**UNIVERSIDAD NACIONAL DE SAN ANTONIO ABAD DEL CUSCO**

**CARRERA PROFESIONAL DE INGENIERIA INFORMATICA Y DE SISTEMAS**

**COMPUTACION GRAFICA 2**

DOCENTE: M.SC. HECTOR E. UGARTE R.

---

## HOJA DE EJERCICIOS 3: RAYTRACING

### 1. COMPETENCIAS

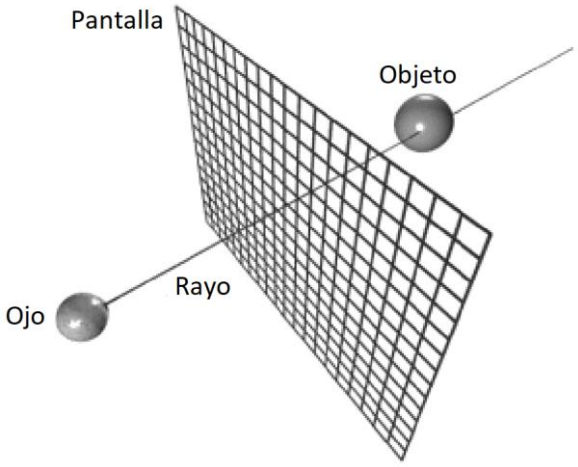
- El estudiante conoce la técnica del trazado de rayos (RayTracing).
- El estudiante utiliza raytracing para formas básicas.

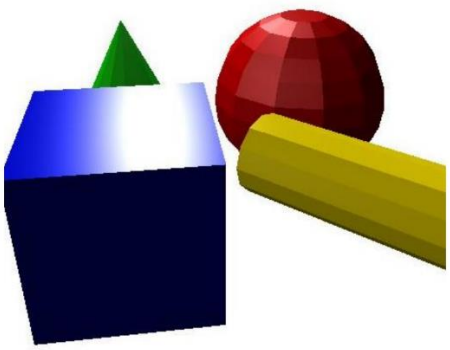
### 2. MARCO TEORICO

El trazado de rayos o raytracing es un algoritmo para la síntesis de imágenes bidimensionales que representan de forma físicamente realista una escena determinada. Se basa en el trazado de rayos y la intersección de estos con las superficies que contiene la escena que se quiere representar. Es un algoritmo cuya principal característica es intentar representar de forma fiel el comportamiento de la luz en la vida real. Para ello, es habitual que la generación de imágenes mediante el algoritmo de raytracing suela requerir un nivel intenso de computación.

El punto de partida del raytracing computacional se produce en 1968 con la definición del algoritmo de raycasting por parte de Arthur Appel. El raycasting era explicado como un algoritmo para obtener imágenes de sólidos a partir del trazado de rayos. Los rayos se lanzaban desde el observador y pasaban por un plano perpendicular situado justo delante del él. Este plano era en realidad una cuadrícula en la que cada una de sus posiciones representaba un píxel de la imagen resultante.

El rayo lanzado pasaba a través de uno de los elementos de la cuadrícula, es decir, por un píxel de la imagen resultante. Sólo quedaba determinar el color de este píxel dependiendo de si el sólido que se quería representar se encontraba en medio del camino que el rayo realizaba. En caso afirmativo, el color del sólido pasaba a ser el color del rayo y a su vez el color del píxel de la imagen generada. De esta forma y repitiendo el trabajo para cada uno de los elementos de la cuadrícula o píxel se obtenía una imagen completa con la representación del sólido.

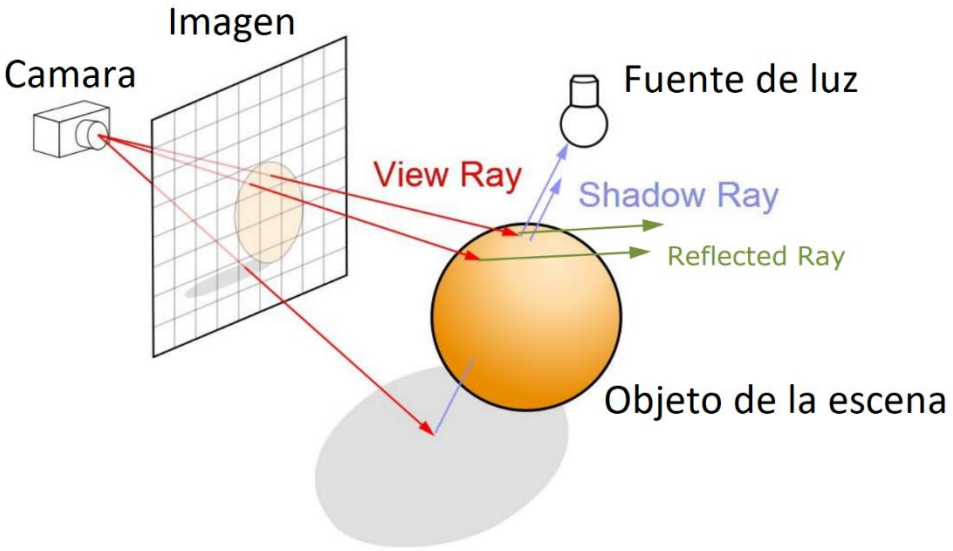


Turner Whitted evoluciona el concepto de raycasting en un artículo científico extendiéndolo y convirtiéndolo en recursivo. Basándose en que la luz, en su propagación, realiza rebotes cuando se encuentra con superficies, por ejemplo, reflectantes, extiende el trazado de rayos inicialmente planteado por Appel e incluye en su modelo la posibilidad de que a partir de la primera intersección rayo-escena se generen nuevos rayos dependiendo de las características del modelo con el que se interseca. Se identifican dos tipos de rayos:

Primary Rays: Son los primeros rayos que genera el algoritmo. Tienen como origen la posición del observador y atraviesan la cuadrícula que representa la imagen.
Secondary Rays: Son los rayos que se han generado a partir de la intersección de un rayo primario contra un modelo cuyas propiedades favorecen el rebote de la luz. Están incluidos todos los niveles de rebote a partir de la primera intersección. De este tipo de rayos podemos encontrar:
- Rayos por reflexión de la luz. Se generan a partir de la intersección de un rayo contra un modelo que presenta propiedades reflectantes.
- Rayos por refracción de la luz. Se generan a partir de la intersección de un rayo contra un modelo que presenta propiedades refractantes.

Shadow ray, es un rayo secundario que se utiliza para determinar partes de los modelos que se encuentran a la sombra. Que una región de un modelo se encuentra a la sombra, significa que a ningún punto de esa región le llega luz desde las fuentes de luz que tiene la escena. Esto lo resumimos en la siguiente imagen:




Tendríamos como un seudocódigo inicial simplificado:

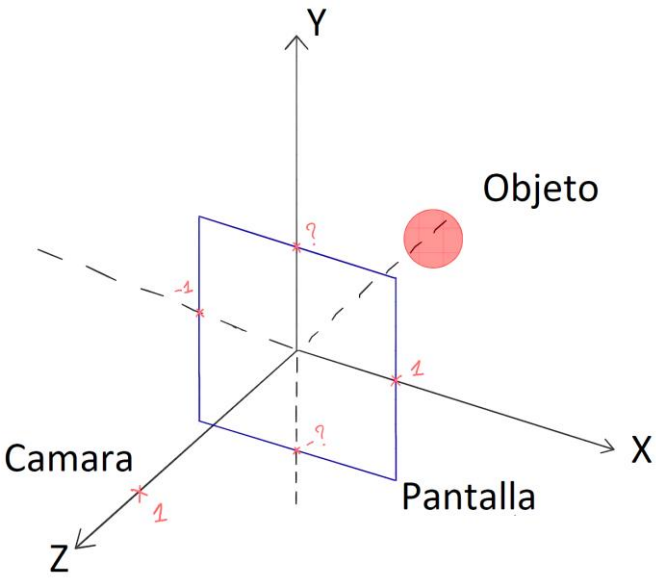
```
para cada píxel hacer
      calcular el rayo de visualización (view ray)
      si (rayo golpea un objeto con t ∈ [0,∞)) entonces
            Calcular n (punto)
            Evalúe el modelo de sombreado y establezca el píxel en
            ese color
      caso contrario
            establecer el color del píxel en el color de fondo
```

## 3. PRACTICA

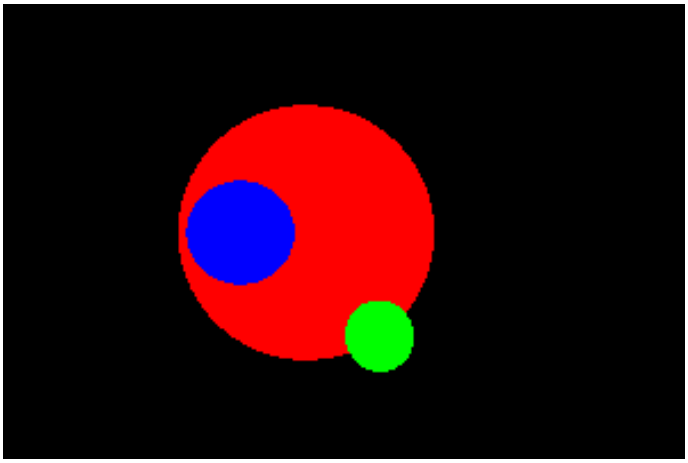
A. Revisa el siguiente código en Python que implementa ray tracing básico (sin considerar sombras, reflexiones, etc). Similar al visto en clase. Se están implementando 3 esferas de colores rojo, verde y azul.

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

def normalizar(vector):
    return vector / np.linalg.norm(vector) #norma del vector

def interseccion_esfera(centro, radio, origen_rayo, direccion_rayo): #calcular t
    a = np.dot(direccion_rayo, direccion_rayo) #producto punto
    b = 2 * np.dot(direccion_rayo, origen_rayo - centro)
    c = np.dot(origen_rayo - centro, origen_rayo - centro) - radio ** 2
    delta = b ** 2 - 4 * a * c
    if delta > 0:
        t1 = (-b + np.sqrt(delta)) / 2 * a
        t2 = (-b - np.sqrt(delta)) / 2 * a
        if t1 > 0 and t2 > 0:
            return min(t1, t2)
    return None

def objeto_intersectado_mas_cercano(objetos, origen_rayo, direccion_rayo):
    distancias = [interseccion_esfera(obj['centro'], obj['radio'], origen_rayo,
direccion_rayo) for obj in objetos]
    objeto_mas_cercano = None
    distancia_minima = np.inf #infinito positivo
    for indice, distancia in enumerate(distancias):
        if distancia and distancia < distancia_minima:
            distancia_minima = distancia
            objeto_mas_cercano = objetos[indice]
    return objeto_mas_cercano, distancia_minima

ancho = 300
alto = 200

camara = np.array([0, 0, 1])
proporcion = float(ancho) / alto
pantalla = (-1, 1 / proporcion, 1, -1 / proporcion) # izquierda, arriba, derecha, abajo

objetos = [
    { 'centro': np.array([-0.2, 0, -1]), 'radio': 0.7, 'color': [255, 0, 0] },
    { 'centro': np.array([0.1, -0.3, 0]), 'radio': 0.1, 'color': [0, 255, 0] },
    { 'centro': np.array([-0.3, 0, 0]), 'radio': 0.15, 'color': [0, 0, 255] }
]

imagen = np.zeros((alto, ancho, 3))
for i, y in enumerate(np.linspace(pantalla[1], pantalla[3], alto)):
    for j, x in enumerate(np.linspace(pantalla[0], pantalla[2], ancho)):
        pixel = np.array([x, y, 0])
        origen = camara
        direccion = normalizar(pixel - origen)

        # chequear por intersecciones
        objeto_mas_cercano, distancia_minima = objeto_intersectado_mas_cercano(objetos,
origen, direccion)
        if objeto_mas_cercano is None:
            continue
        # computar el punto de intereseccion entre el rayo y el objeto mas cercano
        interseccion = origen + distancia_minima * direccion
        color = objeto_mas_cercano['color']
        imagen[i, j] = np.clip(color, 0, 1)
        print("%d/%d" % (i + 1, alto))

plt.imsave('imagen.png', imagen)
imgplot = plt.imshow(imagen)
plt.show()
```

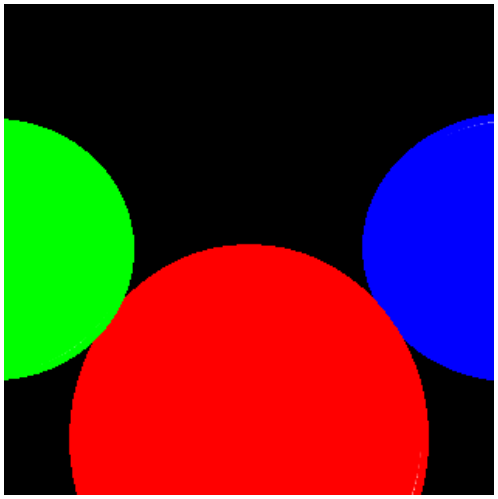
Si analizamos el código veremos que se esta implementando el siguiente escenario:



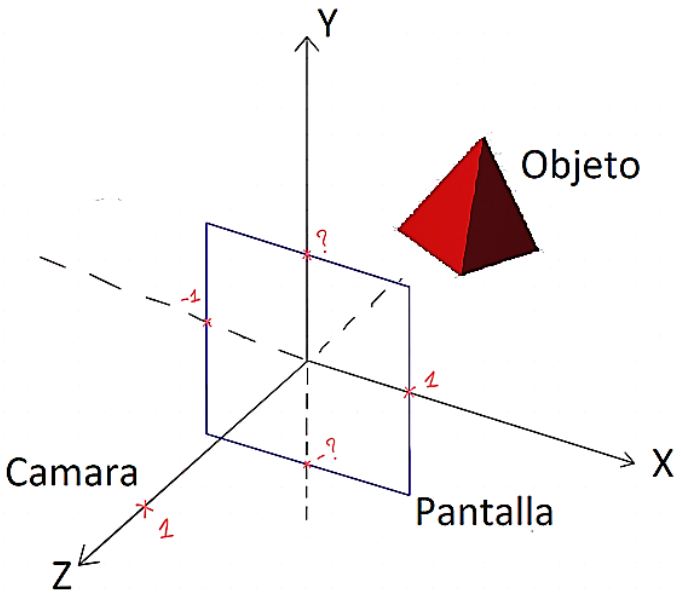

La imagen de salida obtenida es:



B. De manera similar implementa el ejemplo visto en clase, deberías obtener una salida similar a:



C. Ahora veamos el ejemplo de implementar una pirámide, similar al siguiente modelo:




Una pirámide triangular esta formada por 4 triángulos. Una concepción inicial podría ser dibujar los 4 triángulos, cada cara se puede tomar como la de un plano con la diferencia que el área de intersección es finita y con una forma definida, y no infinita como en el plano. Existe un método para calcular la intersección de un rayo con un triangulo sin necesidad de cálculo previo de la ecuación del plano del plano que contiene el triángulo llamado: "Algoritmo de intersección Möller–Trumbore". Revisar: https://en.wikipedia.org/wiki/M%C3%B6ller%E2%80%93Trumbore_intersection_algorithm

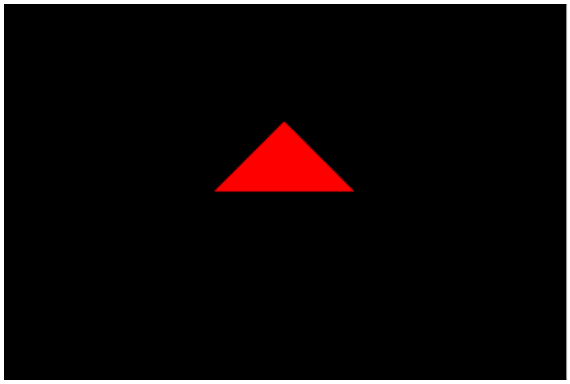
La siguiente función nos devolvería la distancia del triángulo al origen:

```
def interseccion_triangulo(vertice0, vertice1, vertice2, origen_rayo, direccion_rayo):
    EPSILON = 0.0000001;
    EDGE1 = vertice1 - vertice0
    EDGE2 = vertice2 - vertice0
    H = np.cross(direccion_rayo,EDGE2)
    A = np.dot(EDGE1, H)
    if ( A > -1 * EPSILON and A < EPSILON):
        return None
    F = 1.0 / A
    S = origen_rayo - vertice0
    U = F * (np.dot(S,H))
    if (U < 0.0 or U > 1.0):
        return None
    Q = np.cross(S,EDGE1)
    V = F * np.dot(direccion_rayo,Q)
    if (V < 0.0 or U + V > 1.0):
        return None
    T = F * np.dot(EDGE2,Q)
    if (T > EPSILON):
        print(T)
        return T
```

Prueba agregando un objeto triangulo, algo como:

```
objetos = [
    { 'vertice0': np.array([-0.5, 0, -1]), 'vertice1': np.array([0.5, 0, -1]), 'vertice2':
np.array([0, 0.5, -1]), 'color': [255, 0, 0] }
]
```

Verifica el correcto funcionamiento, deberías obtener una salida semejante a:



D. Ahora de manera similar implementa alguno de los siguientes objetos: (Naturalmente sin reflexiones, sombras, brillos, etc)