Línea 1: import numpy as np

- Importa la biblioteca `numpy` y le asigna el alias `np`. NumPy es fundamental para operaciones numéricas eficientes, especialmente con arrays multidimensionales, que se utilizarán para representar la imagen, vértices, colores y realizar cálculos vectoriales.

Línea 2: import matplotlib.pyplot as plt

- Importa el submódulo `pyplot` de la biblioteca `matplotlib` y le asigna el alias `plt`. Este módulo se utilizará para mostrar la imagen renderizada del triángulo sombreado.

Línea 3: (En blanco)

- Línea en blanco: Separa la sección de importaciones de la definición de los parámetros de la imagen.

Línea 4: # Dimensiones de la imagen

- Comentario: Indica que las siguientes líneas definirán las dimensiones del lienzo sobre el que se dibujará el triángulo.

Línea 5: width, height = 400, 400

- Define dos variables, `width` y `height`, y les asigna el valor entero `400` a ambas. Esto significa que la imagen generada tendrá un tamaño de 400 píxeles de ancho por 400 píxeles de alto.

Línea 6: image = np.ones((height, width, 3))

- Crea un array NumPy tridimensional llamado `image`.

- `np.ones((height, width, 3))`: Crea un array con las dimensiones `(height, width, 3)` (400x400x3) y lo inicializa con valores de `1.0` (tipo `float`).

- Un array de 400x400x3 representa una imagen de 400x400 píxeles con 3 canales de color (RGB). Inicializarlo con `1.0` en todos los canales resulta en una imagen completamente blanca.

Línea 7: (En blanco)

- Línea en blanco: Separa la inicialización de la imagen de la definición de los vértices del triángulo.

Línea 8: # Definición de los vértices del triángulo (coordenadas en píxeles)

- Comentario: Indica que las siguientes líneas definirán las coordenadas de los tres vértices del triángulo que se va a renderizar. Las coordenadas se especifican en el sistema de coordenadas de la imagen (píxeles).

Línea 9: v0 = np.array([100, 100])

- Define un array NumPy llamado `v0` que representa las coordenadas (x, y) del primer vértice del triángulo: `[100, 100]`.

Línea 10: v1 = np.array([300, 180])

- Define un array NumPy llamado `v1` que representa las coordenadas (x, y) del segundo vértice del triángulo: `[300, 180]`.

Línea 11: v2 = np.array([200, 300])

- Define un array NumPy llamado `v2` que representa las coordenadas (x, y) del tercer vértice del triángulo: `[200, 300]`.

Línea 12: vertices = np.array([v0, v1, v2])

- Crea un array NumPy bidimensional llamado `vertices` apilando los tres arrays de vértices `v0`, `v1`, `v2`. El resultado es un array de forma `(3, 2)`, donde cada fila es un vértice `[x, y]`.

Línea 13: (En blanco)

- Línea en blanco: Separa la definición de los vértices de la definición de sus colores.

Línea 14: # Colores asociados a cada vértice (en formato RGB, valores entre 0 y 1)

- Comentario: Indica que las siguientes líneas definirán los colores RGB (con componentes entre 0 y 1) que se asociarán a cada vértice del triángulo. Estos colores se usarán para la interpolación.

Línea 15: color_v0 = np.array([1.0, 0.0, 0.0]) # Rojo

- Define un array NumPy `color_v0` que representa el color RGB para el vértice `v0`: `[1.0, 0.0, 0.0]` (rojo puro).

Línea 16: color_v1 = np.array([0.0, 1.0, 0.0]) # Verde

- Define un array NumPy `color_v1` que representa el color RGB para el vértice `v1`: `[0.0, 1.0, 0.0]` (verde puro).

Línea 17: color_v2 = np.array([0.0, 0.0, 1.0]) # Azul

- Define un array NumPy `color_v2` que representa el color RGB para el vértice `v2`: `[0.0, 0.0, 1.0]` (azul puro).

Línea 18: colors = np.array([color_v0, color_v1, color_v2])

- Crea un array NumPy bidimensional llamado `colors` apilando los tres arrays de colores. El resultado es un array de forma `(3, 3)`, donde cada fila es un color `[R, G, B]`.

Línea 19: (En blanco)

- Línea en blanco: Separa la definición de los colores de la función `barycentric_coords`.

Línea 20: def barycentric_coords(p, a, b, c):

- Define una función llamada `barycentric_coords` que acepta cuatro parámetros: `p` (el punto para el que se calcularán las coordenadas baricéntricas) y `a`, `b`, `c` (los tres vértices del triángulo).

Línea 21: """

- Inicio del docstring de la función `barycentric_coords`.

Línea 22: Calcula las coordenadas baricéntricas de un punto p

- Línea del docstring: Describe el propósito principal de la función: calcular las coordenadas baricéntricas.

Línea 23: con respecto al triángulo formado por los puntos a, b y c.

- Línea del docstring: Completa la descripción, especificando el contexto del cálculo (dentro de un triángulo definido por `a`, `b`, `c`).

Línea 24: """

- Fin del docstring de la función `barycentric_coords`.

Línea 25: # Vectores del triángulo

- Dentro de la función `barycentric_coords`: Comentario que indica que las siguientes líneas calculan vectores que forman los lados del triángulo, relativos al vértice `a`.

Línea 26: v0 = b - a

- Dentro de la función `barycentric_coords`: Calcula el vector `v0` restando el punto `a` del punto `b`. Este vector representa el lado del triángulo desde `a` a `b`.

Línea 27: v1 = c - a

- Dentro de la función `barycentric_coords`: Calcula el vector `v1` restando el punto `a` del punto `c`. Este vector representa el lado del triángulo desde `a` a `c`.

Línea 28: v2 = p - a

- Dentro de la función `barycentric_coords`: Calcula el vector `v2` restando el punto `a` del punto `p`. Este vector va desde el vértice `a` hasta el punto `p` cuya ubicación baricéntrica se busca.

Línea 29: (En blanco)

- Dentro de la función `barycentric_coords`: Línea en blanco para separar los cálculos de vectores del cálculo del denominador.

Línea 30: # Producto cruzado (área * 2) para la región del triángulo

- Dentro de la función `barycentric_coords`: Comentario que explica que la siguiente línea calcula el doble del área del paralelogramo formado por `v0` y `v1`, que es equivalente a la magnitud del producto cruzado 2D (o el determinante de una matriz 2x2 formada por los vectores).

Línea 31: denom = v0[0] * v1[1] - v1[0] * v0[1]

- Dentro de la función `barycentric_coords`: Calcula el determinante de la matriz formada por los vectores `v0` y `v1`. En 2D, esto es `v0_x * v1_y - v1_x * v0_y`. Este valor es crucial para normalizar las coordenadas baricéntricas y representa el doble del área del triángulo (con signo).

Línea 32: (En blanco)

- Dentro de la función `barycentric_coords`: Línea en blanco para separar el cálculo del denominador del chequeo por división por cero.

Línea 33: # Evitar división por cero (triángulo degenerado)

- Dentro de la función `barycentric_coords`: Comentario que explica la importancia de la siguiente línea: evitar errores si el triángulo es "degenerado" (por ejemplo, si sus vértices son colineales y no forman un área real, `denom` sería cero).

Línea 34: if np.abs(denom) < 1e-6:

- Dentro de la función `barycentric_coords`: Condicional que verifica si el valor absoluto del `denom` es muy cercano a cero (menor que `1e-6`, un valor epsilon pequeño). Esto indica que el triángulo es degenerado o casi degenerado.

Línea 35: return -1, -1, -1

- Dentro de la función `barycentric_coords`: Si el triángulo es degenerado (la condición anterior es `True`), la función retorna `(-1, -1, -1)`. Estos valores negativos son una señal de que el punto no puede tener coordenadas baricéntricas significativas dentro de un triángulo válido.

Línea 36: (En blanco)

- Dentro de la función `barycentric_coords`: Línea en blanco para separar el manejo de casos degenerados del cálculo de las coordenadas.

Línea 37: # Cálculo de las coordenadas baricéntricas

- Dentro de la función `barycentric_coords`: Comentario que indica que las siguientes líneas calculan las tres coordenadas baricéntricas: alpha, beta y gamma.

Línea 38: alpha = (v2[0] * v1[1] - v1[0] * v2[1]) / denom

- Dentro de la función `barycentric_coords`: Calcula la coordenada baricéntrica `alpha`. Esta es la contribución del vértice `a` al punto `p`. La fórmula implica el determinante de los vectores `v2` y `v1` (o doble área del triángulo `pbc`).

Línea 39: beta = (v0[0] * v2[1] - v2[0] * v0[1]) / denom

- Dentro de la función `barycentric_coords`: Calcula la coordenada baricéntrica `beta`. Esta es la contribución del vértice `b` al punto `p`. La fórmula implica el determinante de los vectores `v0` y `v2` (o doble área del triángulo `pac`).

Línea 40: gamma = 1 - alpha - beta

- Dentro de la función `barycentric_coords`: Calcula la coordenada baricéntrica `gamma`. Dado que la suma de las coordenadas baricéntricas siempre debe ser 1 (`alpha + beta + gamma = 1`), `gamma` se puede calcular como `1 - alpha - beta`. Esta es la contribución del vértice `c` al punto `p`.

Línea 41: (En blanco)

- Dentro de la función `barycentric_coords`: Línea en blanco para separar los cálculos del retorno.

Línea 42: return alpha, beta, gamma

- Dentro de la función `barycentric_coords`: Retorna los tres valores `alpha`, `beta`, `gamma` como una tupla.

Línea 43: (En blanco)

- Línea en blanco: Separa la definición de la función `barycentric_coords` del cálculo del rectángulo contenedor.

Línea 44: # Calcular el rectángulo contenedor (bounding box) del triángulo

- Comentario: Indica que las siguientes líneas calcularán el "bounding box" (rectángulo que encierra) el triángulo. Esto es una optimización para evitar iterar sobre todos los píxeles de la imagen, limitando la búsqueda a una región más pequeña.

Línea 45: min_x = int(np.clip(np.min(vertices[:, 0]), 0, width-1))

- Calcula el valor mínimo de la coordenada X de los vértices: `np.min(vertices[:, 0])`.

- `np.clip(..., 0, width-1)`: Asegura que este valor mínimo no sea menor que 0 (borde izquierdo de la imagen) ni mayor que `width-1` (borde derecho). Esto es una precaución para triángulos que podrían estar parcialmente fuera de los límites de la imagen.

- `int(...)`: Convierte el resultado a un entero, ya que las coordenadas de píxeles deben ser enteras.

- El resultado se asigna a `min_x`, que es la coordenada X mínima del bounding box.

Línea 46: max_x = int(np.clip(np.max(vertices[:, 0]), 0, width-1))

- Calcula el valor máximo de la coordenada X de los vértices: `np.max(vertices[:, 0])`.

- `np.clip(..., 0, width-1)`: Similar a `min_x`, asegura que este valor máximo esté dentro de los límites válidos de la imagen.

- `int(...)`: Convierte a entero.

- El resultado se asigna a `max_x`, que es la coordenada X máxima del bounding box.

Línea 47: min_y = int(np.clip(np.min(vertices[:, 1]), 0, height-1))

- Calcula el valor mínimo de la coordenada Y de los vértices: `np.min(vertices[:, 1])`.

- `np.clip(..., 0, height-1)`: Asegura que este valor mínimo esté dentro de los límites válidos de la imagen.

- `int(...)`: Convierte a entero.

- El resultado se asigna a `min_y`, que es la coordenada Y mínima del bounding box.

Línea 48: max_y = int(np.clip(np.max(vertices[:, 1]), 0, height-1))

- Calcula el valor máximo de la coordenada Y de los vértices: `np.max(vertices[:, 1])`.

- `np.clip(..., 0, height-1)`: Asegura que este valor máximo esté dentro de los límites válidos de la imagen.

- `int(...)`: Convierte a entero.

- El resultado se asigna a `max_y`, que es la coordenada Y máxima del bounding box.

Línea 49: (En blanco)

- Línea en blanco: Separa el cálculo del bounding box del bucle principal de rasterización.

Línea 50: # Recorrer los píxeles dentro del bounding box

- Comentario: Indica que las siguientes líneas implementarán un bucle anidado que iterará sobre cada píxel dentro del rectángulo contenedor calculado, en lugar de toda la imagen.

Línea 51: for y in range(min_y, max_y+1):

- Inicia el bucle exterior `for`, que iterará sobre las coordenadas `y` de los píxeles, desde `min_y` hasta `max_y` (inclusive).

Línea 52: for x in range(min_x, max_x+1):

- Dentro del bucle `for y`: Inicia el bucle interior `for`, que iterará sobre las coordenadas `x` de los píxeles, desde `min_x` hasta `max_x` (inclusive). En cada iteración de este bucle, `(x, y)` representa la coordenada del píxel actual.

Línea 53: p = np.array([x, y])

- Dentro del bucle anidado: Crea un array NumPy `p` que representa el punto de píxel actual `[x, y]`.

Línea 54: (En blanco)

- Dentro del bucle anidado: Línea en blanco para separar la definición del punto `p` de la llamada a `barycentric_coords`.

Línea 55: # Obtener coordenadas baricéntricas para el píxel

- Dentro del bucle anidado: Comentario que explica el propósito de la siguiente línea: calcular las coordenadas baricéntricas para el píxel actual.

Línea 56: alpha, beta, gamma = barycentric_coords(p, v0, v1, v2)

- Dentro del bucle anidado: Llama a la función `barycentric_coords` con el punto `p` actual y los vértices `v0`, `v1`, `v2` del triángulo. Los valores devueltos (alpha, beta, gamma) se asignan a las variables correspondientes.

Línea 57: (En blanco)

- Dentro del bucle anidado: Línea en blanco para separar el cálculo de coordenadas baricéntricas de la verificación de pertenencia al triángulo.

Línea 58: # Verificar si el punto se encuentra dentro del triángulo

- Dentro del bucle anidado: Comentario que explica el propósito de la siguiente línea: usar las coordenadas baricéntricas para determinar si el píxel está dentro del triángulo.

Línea 59: if alpha >= 0 and beta >= 0 and gamma >= 0:

- Dentro del bucle anidado: Condicional que verifica si el punto `p` está dentro del triángulo. Un punto está dentro o en el borde del triángulo si y solo si todas sus coordenadas baricéntricas (`alpha`, `beta`, `gamma`) son mayores o iguales a cero.

- Si la suma de `alpha`, `beta`, `gamma` es 1 y todas son no negativas, el punto está dentro. Si alguna es negativa, el punto está fuera (en uno de los "lados" extendidos del triángulo).

Líne 60: # Interpolación de colores usando las coordenadas baricéntricas

- Dentro del bloque `if` : Comentario que indica que, dado que el píxel está dentro del triángulo, se procederá a interpolar su color.

Línea 61: pixel_color = alpha * color_v0 + beta * color_v1 + gamma * color_v2

- Dentro del bloque `if` : Calcula el color del píxel actual (`pixel_color`) mediante interpolación lineal de los colores de los vértices (`color_v0`, `color_v1`, `color_v2`) utilizando las coordenadas baricéntricas `alpha`, `beta`, `gamma`.

- `alpha * color_v0` : Multiplica el escalar `alpha` por el array de color `color_v0` (componente a componente).

- `beta * color_v1` : Similar para `beta` y `color_v1`.

- `gamma * color_v2` : Similar para `gamma` y `color_v2`.

- Los tres resultados (que son arrays RGB) se suman componente a componente para obtener el color final del píxel.

Línea 62: image[y, x] = pixel_color

- Dentro del bloque `if` : Asigna el `pixel_color` calculado a la posición `(y, x)` en el array `image`. Esto colorea el píxel correspondiente en la imagen.

Línea 63: (En blanco)

- Línea en blanco: Separa la lógica de rasterización de la sección de visualización.

Línea 64: # Mostrar el resultado

- Comentario: Indica el inicio de la sección donde se utilizará Matplotlib para mostrar la imagen generada.

Línea 65: plt.figure(figsize=(6, 6))

- Crea una nueva figura de Matplotlib.

- `figsize=(6, 6)`: Especifica el tamaño de la figura en pulgadas (6 pulgadas de ancho por 6 pulgadas de alto).

Línea 66: plt.imshow(image)

- Muestra la imagen `image` (el array NumPy 400x400x3 que ahora contiene el triángulo coloreado) en la figura actual. `plt.imshow` es la función estándar de Matplotlib para mostrar datos de array como una imagen.

Línea 67: plt.title("Sombreado con Interpolación (Baricéntrico)")

- Establece el título del gráfico como la cadena "Sombreado con Interpolación (Baricéntrico)".

Línea 68: plt.axis("off")

- Desactiva la visualización de los ejes (números, marcas y líneas de los ejes X e Y) en el gráfico, ya que no son relevantes para mostrar una imagen renderizada.

Línea 69: plt.show()

- Muestra la figura de Matplotlib con la imagen renderizada y el título. Esta función también inicia el bucle de eventos de la GUI de Matplotlib, si es necesario, y el script generalmente se pausará aquí hasta que se cierre la ventana del gráfico.

Resumen del Código

Este script de Python utiliza las bibliotecas numpy y matplotlib para renderizar un triángulo 2D en una imagen, aplicando un sombreado con interpolación de color basada en coordenadas baricéntricas.

1. **Configuración Inicial:**
    - Define las dimensiones (width, height) de la imagen a generar, creando un lienzo en blanco (blanco puro) de ese tamaño.

- Establece las coordenadas en píxeles (v0, v1, v2) de los tres vértices del triángulo a dibujar.
- Asocia un color RGB distinto a cada vértice: rojo para v0, verde para v1 y azul para v2.

2. **Función de Coordenadas Baricéntricas:**
    - Define una función barycentric_coords(p, a, b, c) que calcula las coordenadas baricéntricas (alpha, beta, gamma) para un punto p dado, relativas a un triángulo definido por sus vértices a, b, y c.
    - Internamente, calcula el doble del área del triángulo (o un determinante) para el denominador.
    - Incluye una comprobación para triángulos degenerados (colineales o de área cero) para evitar divisiones por cero, en cuyo caso devuelve (-1, -1, -1).
    - Las coordenadas baricéntricas indican el "peso" de cada vértice en la posición del punto p. Si todas son no negativas, el punto está dentro o en el borde del triángulo.
3. **Rasterización y Sombreado:**
    - **Bounding Box:** Primero, calcula el "rectángulo contenedor" (bounding box) que encierra completamente el triángulo. Esto optimiza el proceso de rasterización al limitar la iteración de píxeles solo a esta región, en lugar de toda la imagen.
    - **Recorrido de Píxeles:** Itera a través de cada píxel (x, y) dentro de este bounding box.
    - **Verificación de Pertenencia:** Para cada píxel, llama a barycentric_coords para obtener sus coordenadas baricéntricas. Si todas las coordenadas son mayores o iguales a cero, significa que el píxel está dentro del triángulo.
    - **Interpolación de Color:** Si el píxel está dentro del triángulo, su color final se calcula como una combinación lineal de los colores de los vértices, ponderada por las coordenadas baricéntricas. Por ejemplo, si el píxel está cerca de v0, su color será más parecido al rojo de color_v0.

- **Asignación de Color:** El color interpolado se asigna al píxel correspondiente en el array image.

4. **Visualización del Resultado:**
    - Finalmente, utiliza matplotlib.pyplot para mostrar la image generada.
    - Añade un título descriptivo y desactiva los ejes del gráfico para una visualización limpia.

El resultado es una imagen que muestra un triángulo con un sombreado degradado suave, donde los colores de los vértices (rojo, verde, azul) se mezclan armónicamente en el interior del triángulo, ilustrando el concepto de interpolación de atributos en gráficos por computadora.

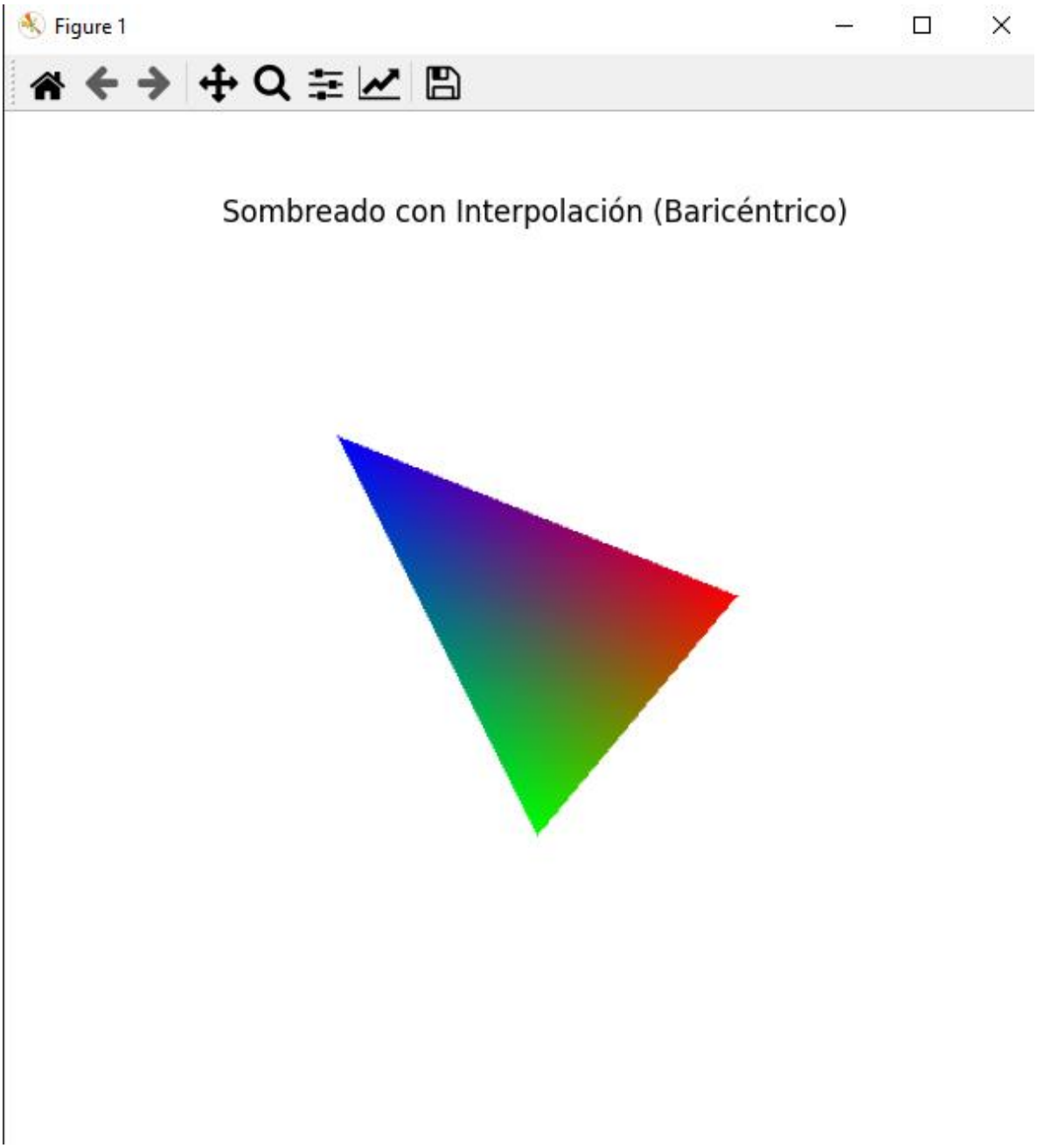
Figure 1
Sombreado con Interpolación (Baricéntrico)