P2. Engadir ao motor unha nova primitiva gráfica coa que traballar: triángulo. O código orixinal deste raytracer unicamente emprega esferas ou planos para construír unha escena e iluminala. Tedes que engadir a posibilidade de ter tamén triángulos na escena.

Modifica a escena predeterminada para que inclúa algún triángulo, ademais de esferas e planos como a escena orixinal.

La solución se ha dividido en varias partes:

1) Creación de **intersect_triangle**:

```
def intersect_triangle(O, D, V0, V1, V2):
   epsilon = 1e-8

   e1 = np.array(V1) - np.array(V0)
   e2 = np.array(V2) – np.array(V0)


   P = np.cross(D, e2)
   det = np.dot(e1, P)


 if det > -epsilon and det < epsilon:
  return np.inf

 inv_det = 1.0 / det

 T = O - V0

 u = np.dot(T, P) * inv_det
 if u < 0.0 or u > 1.0:
  return np.inf

 Q = np.cross(T, e1)

 v = np.dot(D, Q) * inv_det
 if v < 0.0 or u + v > 1.0:
  return np.inf


 t = np.dot(e2, Q) * inv_det

 if t > epsilon:
      return t


   return np.inf
```

Para la creación de la función que permite la intersección del triángulo en el plano se ha utilizado el algoritmo de **Möller-Trumbore** (Möller–Trumbore intersection algorithm – Wikipedia).

En **intersect_triangle** se verifica si el rayo se ha cruzado con el triángulo, calcula los vectores de los bordes y el determinante para ver si el rayo se encuentra en el plano del triángulo, en caso contrario no habrá intersección. Se calcula los parámetros **u** y **v** para ver si el punto de intersección está dentro del triángulo, en caso contrario no existirá ninguna intersección en el triángulo. Finalmente se calcula la distancia desde el origen del rayo hasta el punto de intersección.

2) Adaptación de **intersect**:

```
def intersect(O, D, obj):
    if obj['type'] == 'plane':
        return intersect_plane(O, D, obj['position'], obj['normal'])
    elif obj['type'] == 'sphere':
        return intersect_sphere(O, D, obj['position'], obj['radius'])
    elif obj['type'] == 'triangle':
        return intersect_triangle(O, D, obj['vertices'][0], obj['vertices'][1], obj['vertices'][2])
```

Simplemente se añade el tipo del triángulo y se pasan sus parámetros.

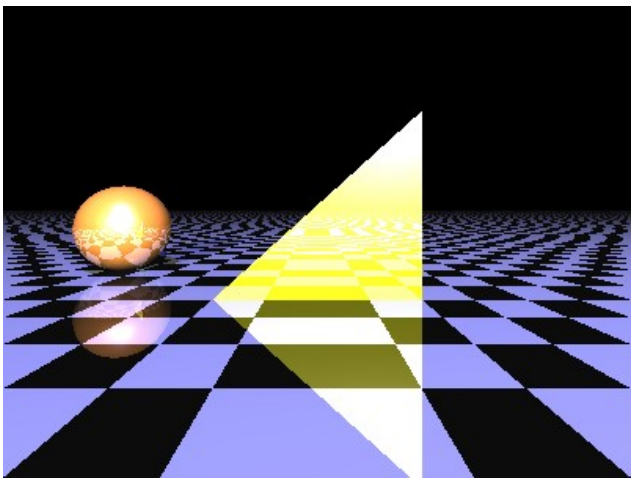
3) Permitir obtener la normal del triángulo:

```
def get_normal(obj, M):
    # Find normal.
    if obj['type'] == 'sphere':
        N = normalize(M - obj['position'])
    elif obj['type'] == 'plane':
        N = obj['normal']
    #Parte 3
    elif obj['type'] == 'triangle':
        v0, v1, v2 = obj['vertices']
        N = np.cross(v1 - v0, v2 - v0)
        N = normalize(N)
    return N
```

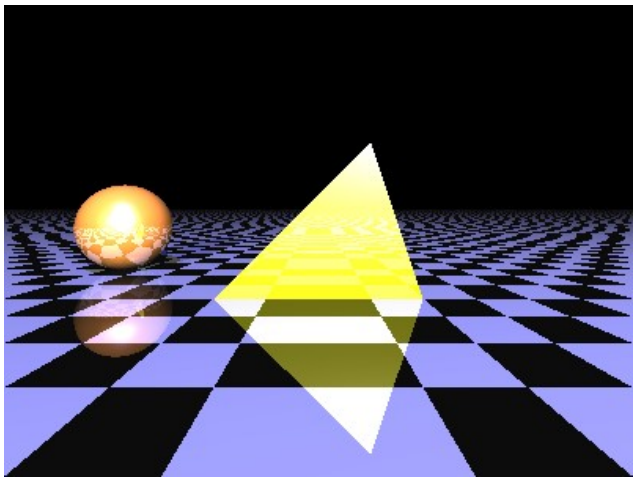
Debido a que para calcular la iluminación y el renderizado en 3D del triángulo es necesario disponer de la normal del triángulo, se ha calculado como el producto en cruz de los lados, lo que da un vector perpendicular al plano del triángulo y se normaliza a longitud 1.

4) Implementar la función **add_triangle**.

```
def add_triangle(v0, v1, v2, color):
    return dict(
        type='triangle',
        vertices=[np.array(v0), np.array(v1), np.array(v2)],
        color=np.array(color),
        reflection=0.3
    )
```



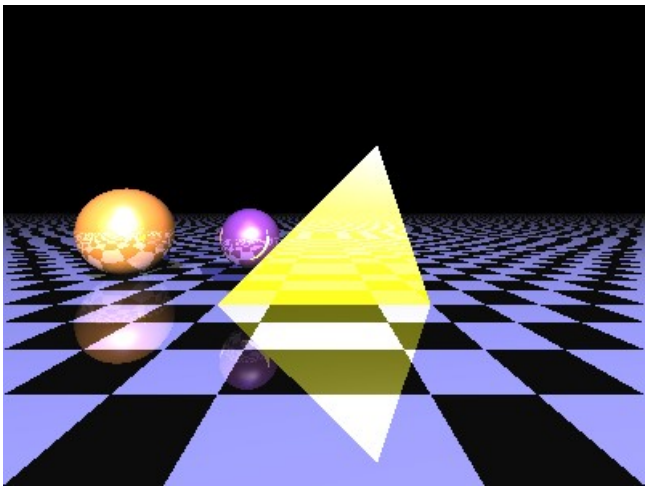
P2-1



P2-2

Para probar el correcto funcionamiento del triángulo se han generado dos triángulos diferentes, **P2-1** con reflexión a 0.5 y **P2-2** con reflexión a 0.3, se han eliminado las dos primeras esferas para evitar problemas de colisión de objetos (se pueden añadir todas las deseadas teniendo en cuenta que los objetos no entren unos dentro de otros).

Por ejemplo:



P3-2