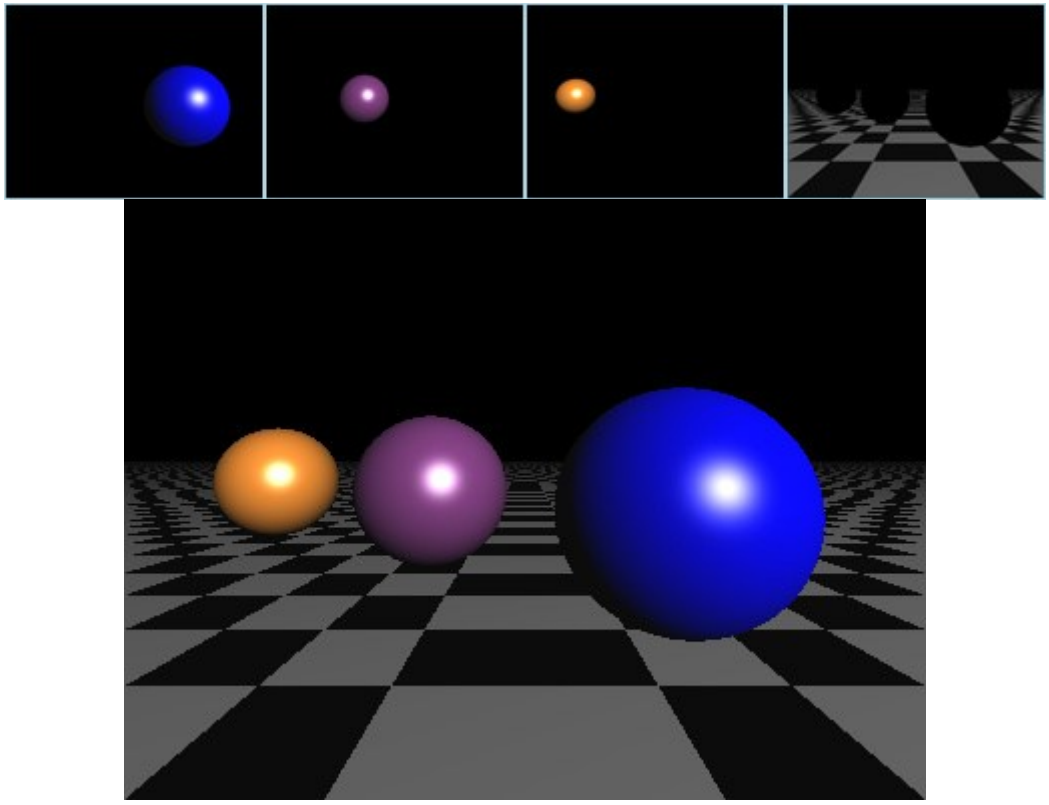


La comparaison de la distance de chaque objet par rapport à la valeur « la plus proche » donne quatre masques. Un pixel blanc dans le masque signifie que l'objet a « gagné » le concours de visibilité.

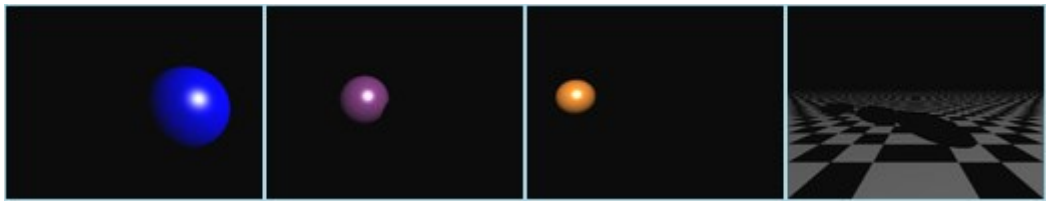
L'étape suivante consiste à exécuter la fonction shader de chaque objet, pour savoir quelle est sa couleur. Ceci est fait pour chaque pixel à l'écran, rendant efficacement chaque objet comme une image séparée.



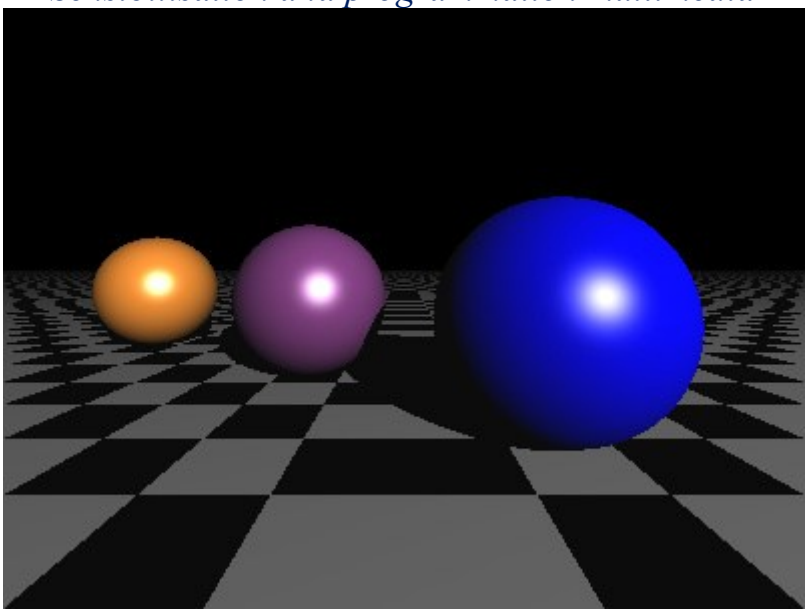
Maintenant, `raytrace()` multiplie chaque image couleur avec son masque et additionne les quatre images résultantes. Étant donné que les masques déterminent quelle image est la plus « proche » pour un pixel particulier, l'ordre n'a pas d'importance. Cela donne à ce composite, un trace-rayon de premier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



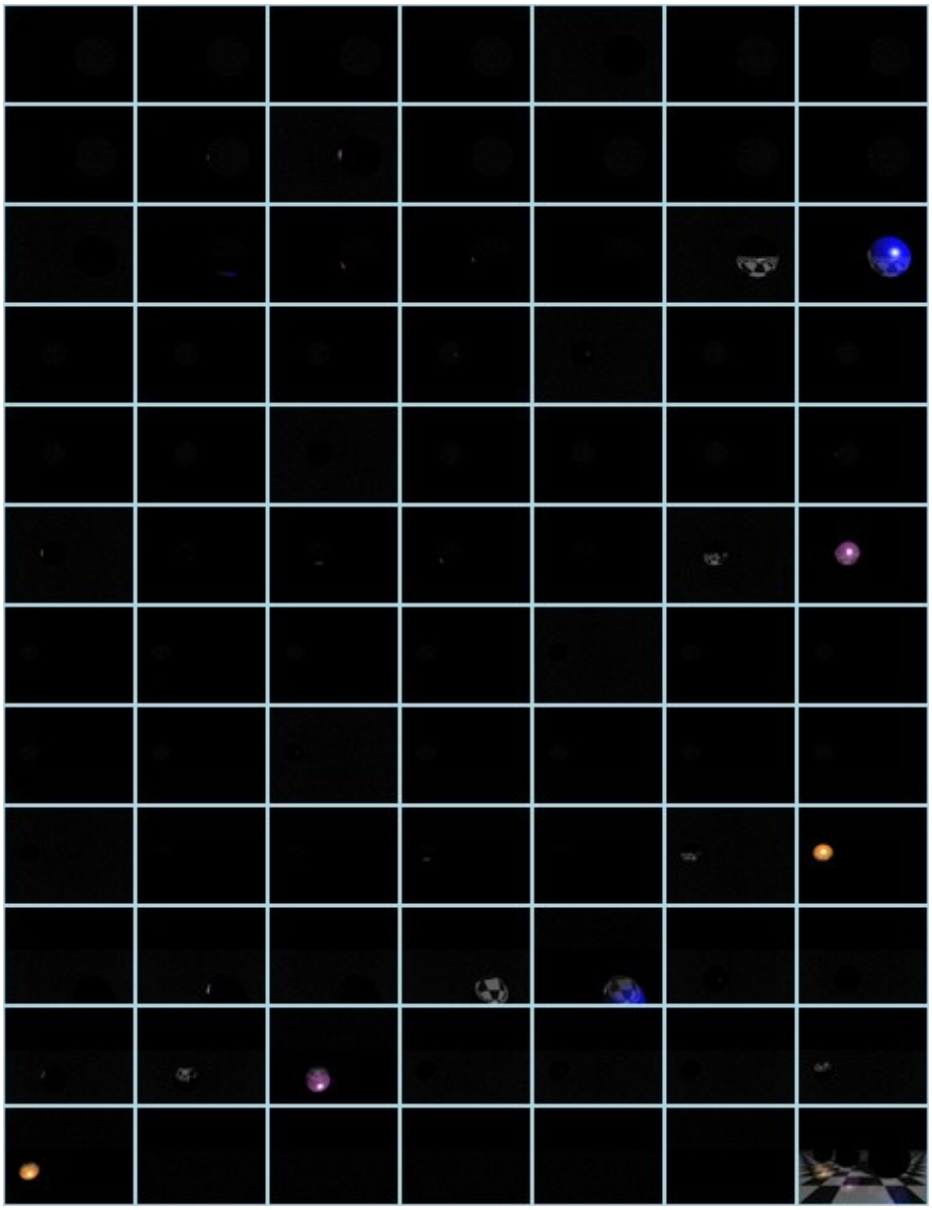
À partir de là, il y a quelques améliorations, toutes impliquant une fonction d'éclairage légèrement plus sophistiquée. Les ombres sont un repère visuel important, et la mise en œuvre consiste simplement à tester chaque pixel pour voir s'il peut « voir » la lumière. Si le pixel peut voir la lumière,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



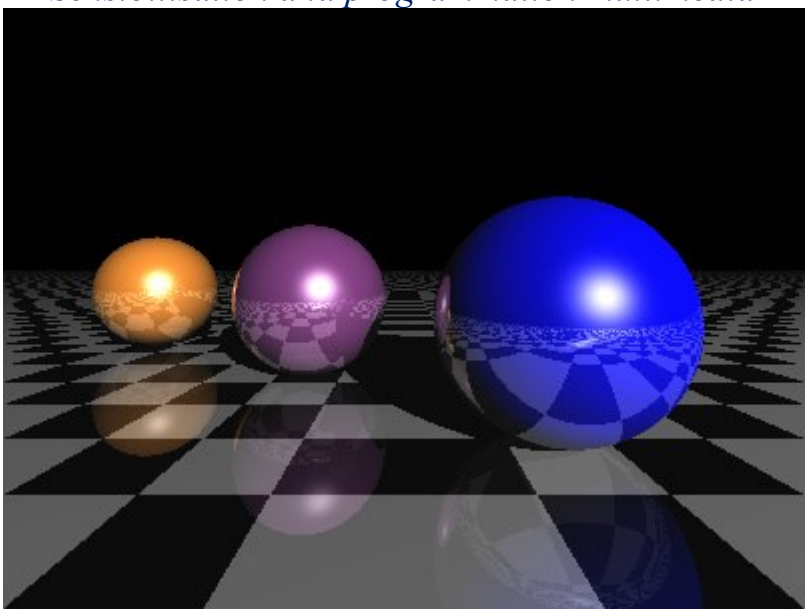
Ensemble, les choses semblent légèrement plus convaincantes :



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.

B. 2ème raytracing :

Tester les programmes de Rafael de la Fuente :

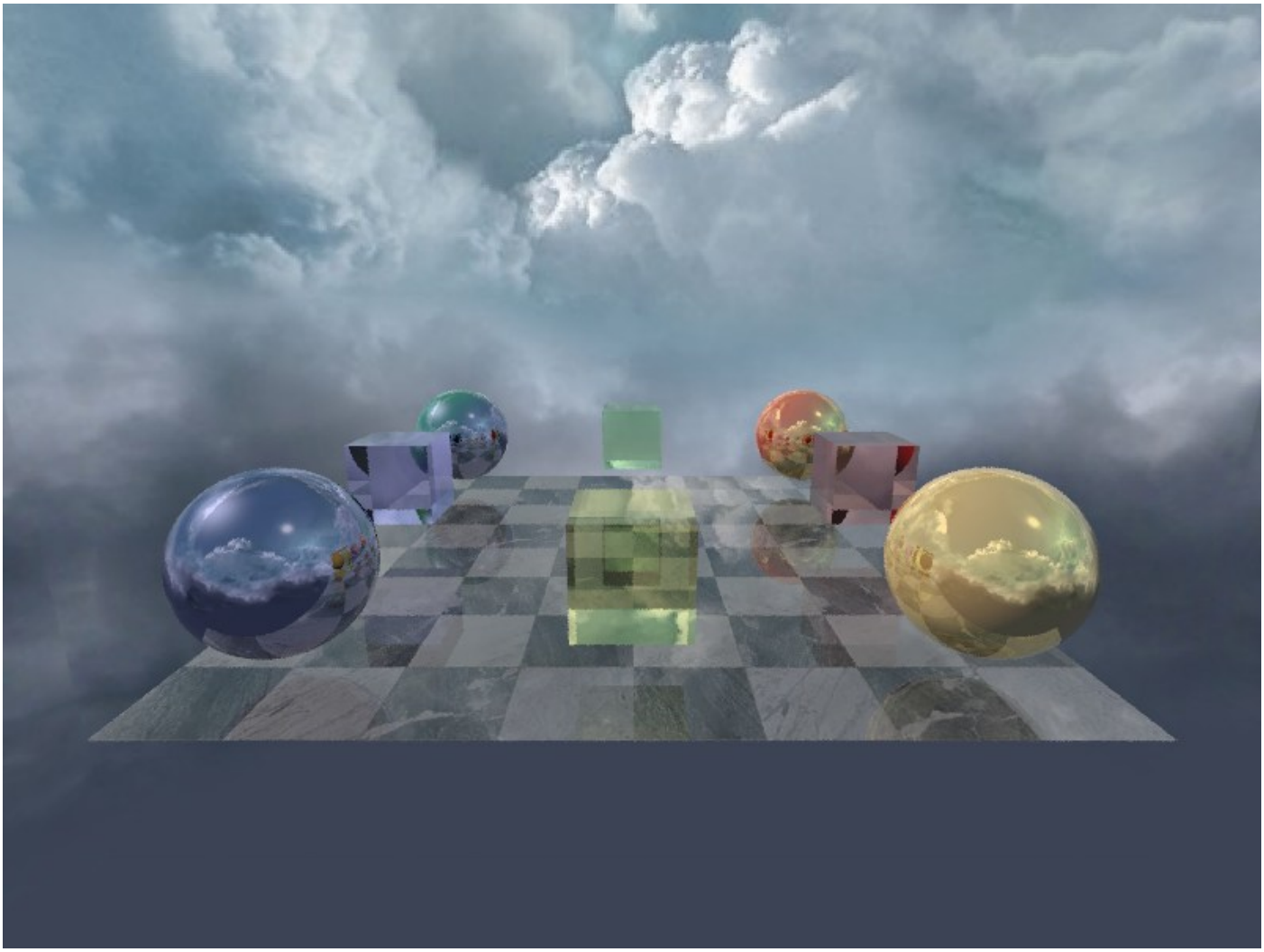
- https://github.com/rafael-fuente/Python-Raytracer

C. Synthèse :

1°) Exercice 1 :

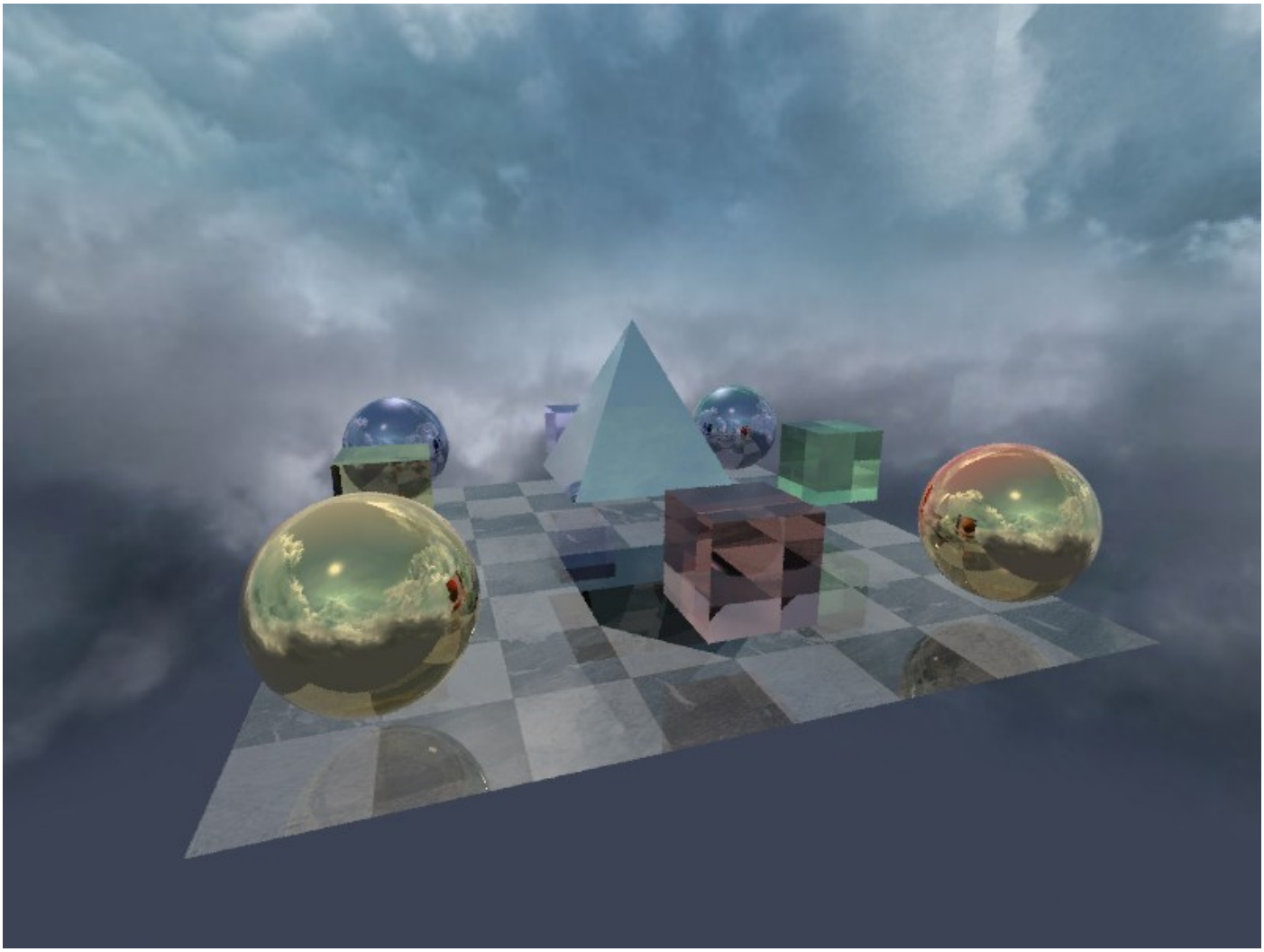
A partir des exemples fournis sur le site de Rafael, créer la scène suivante :

- *Une sphère de couleurs métalliques différentes à chaque coin du plan de base,*
- *Un cube de couleurs transparentes différentes sur les côtés du plan.*



2°) ***Exercice 2 :***

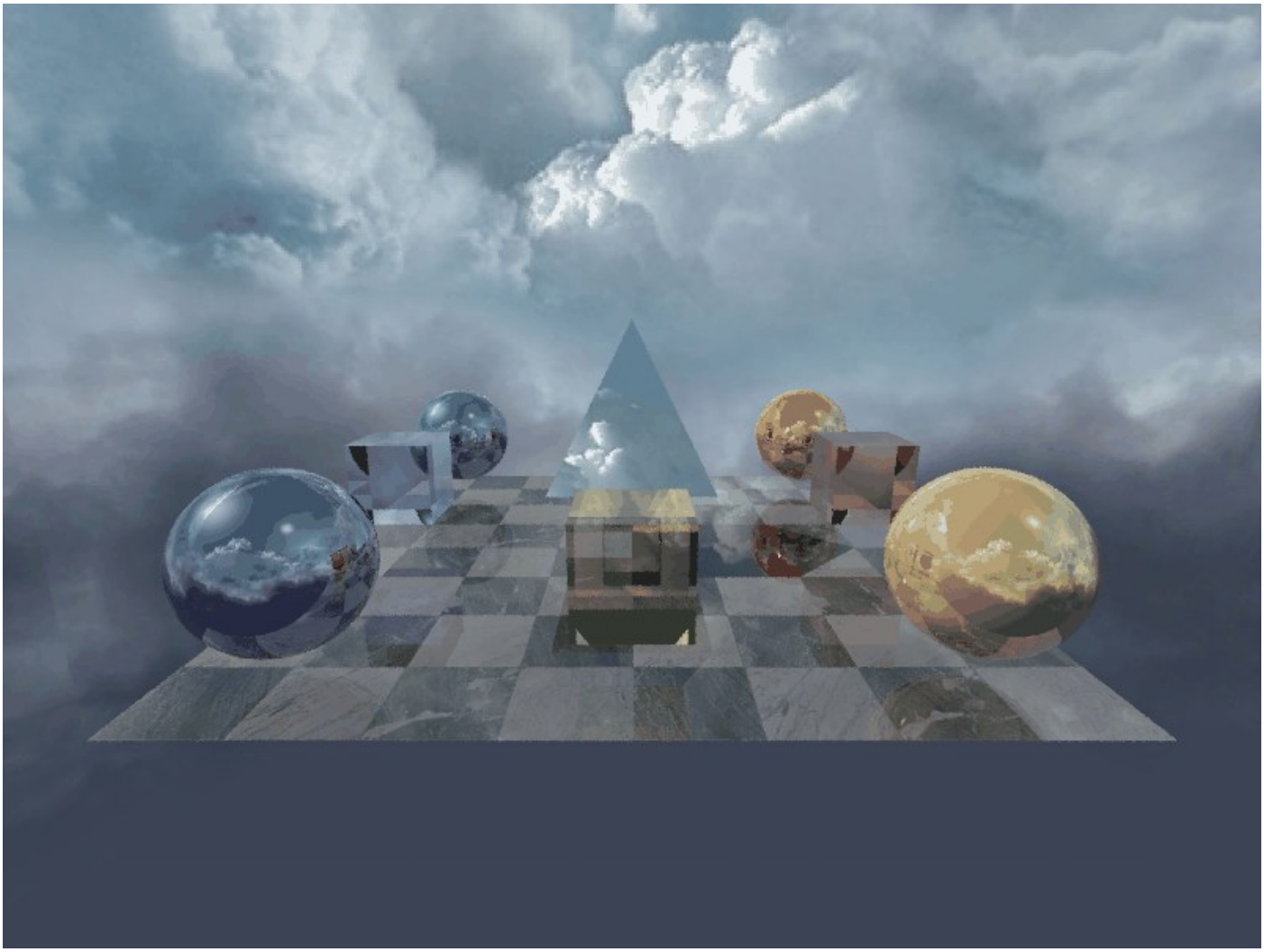
Rajouter un triangle de couleur métallique argent au milieu de la scène précédente :



Merci à clément BARATIN pour son aide pour la correction du code du triangle.

3°) Exercice 3 :

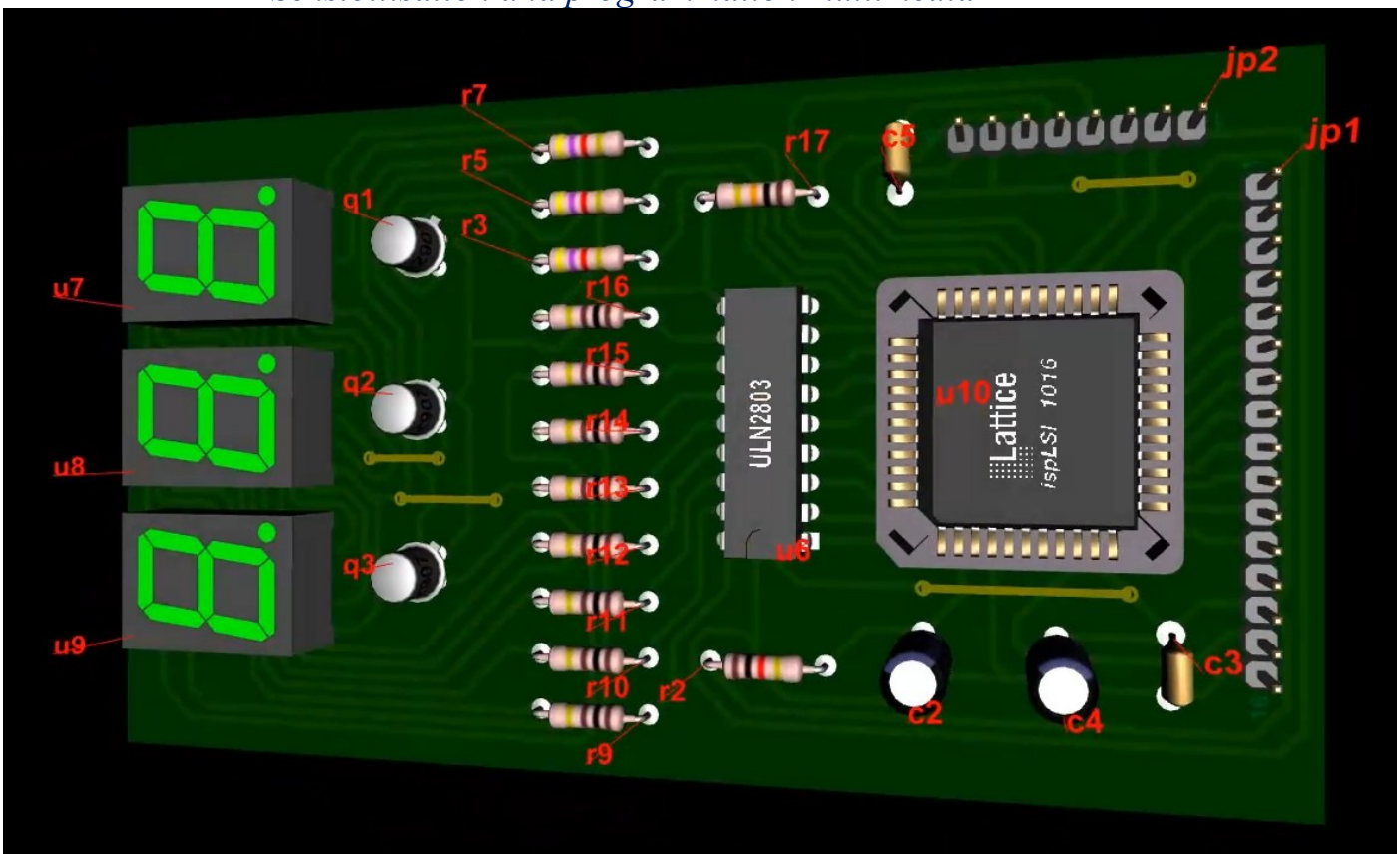
A partir de la scène précédente, réaliser une animation panoramique sur 360° en .gif :



D. Aller plus loin :

Si vous voulez aller plus en avant dans le Raytracing, je vous conseille le logiciel POV-Ray qui permet de faire de belles images :

- http://www.povray.org/
- https://hof.povray.org/



Carte réalisée avec POV Ray, JFA 2008

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

N'oubliez pas le NETTOYAGE et la reconfiguration à l'état d'origine des machines en fin de séance