Línea 1: import numpy as np

- Importa la biblioteca `numpy` y le asigna el alias `np`. Se utilizará para manejar la imagen como un arreglo numérico para la manipulación eficiente de píxeles.

Línea 2: from PIL import Image, ImageDraw

- Importa las clases `Image` e `ImageDraw` del módulo `PIL` (Python Imaging Library, comúnmente usada a través del fork `Pillow`).

- `Image` se usará para crear y manejar objetos de imagen.

- `ImageDraw` proporciona capacidades de dibujo 2D para objetos `Image` (aunque no se utiliza directamente en este código).

Línea 3: import math

- Importa el módulo `math` incorporado en Python. Proporciona acceso a funciones matemáticas como `sin`, `cos` y `pi`, que se usarán para calcular los vértices del octágono.

Línea 4: import matplotlib.pyplot as plt

- Importa el submódulo `pyplot` de la biblioteca `matplotlib` y le asigna el alias `plt`. Se utilizará para mostrar la imagen resultante en una ventana de gráfico.

Línea 5: (En blanco)

- Línea en blanco: Separa las importaciones de la definición de la función.

Línea 6: def triangle_gouraud_fill(img_array, verts, colors):

- Define una función llamada `triangle_gouraud_fill` que toma tres argumentos:

  - `img_array`: Se espera que sea un arreglo NumPy que representa la imagen a modificar.

  - `verts`: Se espera que sea una lista o tupla de 3 vértices que definen un triángulo.

- `colors`: Se espera que sea una lista o tupla de 3 colores, uno para cada vértice en `verts`.

Línea 7: """

- Inicio del docstring para la función `triangle_gouraud_fill`.

Línea 8: Rellena un triángulo definido por 'verts' (lista de 3 tuplas (x, y))

- Línea del docstring: Describe la acción principal de la función: rellenar un triángulo. Especifica el formato esperado para `verts`.

Línea 9: usando los colores en 'colors' (lista de 3 tuplas (R, G, B)) con interpolación

- Línea del docstring: Continúa la descripción, especificando el formato esperado para `colors` (tuplas RGB) y el método de relleno (interpolación).

Línea 10: Gouraud (basada en coordenadas baricéntricas).

- Línea del docstring: Nombra el algoritmo de interpolación específico (Sombreado Gouraud) y menciona que se basa en coordenadas baricéntricas.

Línea 11: (En blanco)

- Línea del docstring: Línea en blanco para mejorar la legibilidad.

Línea 12: Parámetros:

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

Línea 13: - img_array: arreglo NumPy de la imagen (modificable in situ).

- Línea del docstring: Describe el parámetro `img_array`, su tipo esperado (arreglo NumPy) y menciona que la función lo modificará directamente ("in situ").

Línea 14: - verts: lista de 3 vértices del triángulo.

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

Línea 15: - colors: lista de 3 colores correspondientes a cada vértice.

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

Línea 16: """

- Fin del docstring de la función.

Línea 17: # Extraer coordenadas x e y de los vértices

- Dentro de la función `triangle_gouraud_fill`: Comentario que explica el propósito de las siguientes líneas.

Línea 18: xs = [v[0] for v in verts]

- Dentro de la función `triangle_gouraud_fill`: Crea una lista `xs` extrayendo el primer elemento (coordenada x) de cada tupla `v` en la lista `verts`, usando una lista por comprensión.

Línea 19: ys = [v[1] for v in verts]

- Dentro de la función `triangle_gouraud_fill`: Crea una lista `ys` extrayendo el segundo elemento (coordenada y) de cada tupla `v` en la lista `verts`, usando una lista por comprensión.

Línea 20: # Calcular la caja envolvente del triángulo

- Dentro de la función `triangle_gouraud_fill`: Comentario que explica el propósito de las siguientes líneas.

Línea 21: min_x = max(min(xs), 0)

- Dentro de la función `triangle_gouraud_fill`: Calcula la coordenada x mínima de la caja envolvente. `min(xs)` encuentra la x más pequeña de los vértices. `max(..., 0)` asegura que `min_x` no sea menor que 0 (el borde izquierdo de la imagen).

Línea 22: max_x = min(max(xs), img_array.shape[1] - 1)

- Dentro de la función `triangle_gouraud_fill`: Calcula la coordenada x máxima de la caja envolvente. `max(xs)` encuentra la x más grande de los vértices. `img_array.shape[1]` da el ancho de la imagen. `min(..., img_array.shape[1] - 1)` asegura que `max_x` no exceda el índice x máximo válido de la imagen (ancho - 1).

Línea 23: min_y = max(min(ys), 0)

- Dentro de la función `triangle_gouraud_fill`: Calcula la coordenada y mínima de la caja envolvente, asegurando que no sea menor que 0 (borde superior de la imagen). Similar a `min_x`.

Línea 24: max_y = min(max(ys), img_array.shape[0] - 1)

- Dentro de la función `triangle_gouraud_fill`: Calcula la coordenada y máxima de la caja envolvente, asegurando que no exceda el índice y máximo válido de la imagen (alto - 1). `img_array.shape[0]` da la altura de la imagen. Similar a `max_x`.

Línea 25: (En blanco)

- Dentro de la función `triangle_gouraud_fill`: Línea en blanco para separación visual.

Línea 26: # Extraer vértices

- Dentro de la función `triangle_gouraud_fill`: Comentario que indica que se desempaquetarán las coordenadas de los vértices.

Línea 27: x0, y0 = verts[0]

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

Línea 28: x1, y1 = verts[1]

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

Línea 29: x2, y2 = verts[2]

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

Línea 30: (En blanco)

- Dentro de la función `triangle_gouraud_fill`: Línea en blanco para separación visual.

Línea 31: # Precalcular el denominador para las coordenadas baricéntricas

- Dentro de la función `triangle_gouraud_fill`: Comentario explicando el cálculo siguiente.

Línea 32: denom = ((y1 - y2) * (x0 - x2) + (x2 - x1) * (y0 - y2))

- Dentro de la función `triangle_gouraud_fill`: Calcula el denominador utilizado en las fórmulas de las coordenadas baricéntricas. Este valor está relacionado con el área del triángulo.

Línea 33: if denom == 0:

- Dentro de la función `triangle_gouraud_fill`: Comprueba si el denominador es cero.

Línea 34: return # El triángulo es degenerado

- Dentro de la función `triangle_gouraud_fill`: Si `denom` es cero, el triángulo es degenerado (una línea o un punto) y no se puede rellenar de forma significativa. La función termina prematuramente con `return`. El comentario aclara la razón. Pertenece al bloque `if`.

Línea 35: (En blanco)

- Dentro de la función `triangle_gouraud_fill`: Línea en blanco para separación visual.

Línea 36: # Recorrer cada píxel dentro de la caja envolvente

- Dentro de la función `triangle_gouraud_fill`: Comentario que explica los bucles anidados siguientes.

Línea 37: for y in range(int(min_y), int(max_y) + 1):

- Dentro de la función `triangle_gouraud_fill`: Inicia un bucle `for` que itera sobre cada coordenada `y` entera desde `min_y` hasta `max_y` (inclusive). `int()` convierte los límites (que podrían ser flotantes si los vértices lo fueran) a enteros. `+ 1` es necesario porque `range` excluye el límite superior.

Línea 38:   for x in range(int(min_x), int(max_x) + 1):

- Dentro de la función `triangle_gouraud_fill`: Inicia un bucle `for` anidado que itera sobre cada coordenada `x` entera desde `min_x` hasta `max_x` (inclusive) para la `y` actual. Pertenece al bloque del `for y`.

Línea 39:   # Calcular coordenadas baricéntricas

- Dentro de la función `triangle_gouraud_fill`: Comentario explicando los cálculos siguientes. Pertenece al bloque del `for x`.

Línea 40:   w0 = ((y1 - y2) * (x - x2) + (x2 - x1) * (y - y2)) / denom

- Dentro de la función `triangle_gouraud_fill`: Calcula la coordenada baricéntrica `w0` (peso asociado al vértice 0) para el píxel actual `(x, y)` usando la fórmula estándar y el denominador precalculado. Pertenece al bloque del `for x`.

Línea 41: w1 = ((y2 - y0) * (x - x2) + (x0 - x2) * (y - y2)) / denom

- Dentro de la función `triangle_gouraud_fill`: Calcula la coordenada baricéntrica `w1` (peso asociado al vértice 1) para el píxel actual `(x, y)`. Pertenece al bloque del `for x`.

Línea 42: w2 = 1 - w0 - w1

- Dentro de la función `triangle_gouraud_fill`: Calcula la coordenada baricéntrica `w2` (peso asociado al vértice 2) aprovechando que la suma de las coordenadas baricéntricas es 1 (`w0 + w1 + w2 = 1`). Pertenece al bloque del `for x`.

Línea 43: (En blanco)

- Dentro de la función `triangle_gouraud_fill`: Línea en blanco para separación dentro del bucle. Pertenece al bloque del `for x`.

Línea 44: # Si el píxel se encuentra dentro del triángulo

- Dentro de la función `triangle_gouraud_fill`: Comentario explicando la condición siguiente. Pertenece al bloque del `for x`.

Línea 45: if w0 >= 0 and w1 >= 0 and w2 >= 0:

- Dentro de la función `triangle_gouraud_fill`: Comprueba si el píxel `(x, y)` está dentro o en el borde del triángulo. Esto es cierto si todas las coordenadas baricéntricas son no negativas. Pertenece al bloque del `for x`.

Línea 46: # Interpolar el color según las coordenadas baricéntricas

- Dentro de la función `triangle_gouraud_fill`: Comentario explicando los cálculos de color siguientes. Pertenece al bloque `if`.

Línea 47: r = int(w0 * colors[0][0] + w1 * colors[1][0] + w2 * colors[2][0])

- Dentro de la función `triangle_gouraud_fill`: Calcula el componente Rojo (`r`) del color interpolado para el píxel. Multiplica el componente rojo de cada color de vértice (`colors[i][0]`) por su peso baricéntrico correspondiente (`wi`) y suma los resultados. `int()` convierte el resultado a un entero. Pertenece al bloque `if`.

Línea 48: g = int(w0 * colors[0][1] + w1 * colors[1][1] + w2 * colors[2][1])

- Dentro de la función `triangle_gouraud_fill`: Calcula el componente Verde (`g`) del color interpolado de forma análoga al componente rojo. Pertenece al bloque `if`.

Línea 49: b = int(w0 * colors[0][2] + w1 * colors[1][2] + w2 * colors[2][2])

- Dentro de la función `triangle_gouraud_fill`: Calcula el componente Azul (`b`) del color interpolado de forma análoga. Pertenece al bloque `if`.

Línea 50: img_array[y, x] = (r, g, b)

- Dentro de la función `triangle_gouraud_fill`: Asigna el color RGB interpolado `(r, g, b)` al píxel en la posición `(y, x)` del arreglo NumPy `img_array`. La indexación `[y, x]` es común en NumPy para (fila, columna). Pertenece al bloque `if`.

Línea 51: (En blanco)

- Línea en blanco: Separa la definición de la función del código principal.

Línea 52: # Dimensiones de la imagen

- Comentario que indica el propósito de la siguiente línea.

Línea 53: width, height = 400, 400

- Define las variables `width` (ancho) y `height` (alto) para la imagen y les asigna el valor 400 a ambas mediante desempaquetado de tupla.

Línea 54: (En blanco)

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

Línea 55: # Crear una imagen en blanco y obtener su arreglo NumPy

- Comentario que explica las dos líneas siguientes.

Línea 56: img = Image.new("RGB", (width, height), "white")

- Crea un nuevo objeto de imagen PIL (`img`) usando `Image.new()`.

  - `"RGB"`: Especifica el modo de color (Rojo, Verde, Azul).

  - `(width, height)`: Especifica las dimensiones de la imagen usando las variables definidas anteriormente.

  - `"white"`: Especifica el color de fondo inicial de la imagen.

Línea 57: img_array = np.array(img)

- Convierte el objeto de imagen PIL `img` en un arreglo NumPy `img_array`. Esto crea una representación de la imagen como una matriz 3D (alto x ancho x 3 canales de color) que puede ser manipulada eficientemente por NumPy y por la función `triangle_gouraud_fill`.

Línea 58: (En blanco)

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

Línea 59: # Parámetros del octágono (8 lados)

- Comentario que describe las variables relacionadas con el octágono.

Línea 60: num_sides = 8

- Define la variable `num_sides` y le asigna el valor 8, indicando que se va a dibujar un octágono.

Línea 61: radius = 150

- Define la variable `radius` (radio) y le asigna el valor 150. Este será el radio del círculo circunscrito al octágono.

Línea 62: center = (width // 2, height // 2)

- Calcula las coordenadas del centro de la imagen usando división entera (`//`) y las almacena en la tupla `center`.

Línea 63: octagon = []

- Inicializa una lista vacía llamada `octagon` que almacenará las coordenadas de los vértices del octágono.

Línea 64: (En blanco)

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

Línea 65: # Generar los vértices del octágono distribuidos uniformemente

- Comentario que explica el propósito del siguiente bucle `for`.

Línea 66: for i in range(num_sides):

- Inicia un bucle `for` que itera `num_sides` (8) veces, con la variable `i` tomando valores de 0 a 7.

Línea 67: angle = 2 * math.pi * i / num_sides

- Dentro del bucle `for`: Calcula el ángulo en radianes para el vértice actual `i`. Distribuye los vértices uniformemente alrededor de un círculo (2 * pi radianes) dividiendo por `num_sides`.

Línea 68: x = center[0] + radius * math.cos(angle)

- Dentro del bucle `for`: Calcula la coordenada x del vértice actual usando trigonometría. Se parte de la coordenada x del centro (`center[0]`) y se añade el desplazamiento horizontal (`radius * cos(angle)`).

Línea 69: y = center[1] + radius * math.sin(angle)

- Dentro del bucle `for`: Calcula la coordenada y del vértice actual de forma análoga, usando `sin(angle)` para el desplazamiento vertical.

Línea 70: octagon.append((int(x), int(y)))

- Dentro del bucle `for`: Añade las coordenadas calculadas `(x, y)` como una tupla a la lista `octagon`. Se convierten a enteros (`int()`) porque las coordenadas de los píxeles deben ser enteras.

Línea 71: (En blanco)

- Línea en blanco: Separa el cálculo de vértices de la definición de colores.

Línea 72: # Definir un color para cada vértice (por ejemplo, usando una gama de colores)

- Comentario que explica la siguiente estructura de datos.

Línea 73: vertex_colors = [

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

Línea 74: (255, 0, 0), # Rojo

- Define el primer color (para el primer vértice) como una tupla RGB (Rojo puro). El comentario indica el nombre del color.

Línea 75: (255, 127, 0), # Naranja

- Define el segundo color (Naranja).

Línea 76: (255, 255, 0), # Amarillo

- Define el tercer color (Amarillo).

Línea 77: (0, 255, 0), # Verde

- Define el cuarto color (Verde puro).

Línea 78: (0, 0, 255), # Azul

- Define el quinto color (Azul puro).

Línea 79: (75, 0, 130), # Índigo

- Define el sexto color (Índigo).

Línea 80: (148, 0, 211), # Violeta

- Define el séptimo color (Violeta).

Línea 81: (255, 192, 203) # Rosa

- Define el octavo color (Rosa).

Línea 82:]

- Cierra la definición de la lista `vertex_colors`. Ahora contiene 8 tuplas RGB, una para cada vértice del octágono.

Línea 83: (En blanco)

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

Línea 84: # Calcular el centroide del octágono (usado para triangulación en forma de abanico)

- Comentario que explica el propósito de las siguientes líneas y cómo se usará el centroide (para triangular el polígono).

Línea 85: cx = sum(x for x, y in octagon) / len(octagon)

- Calcula la coordenada x promedio (`cx`) de todos los vértices en la lista `octagon`. Utiliza una expresión generadora `(x for x, y in octagon)` para obtener todas las coordenadas x, `sum()` para sumarlas, y `len(octagon)` para obtener el número de vértices por el cual dividir.

Línea 86: cy = sum(y for x, y in octagon) / len(octagon)

- Calcula la coordenada y promedio (`cy`) de forma análoga, usando `(y for x, y in octagon)`.

Línea 87: centroid = (int(cx), int(cy))

- Crea la tupla `centroid` almacenando las coordenadas promedio calculadas, convertidas a enteros.

Línea 88: (En blanco)

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

Línea 89: # Calcular el color del centroide como el promedio de los colores de los vértices

- Comentario que explica el cálculo del color para el punto central.

Línea 90: centroid_color = tuple(

- Inicia la creación de una tupla `centroid_color` usando una expresión generadora dentro de la llamada a `tuple()`.

Línea 91: sum(color[i] for color in vertex_colors) // len(vertex_colors)

- Esta es la parte principal de la expresión generadora. `sum(color[i] for color in vertex_colors)` suma los componentes de color `i` (donde `i` será 0, 1 o 2) de todos los colores en `vertex_colors`. `// len(vertex_colors)` calcula el promedio de ese componente usando división entera.

Línea 92: for i in range(3)

- Esta es la parte `for` de la expresión generadora. Itera `i` sobre 0, 1 y 2 (para R, G y B). Para cada `i`, se ejecuta la expresión de la línea 91.

Línea 93:)

- Cierra la llamada a `tuple()`. `centroid_color` ahora contiene una tupla RGB que es el color promedio de todos los `vertex_colors`.

Línea 94: (En blanco)

- Línea en blanco: Separa el cálculo del color del centroide del bucle de triangulación y relleno.

Línea 95: # Triangular el octágono usando el centroide y rellenar cada triángulo con sombreado Gouraud

- Comentario que explica el propósito del siguiente bucle: dividir el octágono en triángulos (usando el centroide) y rellenar cada uno.

Línea 96: for i in range(num_sides):

- Inicia un bucle `for` que itera `num_sides` (8) veces, procesando cada lado del octágono para formar un triángulo con el centroide.

Línea 97: v1 = octagon[i]

- Dentro del bucle `for`: Obtiene el primer vértice (`v1`) del lado actual del octágono usando el índice `i`.

Línea 98: v2 = octagon[(i + 1) % num_sides]

- Dentro del bucle `for`: Obtiene el segundo vértice (`v2`) del lado actual. `(i + 1) % num_sides` asegura que cuando `i` es el último índice (7), `(i + 1)` se convierta en 0, conectando así el último vértice con el primero para cerrar el polígono.

Línea 99: c1 = vertex_colors[i]

- Dentro del bucle `for`: Obtiene el color `c1` correspondiente al vértice `v1`.

Línea 100: c2 = vertex_colors[(i + 1) % num_sides]

- Dentro del bucle `for`: Obtiene el color `c2` correspondiente al vértice `v2`, usando la misma lógica modular que para `v2`.

Línea 101: (En blanco)

- Dentro del bucle `for`: Línea en blanco para mejorar legibilidad.

Línea 102: # Definir el triángulo: dos vértices del octágono y el centroide

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

Línea 103: triangle = [v1, v2, centroid]

- Dentro del bucle `for`: Crea una lista `triangle` que contiene los tres vértices que forman el triángulo actual: los dos vértices del lado del octágono (`v1`, `v2`) y el `centroid`.

Línea 104: triangle_colors = [c1, c2, centroid_color]

- Dentro del bucle `for`: Crea una lista `triangle_colors` que contiene los colores correspondientes a los vértices en la lista `triangle`: `c1` para `v1`, `c2` para `v2`, y el `centroid_color` calculado previamente para el `centroid`.

Línea 105: triangle_gouraud_fill(img_array, triangle, triangle_colors)

- Dentro del bucle `for`: Llama a la función `triangle_gouraud_fill` definida anteriormente. Le pasa:
  - `img_array`: El arreglo NumPy de la imagen, que será modificado por la función.
  - `triangle`: La lista de vértices del triángulo actual.
  - `triangle_colors`: La lista de colores correspondientes a los vértices del triángulo.
- Esta llamada rellena el triángulo actual con el sombreado Gouraud.

Línea 106: (En blanco)

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

Línea 107: # Convertir el arreglo NumPy modificado de vuelta a una imagen PIL

- Comentario que explica la siguiente línea.

Línea 108: img_out = Image.fromarray(img_array)

- Convierte el arreglo NumPy `img_array` (que ahora contiene el octágono coloreado) de nuevo a un objeto de imagen PIL `img_out` usando `Image.fromarray()`.

Línea 109: (En blanco)

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

Línea 110: # Visualizar el resultado con Matplotlib

- Comentario que explica el bloque de código siguiente.

Línea 111: plt.figure(figsize=(6, 6))

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

Línea 112: plt.imshow(img_out)

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

Línea 113: plt.axis("off")

- Oculta los ejes (marcas de coordenadas y bordes) del gráfico de Matplotlib para mostrar solo la imagen.

Línea 114: plt.title("Octágono con Sombreado Gouraud")

- Establece el título de la ventana del gráfico.

Línea 115: plt.show()

- Abre la ventana de Matplotlib y muestra la figura con la imagen renderizada. La ejecución del script se pausará aquí hasta que la ventana se cierre.

Resumen del Código

Este código genera una imagen de un octágono centrado con un relleno de color suave utilizando el **sombreado Gouraud**.

1. **Inicialización:** Importa las bibliotecas necesarias (numpy, PIL, math, matplotlib), define las dimensiones de la imagen y crea una imagen PIL en blanco que luego convierte a un arreglo NumPy para facilitar la manipulación de píxeles.
2. **Cálculo del Octágono:** Calcula las coordenadas de los 8 vértices de un octágono regular inscrito en un círculo, utilizando funciones trigonométricas.
3. **Definición de Colores:** Asigna un color RGB distinto a cada uno de los 8 vértices del octágono.
4. **Cálculo del Centroide:** Calcula el punto central (centroide) del octágono promediando las coordenadas de sus vértices y también calcula un color promedio para este centroide.
5. **Relleno Gouraud:** Define una función triangle_gouraud_fill que implementa el sombreado Gouraud para un triángulo. Esta función:
    - Calcula la caja envolvente del triángulo.
    - Itera sobre cada píxel dentro de la caja.
    - Para cada píxel, calcula sus coordenadas baricéntricas respecto a los vértices del triángulo.
    - Si el píxel está dentro del triángulo (coordenadas baricéntricas >= 0), interpola linealmente los colores de los vértices usando las coordenadas baricéntricas como pesos.
    - Pinta el píxel en el arreglo NumPy de la imagen con el color interpolado.
6. **Triangulación y Aplicación:** Itera sobre los lados del octágono. Para cada lado, forma un triángulo conectando sus dos vértices con el centroide calculado. Llama a triangle_gouraud_fill para rellenar cada uno de estos triángulos, utilizando los colores de los vértices correspondientes y el color del centroide. Esto divide el octágono en 8 triángulos que lo rellenan completamente.
7. **Visualización:** Convierte el arreglo NumPy modificado (que ahora contiene el octágono sombreado) de nuevo a un objeto de imagen PIL y utiliza matplotlib para mostrar la imagen resultante en una ventana, con un título y sin ejes.

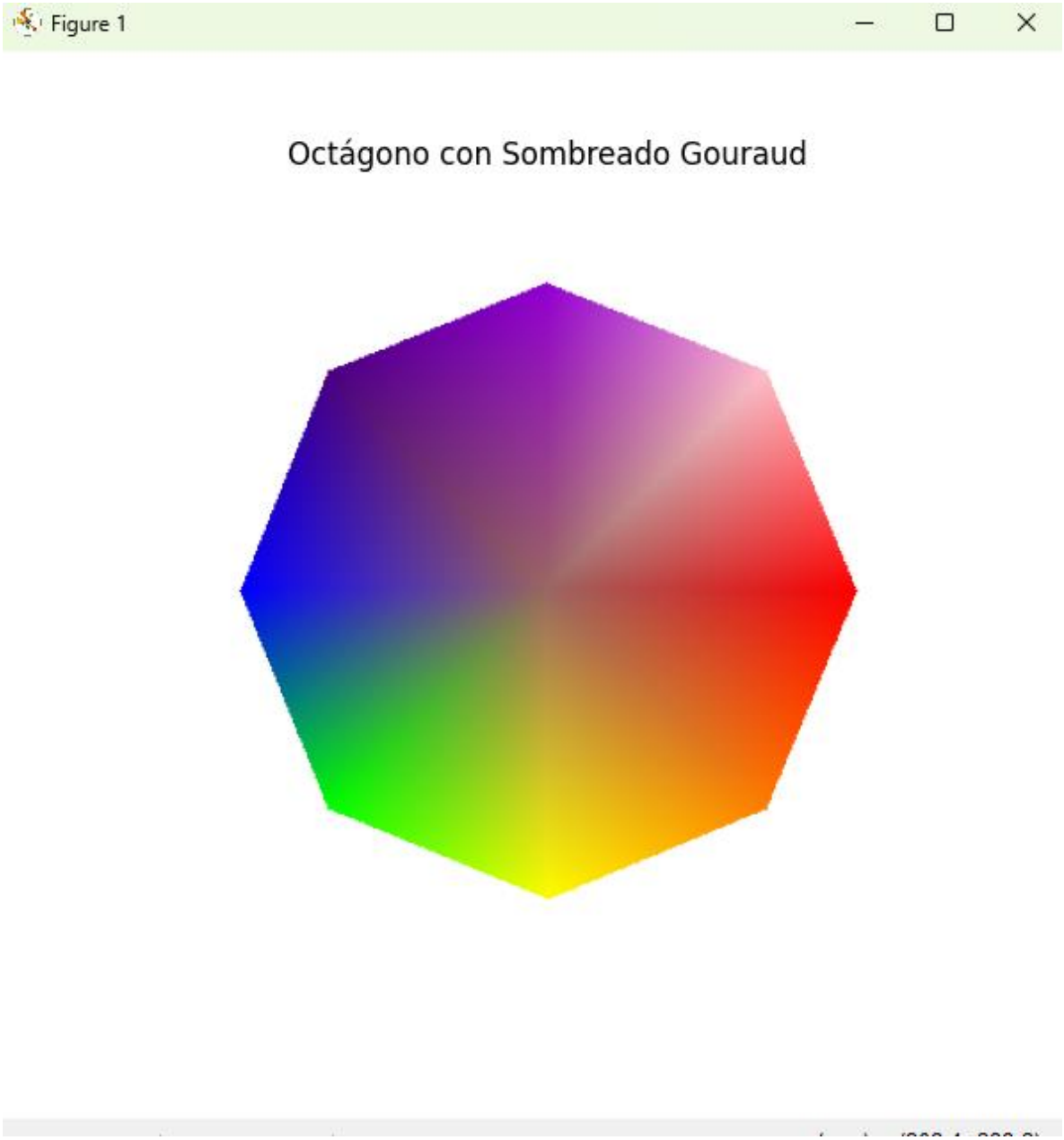


Figure 1
Octágono con Sombreado Gouraud

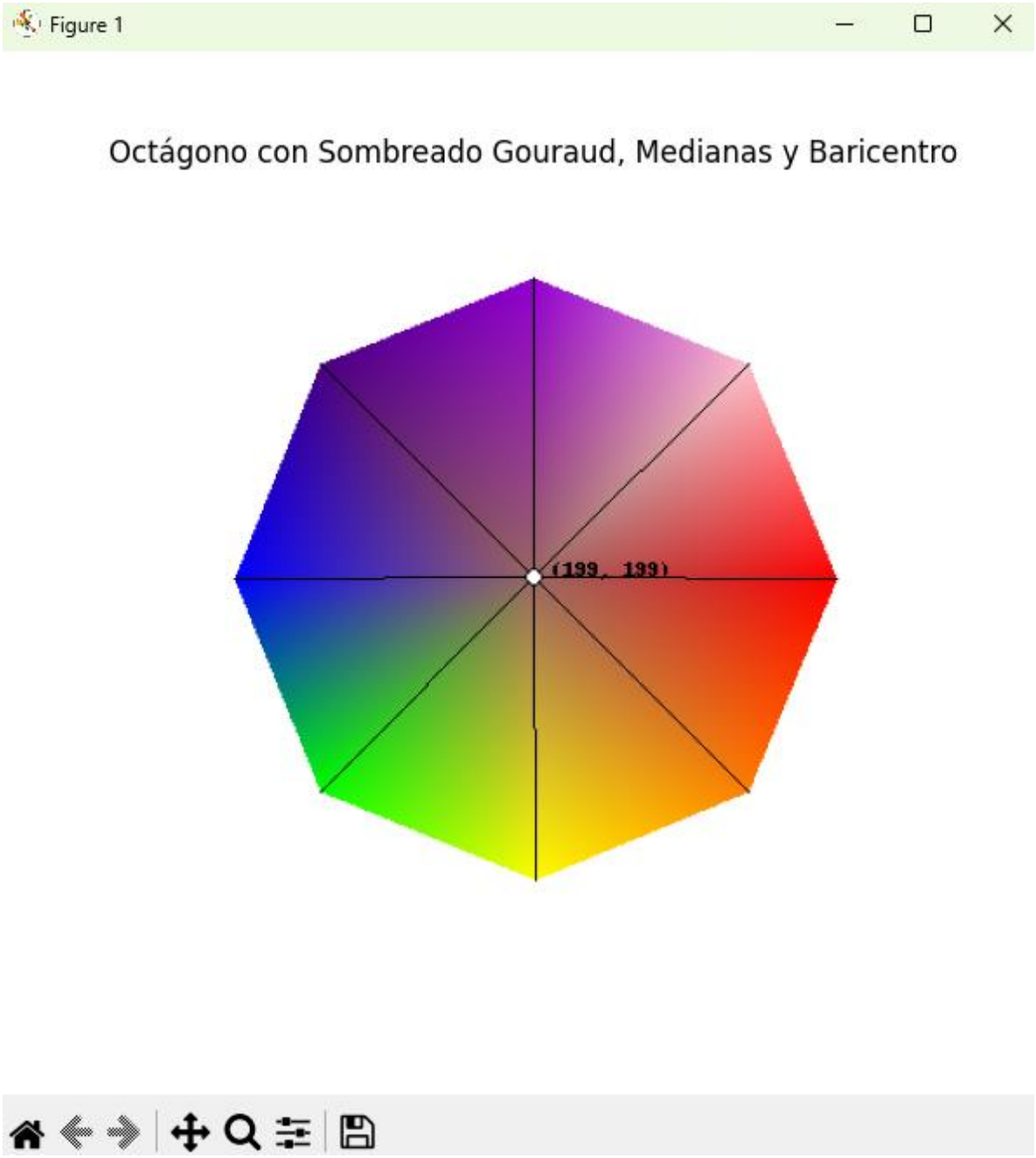
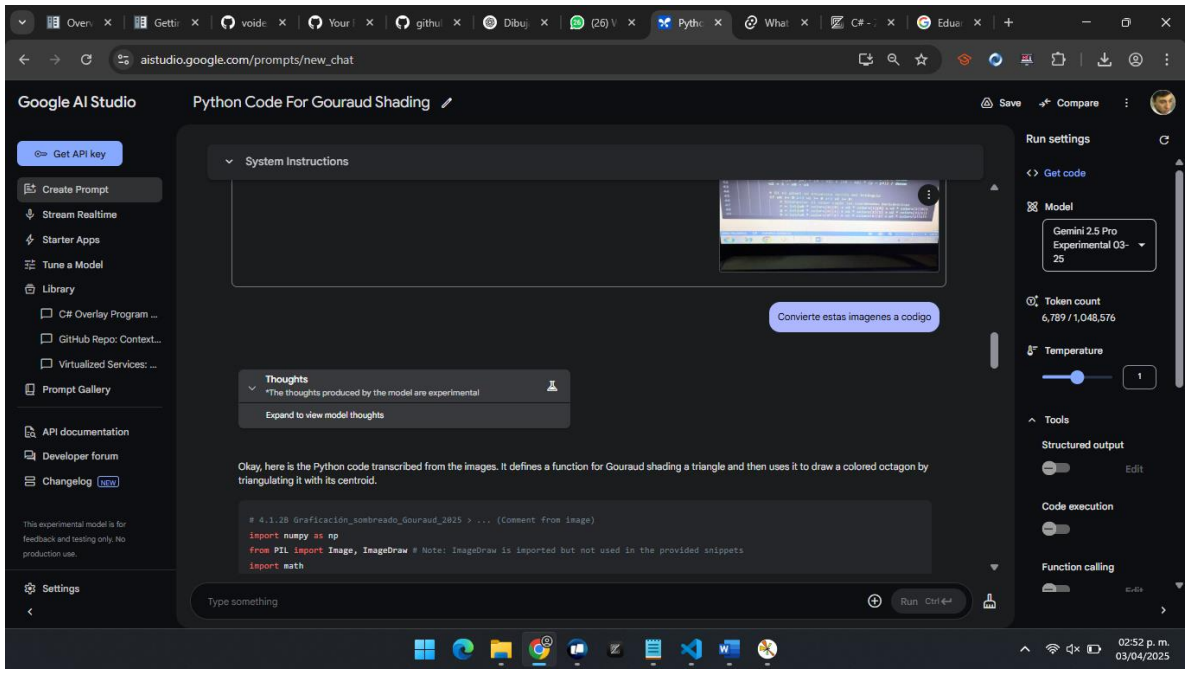
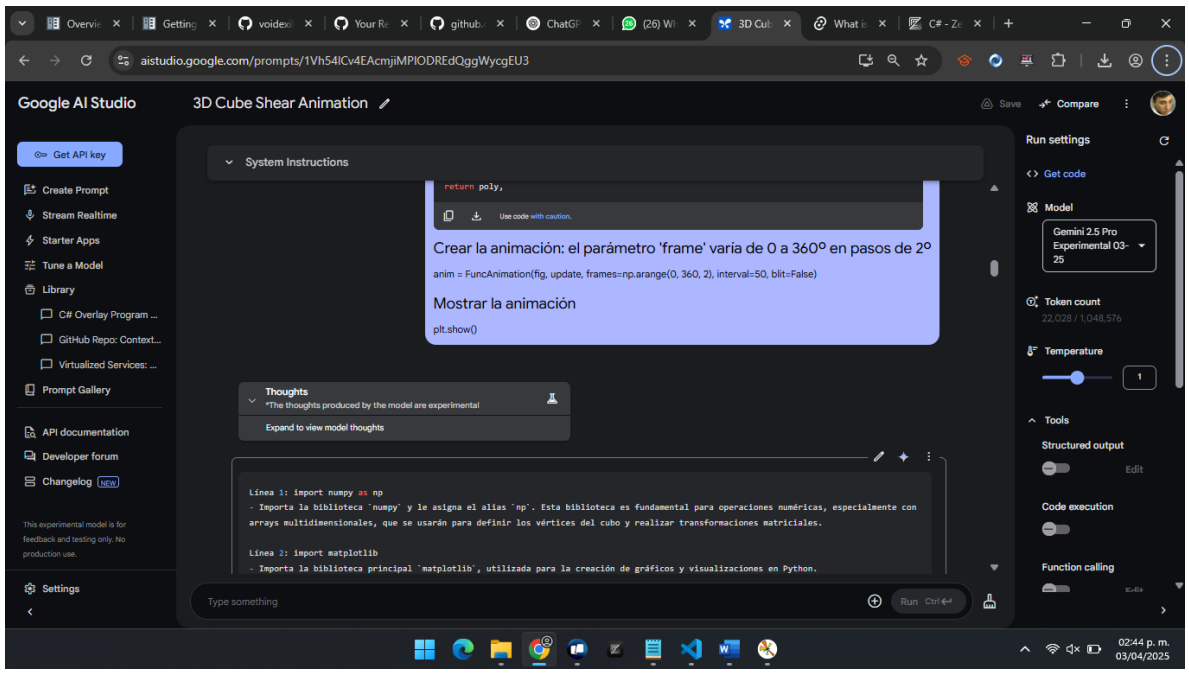


Figure 1
Octágono con Sombreado Gouraud, Medianas y Baricentro
(199, 199)


aistudio.google.com/prompts/new_chat
Google AI Studio
Python Code For Gouraud Shading
Save
Compare
Get API key
Create Prompt
Stream Realtime
Starter Apps
Tune a Model
Library
C# Overlay Program ...
GitHub Repo: Context...
Virtualized Services: ...
Prompt Gallery
API documentation
Developer forum
Changelog NEW
This experimental model is for feedback and testing only. No production use.
Settings
System Instructions
Convierte estas imagenes a codigo
Thoughts
*The thoughts produced by the model are experimental
Expand to view model thoughts
Okay, here is the Python code transcribed from the images. It defines a function for Gouraud shading a triangle and then uses it to draw a colored octagon by triangulating it with its centroid.
# 4.1.28 Graficación_sombreado_Gouraud_2025 > ... (Comment from image)
import numpy as np
from PIL import Image, ImageDraw # Note: ImageDraw is imported but not used in the provided snippets
import math
Type something
Run Ctrl↵
Run settings
Get code
Model
Gemini 2.5 Pro Experimental 03-25
Token count
6,789 / 1,048,576
Temperature
1
Tools
Structured output
Edit
Code execution
Function calling
02:52 p. m.
03/04/2025


aistudio.google.com/prompts/1Vh54ICv4EAcmjiMPlODREdQggWycgEU3
Google AI Studio
3D Cube Shear Animation
Save
Compare
Get API key
Create Prompt
Stream Realtime
Starter Apps
Tune a Model
Library
C# Overlay Program ...
GitHub Repo: Context...
Virtualized Services: ...
Prompt Gallery
API documentation
Developer forum
Changelog NEW
This experimental model is for feedback and testing only. No production use.
Settings
System Instructions
return poly,
Use code with caution.
Crear la animación: el parámetro 'frame' varía de 0 a 360° en pasos de 2°
anim = FuncAnimation(fig, update, frames=np.arange(0, 360, 2), interval=50, blit=False)
Mostrar la animación
plt.show()
Thoughts
*The thoughts produced by the model are experimental
Expand to view model thoughts
Línea 1: import numpy as np
- Importa la biblioteca `numpy` y le asigna el alias `np`. Esta biblioteca es fundamental para operaciones numéricas, especialmente con arrays multidimensionales, que se usarán para definir los vértices del cubo y realizar transformaciones matriciales.
Línea 2: import matplotlib
- Importa la biblioteca principal `matplotlib`, utilizada para la creación de gráficos y visualizaciones en Python.
Type something
Run Ctrl↵
Run settings
Get code
Model
Gemini 2.5 Pro Experimental 03-25
Token count
22,028 / 1,048,576
Temperature
1
Tools
Structured output
Edit
Code execution
Function calling
02:44 p. m.
03/04/2025