Línea 1: # 4.1.2D_Graficación_sombreado_Phong_2025 > ...

- Comentario: Indica una posible ruta de archivo o identificador de proyecto, específico del contexto del usuario. La parte final `...` sugiere que el comentario original podría ser más largo.

Línea 2: import numpy as np

- Importa la biblioteca `numpy` y le asigna el alias `np`. Se utilizará para operaciones numéricas, especialmente con vectores (normales, direcciones) y arrays (imagen, colores).

Línea 3: from PIL import Image, ImageDraw

- Importa las clases `Image` e `ImageDraw` desde la biblioteca `PIL` (Pillow). `Image` se usará para crear y manipular la imagen base, y `ImageDraw`, aunque importado, no se utiliza explícitamente en el código restante (su propósito general es dibujar formas o texto en imágenes).

Línea 4: import math

- Importa el módulo `math`, que proporciona funciones matemáticas básicas como `sin`, `cos` y `pi`, usadas para calcular los vértices del polígono.

Línea 5: import matplotlib.pyplot as plt

- Importa el submódulo `pyplot` de la biblioteca `matplotlib` y le asigna el alias `plt`. Se usará para mostrar la imagen resultante.

Línea 6: (En blanco)

- Línea en blanco: Separa las importaciones de las definiciones de funciones.

Línea 7: def normalize(v):

- Define una función llamada `normalize` que acepta un parámetro `v` (se espera que sea un vector o un array tipo vector).

Línea 8: norm = np.linalg.norm(v)

- Dentro de la función `normalize`: Calcula la norma euclidiana (magnitud o longitud) del vector `v` usando `np.linalg.norm()` y la almacena en la variable `norm`.

Línea 9: return v / norm if norm != 0 else v

- Dentro de la función `normalize`: Retorna el vector normalizado. Utiliza una expresión condicional (operador ternario): si `norm` no es cero, divide el vector `v` por su norma `norm`. Si `norm` es cero (lo que significa que `v` era el vector cero), devuelve el vector `v` original para evitar un error de división por cero.

Línea 10: (En blanco)

- Línea en blanco: Separa la función `normalize` de la siguiente función.

Línea 11: def triangle_phong_fill_color(img_array, verts, normals, base_colors, light_dir, view_dir,

- Define una función llamada `triangle_phong_fill_color`. Acepta varios parámetros: `img_array` (el array NumPy de la imagen a modificar), `verts` (vértices 2D del triángulo), `normals` (normales 3D en los vértices), `base_colors` (colores RGB en los vértices), `light_dir` (dirección de la luz), `view_dir` (dirección de la vista). La definición de parámetros continúa en la siguiente línea.

Línea 12: ambient=0.1, diffuse_coef=1.0, specular_coef=1.0, shininess=20):

- Continuación de la definición de la función `triangle_phong_fill_color`: Define parámetros adicionales con valores por defecto para el modelo de iluminación Phong: `ambient` (intensidad ambiente), `diffuse_coef` (coeficiente difuso), `specular_coef` (coeficiente especular) y `shininess` (exponente de brillo especular).

Línea 13: """

- Inicio del docstring (cadena de documentación) para la función `triangle_phong_fill_color`.

Línea 14: Rellena un triángulo definido por `verts` (lista de 3 tuplas (x, y)) usando el modelo de iluminación Phong.

- Línea del docstring: Describe la acción principal de la función: rellenar un triángulo 2D aplicando el modelo de sombreado de Phong.

Línea 15: Se interpolan los vectores normales (parámetro `normals`) y los colores base (parámetro `base_colors`) mediante

- Línea del docstring: Explica que la función interpola las normales y los colores. La línea continúa.

Línea 16: coordenadas baricéntricas para cada píxel, y se calcula el color final (en RGB) combinando las componentes ambiente,

- Línea del docstring: Continúa la explicación: la interpolación usa coordenadas baricéntricas por píxel, y el color final se calcula combinando componentes de iluminación. La línea continúa.

Línea 17: difusa y especular.

- Línea del docstring: Finaliza la descripción del cálculo del color, mencionando las componentes difusa y especular.

Línea 18: Parámetros:

- Línea del docstring: Indica el inicio de la descripción de los parámetros.

Línea 19: - img_array: arreglo NumPy de la imagen (se modifica in situ).

- Línea del docstring: Describe el parámetro `img_array`, especificando que es un array NumPy y que la función lo modificará directamente.

Línea 20: - verts: lista de 3 vértices (x, y) del triángulo.

- Línea del docstring: Describe el parámetro `verts`, indicando que es una lista de 3 tuplas de coordenadas (x, y).

Línea 21: - normals: lista de 3 vectores normales (cada uno de 3 componentes) asociados a cada vértice.

- Línea del docstring: Describe el parámetro `normals`, indicando que es una lista de 3 vectores normales (arrays o tuplas de 3 elementos) correspondientes a los vértices.

Línea 22: - base_colors: lista de 3 colores base (tuplas RGB, valores 0-255) para cada vértice.

- Línea del docstring: Describe el parámetro `base_colors`, indicando que es una lista de 3 colores RGB (en formato tupla, con valores de 0 a 255).

Línea 23: - light_dir: vector dirección de la luz (normalizado).

- Línea del docstring: Describe el parámetro `light_dir`, esperando un vector normalizado.

Línea 24: - view_dir: vector dirección de la cámara (normalizado).

- Línea del docstring: Describe el parámetro `view_dir`, esperando un vector normalizado.

Línea 25: - ambient: componente ambiente.

- Línea del docstring: Describe el parámetro `ambient`.

Línea 26: - diffuse_coef: coeficiente difuso.

- Línea del docstring: Describe el parámetro `diffuse_coef`.

Línea 27: - specular_coef: coeficiente especular.

- Línea del docstring: Describe el parámetro `specular_coef`.

Línea 28: - shininess: exponente de brillo para la componente especular.

- Línea del docstring: Describe el parámetro `shininess`.

Línea 29: """

- Fin del docstring de la función `triangle_phong_fill_color`.

Línea 30: # Extraer coordenadas de los vértices

- Dentro de la función `triangle_phong_fill_color`: Comentario que indica que las siguientes líneas extraerán las coordenadas X e Y de los vértices proporcionados.

Línea 31: xs = [v[0] for v in verts]

- Dentro de la función `triangle_phong_fill_color`: Utiliza una lista por comprensión para crear una nueva lista `xs` que contiene solo el primer elemento (la coordenada x) de cada tupla de vértice `v` en la lista `verts`.

Línea 32: ys = [v[1] for v in verts]

- Dentro de la función `triangle_phong_fill_color`: Similarmente, crea una lista `ys` con las coordenadas y (el segundo elemento) de cada vértice en `verts`.

Línea 33: (En blanco)

- Dentro de la función `triangle_phong_fill_color`: Línea en blanco para mejorar la legibilidad, separando la extracción de coordenadas del cálculo de la caja envolvente.

Línea 34: # Calcular la caja envolvente del triángulo (limitada a los bordes de la imagen)

- Dentro de la función `triangle_phong_fill_color`: Comentario que explica que las siguientes líneas calcularán los límites (bounding box) del área a escanear para el triángulo, asegurándose de que no se salga de los límites de la imagen.

Línea 35: min_x = max(int(min(xs)), 0)

- Dentro de la función `triangle_phong_fill_color`: Calcula la coordenada x mínima de la caja envolvente. `min(xs)` encuentra la x más pequeña entre los vértices, `int()` la convierte a entero (índice de píxel), y `max(..., 0)` asegura que el valor no sea menor que 0 (el borde izquierdo de la imagen).

Línea 36: max_x = min(int(max(xs)), img_array.shape[1] - 1)

- Dentro de la función `triangle_phong_fill_color`: Calcula la coordenada x máxima. `max(xs)` encuentra la x más grande, `int()` la convierte a entero, y `min(..., img_array.shape[1] - 1)` asegura que el valor no exceda el índice del último píxel en la dirección horizontal (el ancho de la imagen `img_array.shape[1]`, menos 1 porque los índices son 0-based).

Línea 37: min_y = max(int(min(ys)), 0)

- Dentro de la función `triangle_phong_fill_color`: Calcula la coordenada y mínima de manera análoga a `min_x`, usando `min(ys)` y asegurándose de que no sea menor que 0 (borde superior de la imagen).

Línea 38: max_y = min(int(max(ys)), img_array.shape[0] - 1)

- Dentro de la función `triangle_phong_fill_color`: Calcula la coordenada y máxima de manera análoga a `max_x`, usando `max(ys)` y asegurándose de que no exceda el índice del último píxel verticalmente (la altura de la imagen `img_array.shape[0]`, menos 1).

Línea 39: (En blanco)

- Dentro de la función `triangle_phong_fill_color`: Línea en blanco para separar el cálculo de la caja envolvente de la extracción individual de vértices.

Línea 40: # Extraer vértices individuales

- Dentro de la función `triangle_phong_fill_color`: Comentario indicando que se asignarán las coordenadas de cada vértice a variables separadas.

Línea 41: x0, y0 = verts[0]

- Dentro de la función `triangle_phong_fill_color`: Desempaqueta la tupla del primer vértice (`verts[0]`) en las variables `x0` e `y0`.

Línea 42: x1, y1 = verts[1]

- Dentro de la función `triangle_phong_fill_color`: Desempaqueta la tupla del segundo vértice (`verts[1]`) en `x1` e `y1`.

Línea 43: x2, y2 = verts[2]

- Dentro de la función `triangle_phong_fill_color`: Desempaqueta la tupla del tercer vértice (`verts[2]`) en `x2` e `y2`.

Línea 44: (En blanco)

- Dentro de la función `triangle_phong_fill_color`: Línea en blanco separadora.

Línea 45: # Precalcular el denominador para coordenadas baricéntricas

- Dentro de la función `triangle_phong_fill_color`: Comentario explicando que se calculará una parte común de las fórmulas de coordenadas baricéntricas.

Línea 46: denom = ((y1 - y2) * (x0 - x2) + (x2 - x1) * (y0 - y2))

- Dentro de la función `triangle_phong_fill_color`: Calcula el denominador usado en las fórmulas estándar para las coordenadas baricéntricas. Este valor es constante para todos los píxeles dentro del triángulo y es proporcional al área (con signo) del triángulo.

Línea 47: if denom == 0:

- Dentro de la función `triangle_phong_fill_color`: Inicia una estructura condicional `if`. Comprueba si el denominador calculado es igual a 0.

Línea 48: return # Triángulo degenerado

- Dentro del bloque `if`: Si `denom` es 0, significa que los vértices son colineales (el triángulo no tiene área). La función termina inmediatamente con `return` para evitar errores de división por cero más adelante. El comentario indica la razón.

Línea 49: (En blanco)

- Dentro de la función `triangle_phong_fill_color`: Línea en blanco separadora.

Línea 50: # Recorrer cada píxel dentro de la caja envolvente

- Dentro de la función `triangle_phong_fill_color`: Comentario que indica el inicio de los bucles que iterarán sobre los píxeles potencialmente dentro del triángulo.

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

- Dentro de la función `triangle_phong_fill_color`: Inicia un bucle `for` que itera sobre todas las coordenadas `y` enteras desde `min_y` hasta `max_y` (inclusive).

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

- Dentro del bucle `for y`: Inicia un bucle `for` anidado que itera sobre todas las coordenadas `x` enteras desde `min_x` hasta `max_x` (inclusive) para la `y` actual.

Línea 53: # Calcular coordenadas baricéntricas

- Dentro del bucle `for x` : Comentario que indica que se calcularán las coordenadas baricéntricas para el píxel actual `(x, y)` .

Línea 54: w0 = ((y1 - y2) * (x - x2) + (x2 - x1) * (y - y2)) / denom

- Dentro del bucle `for x` : Calcula la coordenada baricéntrica `w0` (peso asociado al vértice 0) para el píxel `(x, y)` usando la fórmula estándar y el denominador precalculado `denom` .

Línea 55: w1 = ((y2 - y0) * (x - x0) + (x0 - x2) * (y - y0)) / denom

- Dentro del bucle `for x` : Calcula la coordenada baricéntrica `w1` (peso asociado al vértice 1).

Línea 56: w2 = 1 - w0 - w1

- Dentro del bucle `for x` : Calcula la coordenada baricéntrica `w2` (peso asociado al vértice 2). Se aprovecha que la suma de las tres coordenadas debe ser 1 (`w0 + w1 + w2 = 1`).

Línea 57: (En blanco)

- Dentro del bucle `for x` : Línea en blanco para separar el cálculo de coordenadas de la comprobación de pertenencia.

Línea 58: # Verificar si el píxel (x, y) se encuentra dentro del triángulo

- Dentro del bucle `for x` : Comentario que explica el propósito de la siguiente condición `if` .

Línea 59: if w0 >= 0 and w1 >= 0 and w2 >= 0:

- Dentro del bucle `for x`: Inicia un bloque `if`. La condición comprueba si las tres coordenadas baricéntricas (`w0`, `w1`, `w2`) son no negativas. Si esto se cumple, el píxel `(x, y)` está dentro del triángulo (o en su borde).

Línea 60: # Interpolar el vector normal y normalizarlo

- Dentro del bloque `if` (el píxel está dentro del triángulo): Comentario explicando que se interpolará el vector normal.

Línea 61: n_interp = w0 * np.array(normals[0]) + w1 * np.array(normals[1]) + w2 * np.array(normals[2])

- Dentro del bloque `if`: Calcula el vector normal interpolado `n_interp` en el píxel actual. Multiplica cada coordenada baricéntrica (`w0`, `w1`, `w2`) por el vector normal del vértice correspondiente (`normals[0]`, `normals[1]`, `normals[2]`, convertidos a arrays NumPy para la operación) y suma los resultados.

Línea 62: n_interp = normalize(n_interp)

- Dentro del bloque `if`: Llama a la función `normalize` (definida anteriormente) para asegurarse de que el vector normal interpolado `n_interp` tenga longitud unitaria, lo cual es necesario para los cálculos de iluminación Phong.

Línea 63: (En blanco)

- Dentro del bloque `if`: Línea en blanco para separar la interpolación de la normal de la interpolación del color.

Línea 64: # Interpolar el color base (convertido a [0,1])

- Dentro del bloque `if`: Comentario explicando que se interpolará el color base del vértice.

Línea 65: color_interp = (w0 * np.array(base_colors[0]) +

- Dentro del bloque `if`: Comienza el cálculo del color base interpolado `color_interp`. Similar a la normal, multiplica `w0` por el color del primer vértice (`base_colors[0]`, convertido a array NumPy). La expresión continúa en las siguientes líneas.

Línea 66: w1 * np.array(base_colors[1]) +

- Dentro del bloque `if`: Continúa el cálculo, añadiendo el producto de `w1` por el color del segundo vértice (`base_colors[1]`).

Línea 67: w2 * np.array(base_colors[2])) / 255.0

- Dentro del bloque `if`: Finaliza el cálculo, añadiendo el producto de `w2` por el color del tercer vértice (`base_colors[2]`). El resultado total de la suma ponderada (que está en el rango 0-255) se divide por `255.0` para normalizar los componentes RGB al rango [0.0, 1.0], requerido para los cálculos de iluminación.

Línea 68: (En blanco)

- Dentro del bloque `if`: Línea en blanco antes de los cálculos del modelo Phong.

Línea 69: # Modelo Phong: I = Ia + Id + Is

- Dentro del bloque `if`: Comentario que indica el inicio de los cálculos del modelo de iluminación Phong y su fórmula básica (Intensidad = Ambiente + Difusa + Especular).

Línea 70: Ia = ambient

- Dentro del bloque `if`: Calcula la componente de intensidad ambiente `Ia`. Simplemente toma el valor del parámetro `ambient` de la función.

Línea 71: NdotL = max(np.dot(n_interp, light_dir), 0)

- Dentro del bloque `if` : Calcula el producto punto (`np.dot`) entre el vector normal interpolado `n_interp` y el vector de dirección de la luz `light_dir`. El resultado se limita a un mínimo de 0 usando `max(..., 0)`, porque la luz que incide desde detrás de la superficie no contribuye a la reflexión difusa o especular. El resultado se guarda en `NdotL`.

Línea 72: Id = diffuse_coef * NdotL

- Dentro del bloque `if` : Calcula la componente de intensidad difusa `Id` multiplicando el coeficiente difuso (`diffuse_coef`) por el `NdotL` calculado previamente.

Línea 73: # Calcular el vector reflejado: R = 2*(N·L)*N - L

- Dentro del bloque `if` : Comentario que explica el cálculo del vector de reflexión `R` y muestra su fórmula matemática.

Línea 74: R = 2 * NdotL * n_interp - light_dir

- Dentro del bloque `if` : Calcula el vector de reflexión `R` de la luz sobre la superficie utilizando la fórmula: `2 * (N · L) * N - L`, donde `N` es `n_interp` y `L` es `light_dir`.

Línea 75: R = normalize(R)

- Dentro del bloque `if` : Normaliza el vector de reflexión `R` recién calculado usando la función `normalize`.

Línea 76: RdotV = max(np.dot(R, view_dir), 0)

- Dentro del bloque `if` : Calcula el producto punto entre el vector de reflexión normalizado `R` y el vector de dirección de la vista `view_dir`. El resultado se limita a un mínimo de 0 con `max(..., 0)` porque el brillo especular solo ocurre si el reflejo va hacia el observador. El resultado se guarda en `RdotV`.

Línea 77: Is = specular_coef * (RdotV ** shininess)

- Dentro del bloque `if`: Calcula la componente de intensidad especular `Is`. Eleva `RdotV` a la potencia del exponente de brillo `shininess` y lo multiplica por el coeficiente especular `specular_coef`.

Línea 78: (En blanco)

- Dentro del bloque `if`: Línea en blanco antes del cálculo del color final.

Línea 79: # Calcular el color final por canal:

- Dentro del bloque `if`: Comentario indicando que se calculará el color final del píxel.

Línea 80: # Se aplica la base de color a la componente ambiente + difusa y se suma la componente especular (usamos blanco para el especular).

- Dentro del bloque `if`: Comentario más detallado explicando cómo se combinan las componentes: el color base modula las intensidades ambiente y difusa, mientras que la componente especular se asume de color blanco y se suma.

Línea 81: final_color = color_interp * (Ia + Id) + Is * np.array([1, 1, 1])

- Dentro del bloque `if`: Calcula el color final del píxel. Multiplica el color base interpolado (`color_interp`, en rango [0,1]) por la suma de las intensidades ambiente y difusa (`Ia + Id`). Luego, suma la intensidad especular (`Is`) multiplicada por un vector de color blanco `np.array([1, 1, 1])`.

Línea 82: final_color = np.clip(final_color, 0, 1) * 255

- Dentro del bloque `if`: Limita (clamping) los componentes del `final_color` calculados para que estén estrictamente en el rango [0, 1] usando `np.clip`. Esto evita valores de color inválidos si las intensidades sumadas superan 1. Luego, multiplica por 255 para escalar los componentes RGB de vuelta al rango [0, 255].

Línea 83: img_array[y, x] = tuple(final_color.astype(np.uint8))

- Dentro del bloque `if`: Convierte los componentes del `final_color` (que pueden ser flotantes) al tipo de dato entero sin signo de 8 bits (`np.uint8`), adecuado para colores RGB. Convierte el array resultante en una tupla. Finalmente, asigna esta tupla de color RGB al píxel correspondiente `(y, x)` en el `img_array`, modificando la imagen. Nótese el acceso `[y, x]` que es (fila, columna).

Línea 84: (En blanco)

- Línea en blanco: Separa el final de la función `triangle_phong_fill_color` del código principal del script.

Línea 85: # Dimensiones de la imagen

- Comentario: Indica que se definirán las dimensiones de la imagen a crear.

Línea 86: width, height = 500, 500

- Define dos variables, `width` y `height`, y les asigna el valor 500 a ambas. Estas determinarán el tamaño de la imagen generada.

Línea 87: (En blanco)

- Línea en blanco: Separa la definición de dimensiones de la creación de la imagen.

Línea 88: # Crear una imagen en blanco y convertirla a arreglo NumPy

- Comentario: Explica los siguientes dos pasos: crear una imagen base y obtener su representación como array NumPy.

Línea 89: img = Image.new("RGB", (width, height), "white")

- Utiliza `Image.new()` de la biblioteca PIL/Pillow para crear un nuevo objeto de imagen.

- `"RGB"`: Especifica el modo de color (rojo, verde, azul).

- `(width, height)`: Define el tamaño de la imagen usando las variables previamente definidas.

- `"white"`: Establece el color inicial de fondo de la imagen como blanco.

- El objeto imagen resultante se almacena en la variable `img`.

Línea 90: img_array = np.array(img)

- Convierte el objeto de imagen PIL `img` en un array NumPy llamado `img_array`. Este array tendrá dimensiones (height, width, 3) y contendrá los valores RGB (0-255) para cada píxel. Trabajar con el array permite modificar los píxeles eficientemente.

Línea 91: (En blanco)

- Línea en blanco: Separa la creación de la imagen de la definición de los parámetros del polígono.

Línea 92: # Parámetros del eneágono (9 lados)

- Comentario: Indica que las siguientes líneas definirán las propiedades de un eneágono (polígono de 9 lados).

Línea 93: num_sides = 9

- Define la variable `num_sides` con el valor 9, especificando el número de lados del polígono.

Línea 94: radius = 200

- Define la variable `radius` con el valor 200. Representa la distancia desde el centro del polígono hasta cada uno de sus vértices (en píxeles).

Línea 95: center = (width // 2, height // 2)

- Calcula las coordenadas del centro de la imagen usando división entera (`//`) de `width` y `height` por 2. El resultado es una tupla `(x_centro, y_centro)` que se almacena en la variable `center`.

Línea 96: polygon = []

- Inicializa una lista vacía llamada `polygon`. Esta lista se usará para almacenar las coordenadas `(x, y)` de los vértices del eneágono.

Línea 97: (En blanco)

- Línea en blanco: Separa la inicialización de parámetros del bucle de generación de vértices.

Línea 98: # Generar los vértices del eneágono

- Comentario: Indica que el siguiente bucle calculará las posiciones de los vértices.

Línea 99: for i in range(num_sides):

- Inicia un bucle `for` que se ejecutará `num_sides` (9) veces. La variable `i` tomará valores de 0 a 8 en cada iteración.

Línea 100: angle = 2 * math.pi * i / num_sides

- Dentro del bucle `for`: Calcula el ángulo (en radianes) para el vértice actual `i`. La fórmula `2 * pi * i / N` distribuye uniformemente los N vértices alrededor de un círculo.

Línea 101: x = center[0] + radius * math.cos(angle)

- Dentro del bucle `for`: Calcula la coordenada x del vértice actual. Comienza en la coordenada x del centro (`center[0]`) y añade el desplazamiento `radius * cos(angle)`.

Línea 102: y = center[1] + radius * math.sin(angle)

- Dentro del bucle `for`: Calcula la coordenada y del vértice actual de forma análoga, usando la coordenada y del centro (`center[1]`) y `radius * sin(angle)`.

Línea 103: polygon.append((int(x), int(y)))

- Dentro del bucle `for`: Convierte las coordenadas `x` e `y` calculadas (que pueden ser flotantes) a enteros usando `int()`. Crea una tupla `(int(x), int(y))` con las coordenadas del vértice y la añade a la lista `polygon`.

Línea 104: (En blanco)

- Línea en blanco: Separa la generación de vértices de la generación de normales.

Línea 105: # Asignar un vector normal a cada vértice.

- Comentario: Indica que se calcularán los vectores normales para cada vértice.

Línea 106: # Se utiliza una componente z no nula para simular una variación en la iluminación.

- Comentario: Explica por qué las normales tendrán un componente z: para que la iluminación varíe aunque los vértices estén en el plano XY.

Línea 107: vertex_normals = []

- Inicializa una lista vacía llamada `vertex_normals`. Esta lista almacenará los vectores normales asociados a cada vértice del polígono.

Línea 108: for i in range(num_sides):

- Inicia otro bucle `for` que itera `num_sides` (9) veces (de `i` = 0 a 8).

Línea 109: angle = 2 * math.pi * i / num_sides

- Dentro del bucle `for`: Recalcula el ángulo para el vértice actual `i`, igual que en el bucle anterior.

Línea 110: n = np.array([math.cos(angle), math.sin(angle), 0.5])

- Dentro del bucle `for`: Crea un vector normal como un array NumPy. Los componentes x e y (`math.cos(angle)`, `math.sin(angle)`) apuntan radialmente hacia afuera desde el centro (perpendicular al borde en ese punto si fuera un círculo). Se añade un componente z constante de `0.5`.

Línea 111: n = normalize(n)

- Dentro del bucle `for`: Llama a la función `normalize` para convertir el vector `n` en un vector unitario (longitud 1).

Línea 112: vertex_normals.append(n)

- Dentro del bucle `for`: Añade el vector normal normalizado `n` a la lista `vertex_normals`. Al final de este bucle, `vertex_normals` tendrá 9 vectores normales, uno por cada vértice.

Línea 113: (En blanco)

- Línea en blanco: Separa la generación de normales de la definición de colores.

Línea 114: # Asignar un color base a cada vértice (valores RGB en [0,255])

- Comentario: Indica que se definirán los colores base para cada vértice.

Línea 115: vertex_colors = [

- Inicia la definición de una lista llamada `vertex_colors`.

Línea 116: (255, 0, 0), # Rojo

- Define el primer elemento de la lista: una tupla `(255, 0, 0)` que representa el color rojo en RGB. El comentario identifica el color.

Línea 117: (255, 127, 0), # Naranja

- Define el segundo color (Naranja).

Línea 118: (255, 255, 0), # Amarillo

- Define el tercer color (Amarillo).

Línea 119: (0, 255, 0), # Verde

- Define el cuarto color (Verde).

Línea 120: (0, 255, 255), # Cian

- Define el quinto color (Cian).

Línea 121: (0, 0, 255), # Azul

- Define el sexto color (Azul).

Línea 122: (127, 0, 255), # Indigo

- Define el séptimo color (Índigo/Violeta).

Línea 123: (255, 0, 255), # Magenta

- Define el octavo color (Magenta).

Línea 124: (255, 0, 127) # Rosa

- Define el noveno y último color (Rosa).

Línea 125:]

- Cierra la definición de la lista `vertex_colors`. Ahora contiene 9 tuplas de colores RGB.

Línea 126: (En blanco)

- Línea en blanco: Separa la definición de colores del cálculo del centroide.

Línea 127: # Calcular el centroide del eneágono

- Comentario: Indica que se calculará la posición del punto central del polígono.

Línea 128: cx = sum(x for x, y in polygon) / len(polygon)

- Calcula la coordenada x promedio (`cx`) de todos los vértices en la lista `polygon`. Usa una expresión generadora `(x for x, y in polygon)` para obtener todas las coordenadas x, las suma con `sum()`, y divide por el número total de vértices (`len(polygon)`).

Línea 129: cy = sum(y for x, y in polygon) / len(polygon)

- Calcula la coordenada y promedio (`cy`) de manera análoga, sumando las coordenadas y.

Línea 130: centroid = (int(cx), int(cy))

- Crea una tupla `centroid` que contiene las coordenadas promedio `cx` y `cy` convertidas a enteros. Este es el punto central geométrico del polígono.

Línea 131: (En blanco)

- Línea en blanco: Separa el cálculo del centroide del cálculo de su normal.

Línea 132: # Calcular el vector normal del centroide como promedio de los vértices

- Comentario: Explica que la normal en el centroide se calculará promediando las normales de los vértices.

Línea 133: centroid_normal = np.mean(vertex_normals, axis=0)

- Calcula el vector normal promedio usando `np.mean`. Se le pasa la lista de arrays `vertex_normals`. `axis=0` indica que el promedio debe calcularse a lo largo del primer eje (es decir, promediar todos los componentes x juntos, todos los componentes y juntos, y todos los componentes z juntos). El resultado es un único vector (array NumPy) que representa la normal promedio, almacenado en `centroid_normal`.

Línea 134: centroid_normal = normalize(centroid_normal)

- Normaliza el vector `centroid_normal` recién calculado usando la función `normalize` para asegurar que tenga longitud unitaria.

Línea 135: (En blanco)

- Línea en blanco: Separa el cálculo de la normal del centroide del cálculo de su color.

Línea 136: # Calcular el color base del centroide como el promedio de los colores de los vértices

- Comentario: Explica que el color en el centroide se calculará promediando los colores de los vértices.

Línea 137: centroid_color = tuple(np.mean(vertex_colors, axis=0).astype(np.uint8))

- Calcula el color promedio del centroide. `np.mean(vertex_colors, axis=0)` calcula el promedio de cada componente (R, G, B) a través de todos los colores en `vertex_colors`. `.astype(np.uint8)` convierte los resultados promedio (que podrían ser flotantes) a enteros sin signo de 8 bits. `tuple(...)` convierte el array NumPy

resultante (con los promedios R, G, B) en una tupla. El color promedio se almacena en `centroid_color`.

Línea 138: (En blanco)

- Línea en blanco: Separa el cálculo del color del centroide de la definición de los parámetros de Phong.

Línea 139: # Parámetros para el modelo Phong

- Comentario: Indica que se definirán los parámetros globales de iluminación Phong.

Línea 140: light_dir = normalize(np.array([1, 1, 1])) # Dirección de la luz

- Define el vector de dirección de la luz como `[1, 1, 1]` (luz desde arriba a la derecha y desde el frente). Lo convierte en un array NumPy y lo normaliza usando la función `normalize`. El resultado se guarda en `light_dir`. El comentario aclara su propósito.

Línea 141: view_dir = normalize(np.array([0, 0, 1])) # Dirección de la cámara (vista frontal)

- Define el vector de dirección de la vista (desde dónde se mira) como `[0, 0, 1]` (mirando directamente a lo largo del eje Z positivo, perpendicular al plano XY de la imagen). Lo normaliza (aunque ya tiene longitud 1) y lo guarda en `view_dir`. El comentario aclara su propósito.

Línea 142: (En blanco)

- Línea en blanco: Separa los parámetros de Phong del bucle de triangulación.

Línea 143: # Triangular el eneágono usando el centroide (triangulación en abanico)

- Comentario: Explica que el siguiente bucle dividirá el eneágono en triángulos conectando cada par de vértices adyacentes con el centroide.

Línea 144: for i in range(num_sides):

- Inicia un bucle `for` que itera `num_sides` (9) veces, una vez por cada triángulo que formará el eneágono.

Línea 145: v1 = polygon[i]

- Dentro del bucle `for`: Obtiene el i-ésimo vértice de la lista `polygon` y lo asigna a `v1`. Este es el primer vértice del triángulo actual.

Línea 146: v2 = polygon[(i + 1) % num_sides]

- Dentro del bucle `for`: Obtiene el siguiente vértice en la lista `polygon`. Usa el operador módulo (`% num_sides`) para manejar el caso del último vértice (`i=8`), asegurando que el siguiente vértice sea el primero (`(8 + 1) % 9 = 0`). Lo asigna a `v2`. Este es el segundo vértice del triángulo actual.

Línea 147: n1 = vertex_normals[i]

- Dentro del bucle `for`: Obtiene la normal correspondiente al vértice `v1` (la i-ésima normal) de la lista `vertex_normals` y la asigna a `n1`.

Línea 148: n2 = vertex_normals[(i + 1) % num_sides]

- Dentro del bucle `for`: Obtiene la normal correspondiente al vértice `v2` (manejando el wrap-around con el módulo) y la asigna a `n2`.

Línea 149: c1 = vertex_colors[i]

- Dentro del bucle `for`: Obtiene el color base correspondiente al vértice `v1` (el i-ésimo color) de la lista `vertex_colors` y lo asigna a `c1`.

Línea 150: c2 = vertex_colors[(i + 1) % num_sides]

- Dentro del bucle `for`: Obtiene el color base correspondiente al vértice `v2` (manejando el wrap-around) y lo asigna a `c2`.

Línea 151: (En blanco)

- Dentro del bucle `for`: Línea en blanco para separar la obtención de datos de los vértices de la definición del triángulo.

Línea 152: # Definir el triángulo: dos vértices del eneágono y el centroide

- Dentro del bucle `for`: Comentario explicando cómo se forma el triángulo actual.

Línea 153: triangle = [v1, v2, centroid]

- Dentro del bucle `for`: Crea una lista `triangle` que contiene los tres vértices del triángulo actual: `v1`, `v2` y el `centroid` calculado anteriormente.

Línea 154: triangle_normals = [n1, n2, centroid_normal]

- Dentro del bucle `for`: Crea una lista `triangle_normals` con las normales correspondientes a los vértices del `triangle`: `n1`, `n2` y la `centroid_normal`.

Línea 155: triangle_colors = [c1, c2, centroid_color]

- Dentro del bucle `for`: Crea una lista `triangle_colors` con los colores base correspondientes a los vértices del `triangle`: `c1`, `c2` y el `centroid_color`.

Línea 156: triangle_phong_fill_color(img_array, triangle, triangle_normals, triangle_colors,

- Dentro del bucle `for`: Llama a la función `triangle_phong_fill_color` para rellenar el triángulo actual. Le pasa el `img_array` (que se modificará), las listas `triangle`, `triangle_normals`, `triangle_colors` recién creadas. La llamada continúa en la siguiente línea.

Línea 157: light_dir, view_dir,

- Dentro del bucle `for`: Continúa pasando argumentos a `triangle_phong_fill_color`: la dirección de la luz `light_dir` y la dirección de la vista `view_dir`. La llamada continúa en la siguiente línea.

Línea 158: ambient=0.1, diffuse_coef=0.8, specular_coef=0.5, shininess=20)

- Dentro del bucle `for`: Termina de pasar argumentos a `triangle_phong_fill_color`, especificando los parámetros del modelo Phong: `ambient`=0.1, `diffuse_coef`=0.8, `specular_coef`=0.5, `shininess`=20. Esta llamada procesará un triángulo del eneágono, coloreando los píxeles correspondientes en `img_array`.

Línea 159: (En blanco)

- Línea en blanco: Separa el final del bucle de triangulación de la conversión final de la imagen.

Línea 160: # Convertir el arreglo modificado de vuelta a una imagen PIL

- Comentario: Indica que el array NumPy con la imagen renderizada se convertirá de nuevo a un objeto imagen PIL.

Línea 161: img_out = Image.fromarray(img_array)

- Utiliza `Image.fromarray()` para crear un nuevo objeto de imagen PIL (`img_out`) a partir del contenido del array NumPy `img_array`, que ahora contiene el eneágono renderizado con sombreado Phong.

Línea 162: (En blanco)

- Línea en blanco: Separa la conversión de la imagen de la visualización.

Línea 163: # Visualizar el resultado

- Comentario: Indica que las siguientes líneas mostrarán la imagen generada usando Matplotlib.

Línea 164: plt.figure(figsize=(6, 6))

- Crea una nueva figura de Matplotlib para mostrar el gráfico. `figsize=(6, 6)` establece el tamaño de la figura en 6x6 pulgadas.

Línea 165: plt.imshow(img_out)

- Muestra la imagen contenida en el objeto PIL `img_out` dentro de la figura actual de Matplotlib. `imshow` es la función para mostrar datos como una imagen.

Línea 166: plt.axis('off')

- Oculta los ejes (marcas, números y líneas) alrededor de la imagen mostrada.

Línea 167: plt.title("Eneágono con Sombreado Phong en Color")

- Establece el título del gráfico de Matplotlib como la cadena especificada.

Línea 168: plt.show()

- Abre la ventana de Matplotlib y muestra la figura con la imagen renderizada y el título. La ejecución del script se detendrá aquí hasta que el usuario cierre la ventana del gráfico.

Resumen del Código

Este código genera una imagen 2D de un eneágono (polígono de 9 lados) coloreado utilizando el modelo de iluminación de Phong para simular un efecto 3D con sombreado suave.

1. **Preparación:** Importa las bibliotecas necesarias (numpy para cálculos, PIL para manejo de imágenes, math para

trigonometría, matplotlib para visualización). Define una función normalize para convertir vectores a longitud unitaria.

2. **Función de Relleno Phong (triangle_phong_fill_color):** Define la función principal que rellena un único triángulo. Esta función:
    - Toma como entrada los vértices 2D del triángulo, las normales 3D y colores base asociados a cada vértice, la dirección de la luz, la dirección de la vista y parámetros de iluminación (ambiente, difuso, especular, brillo).
    - Itera sobre cada píxel dentro de la caja envolvente del triángulo.
    - Usa coordenadas baricéntricas para determinar si el píxel está dentro del triángulo y para interpolar suavemente la normal y el color base a partir de los valores de los vértices.
    - Aplica el modelo de iluminación Phong en cada píxel interior, usando la normal interpolada, las direcciones de luz/vista y los coeficientes, para calcular las componentes de luz ambiente, difusa y especular.
    - Combina estas componentes con el color base interpolado para obtener el color final del píxel.
    - Modifica directamente un array NumPy que representa la imagen, asignando el color calculado al píxel.
3. **Generación del Eneágono:**
    - Define las dimensiones de la imagen y crea una imagen blanca inicial (representada como un array NumPy).
    - Calcula las coordenadas de los 9 vértices de un eneágono centrado en la imagen.
    - Asigna a cada vértice un vector normal (con un componente z>0 para simular profundidad) y un color base distinto (creando un degradado de colores alrededor del polígono).
    - Calcula la posición, normal promedio y color promedio del centroide del eneágono.
4. **Triangulación y Renderizado:**

- Divide el eneágono en 9 triángulos, cada uno formado por dos vértices adyacentes y el centroide (triangulación en abanico).
- Llama a la función triangle_phong_fill_color para cada uno de estos triángulos, pasándole los vértices, normales y colores correspondientes (incluyendo los del centroide). Esto rellena el img_array con el eneágono sombreado.

5. **Visualización:**
   - Convierte el array NumPy modificado de nuevo a un objeto de imagen PIL.
   - Utiliza matplotlib.pyplot para mostrar la imagen resultante en una ventana, con un título y sin ejes visibles.