Aquí tienes el análisis línea por línea del código proporcionado:

Línea 1: import pygame

- Importa el módulo pygame, que es una biblioteca de Python utilizada para el desarrollo de videojuegos y aplicaciones multimedia. Es fundamental para crear ventanas gráficas, dibujar y manejar eventos.

Línea 2: import numpy as np

- Importa la biblioteca numpy y le asigna el alias np. NumPy es crucial para realizar operaciones numéricas eficientes con arrays multidimensionales, que se utilizarán para almacenar coordenadas, vectores, normales y realizar cálculos de iluminación.

Línea 3: (En blanco)

- Línea en blanco: Separa la sección de importaciones de la inicialización de Pygame, mejorando la legibilidad del código.

Línea 4: pygame.init()

- Llama a la función init() del módulo pygame. Esta función inicializa todos los módulos de Pygame necesarios para su correcto funcionamiento (como el sistema de video, audio, etc.).

Línea 5: screen = pygame.display.set_mode((600, 600))

- Llama a la función set_mode() del submódulo display de Pygame. Esta función crea la ventana principal de la aplicación, que es la superficie donde se dibujará todo.
- (600, 600): Define las dimensiones de la ventana en píxeles (600 píxeles de ancho por 600 píxeles de alto).
- El objeto Surface retornado (la ventana de la aplicación) se asigna a la variable screen.

Línea 6: pygame.display.set_caption("Pentágono con sombreado Phong")

- Llama a la función set_caption() del submódulo display de Pygame. Esta función establece el texto que aparecerá en la barra de título de la ventana de la aplicación.

Línea 7: clock = pygame.time.Clock()

- Crea una instancia de la clase Clock del submódulo time de Pygame. Este objeto clock se utilizará para controlar la velocidad de fotogramas (FPS) del bucle principal, asegurando una ejecución consistente.

Línea 8: (En blanco)

- Línea en blanco: Separa la inicialización de Pygame de la definición de los parámetros de iluminación y materiales.

Línea 9: # Configuración de iluminación y materiales

- Comentario que indica el inicio de la sección donde se definen las propiedades de la fuente de luz y cómo el material del objeto interactúa con ella.

Línea 10: light_pos = np.array([300, 300, 200])

- Define una variable light_pos y le asigna un array NumPy que representa las coordenadas (x, y, z) de la fuente de luz en el espacio 3D. Las coordenadas X e Y coinciden con el centro de la pantalla, y la Z está por encima.

Línea 11: light_color = np.array([1.0, 1.0, 1.0])

- Define una variable light_color y le asigna un array NumPy que representa el color de la luz emitida por la fuente (R, G, B), con valores flotantes entre 0.0 y 1.0. [1.0, 1.0, 1.0] significa luz blanca pura.

Línea 12: ambient = 0.1

- Define una variable ambient y le asigna el valor flotante 0.1. Este es el coeficiente de la componente de luz ambiental, una luz uniforme que ilumina todas las superficies de la escena, simulando la luz indirecta.

Línea 13: diffuse_strength = 0.7

- Define una variable diffuse_strength y le asigna el valor flotante 0.7. Este es el coeficiente de la componente difusa del material, que determina cuánta luz se dispersa uniformemente desde la superficie.

Línea 14: specular_strength = 0.6

- Define una variable specular_strength y le asigna el valor flotante 0.6. Este es el coeficiente de la componente especular del material, que controla la intensidad del brillo reflejado.

Línea 15: shininess = 32

- Define una variable shininess y le asigna el valor entero 32. Este es el exponente para el brillo especular (también conocido como "shininess"). Un valor más alto produce un brillo más pequeño y concentrado.

Línea 16: (En blanco)

- Línea en blanco: Separa la configuración de iluminación de la definición de la geometría.

Línea 17: # Geometría

- Comentario que indica el inicio de la sección donde se definen las propiedades geométricas del pentágono.

Línea 18: cx, cy = 300, 300

- Define las variables cx y cy como 300, que representan las coordenadas (x, y) del centro del pentágono en la pantalla.

Línea 19: radius = 200

- Define la variable radius como 200, que es el radio del círculo en el que se inscribe el pentágono.

Línea 20: num_vertices = 5

- Define la variable num_vertices como 5, especificando que se trata de un pentágono.

Línea 21: (En blanco)

- Línea en blanco: Separa los parámetros básicos de la geometría de la generación de las coordenadas y normales.

Línea 22: # Coordenadas del pentágono (2D) y sus normales (simples en Z+)

- Comentario que describe la finalidad de las siguientes líneas: generar las coordenadas 2D y 3D de los vértices del pentágono, así como sus vectores normales.

Línea 23: angles = np.linspace(-np.pi / 2, 3 * np.pi / 2, num_vertices, endpoint=False)

- Genera un array NumPy angles que contiene 5 ángulos espaciados uniformemente en un rango de 2π radianes (un círculo completo), empezando desde -π/2 (para que el primer vértice quede arriba) y excluyendo el punto final.

Línea 24: vertices_2d = np.stack([

- Comienza la definición de vertices_2d, un array NumPy que contendrá las coordenadas 2D (x, y) de cada vértice del pentágono.

Línea 25: cx + radius * np.cos(angles),

- Calcula las coordenadas X de cada vértice: cx más el radio multiplicado por el coseno de cada ángulo.

Línea 26: cy + radius * np.sin(angles)

- Calcula las coordenadas Y de cada vértice: cy más el radio multiplicado por el seno de cada ángulo.

Línea 27:], axis=-1)

- Cierra la llamada a np.stack. axis=-1 apila los arrays de X e Y a lo largo de un nuevo último eje, resultando en vertices_2d de forma (5, 2).

Líín 28: vertices_3d = np.hstack([vertices_2d, np.zeros((num_vertices, 1))])

- Crea vertices_3d a partir de vertices_2d.
- np.zeros((num_vertices, 1)): Crea un array de 5 filas y 1 columna lleno de ceros. Esta es la coordenada Z.
- np.hstack([...]): Apila horizontalmente vertices_2d (forma (5, 2)) y el array de ceros (forma (5, 1)). El resultado vertices_3d es un array de forma (5, 3), donde cada fila es [x, y, 0], colocando el pentágono en el plano XY (Z=0).

Línea 29: normals = np.tile(np.array([0, 0, 1]), (num_vertices, 1))

- Define los vectores normales para cada vértice del pentágono.
- np.array([0, 0, 1]): Es un vector normal que apunta directamente "fuera de la pantalla" (a lo largo del eje Z positivo).
- np.tile(..., (num_vertices, 1)): Repite este vector 5 veces (una por cada vértice). Esto simplifica el ejemplo asumiendo que todo el pentágono es plano y mira hacia el observador. normals tendrá forma (5, 3).

Línea 30: # todos apuntan hacia fuera de la pantalla

- Comentario que aclara la dirección de los vectores normales definidos en la línea anterior.

Línea 31: (En blanco)

- Línea en blanco: Separa la definición de los vértices y normales del centro.

Línea 32: # Centro

- Comentario que indica que las siguientes líneas definen el centro del pentágono en 3D y su normal.

Línea 33: center = np.array([cx, cy, 0])

- Define un array NumPy center que representa las coordenadas 3D del centro del pentágono en el plano XY (Z=0).

Línea 34: center_normal = np.array([0, 0, 1])

- Define la normal del centro del pentágono. Al igual que los vértices del perímetro, se asume que apunta hacia fuera de la pantalla.

Línea 35: (En blanco)

- Línea en blanco: Separa la definición de las propiedades del centro de la función de coordenadas baricéntricas.

Línea 36: # Barycentric interpolation

- Comentario que indica el inicio de la sección que define la función para calcular coordenadas baricéntricas.

Línea 37: def barycentric_coords(p, a, b, c):

- Define una función llamada barycentric_coords que toma un punto p (2D) y tres vértices de triángulo a, b, c (que pueden ser 3D, pero solo se usarán sus componentes XY).

Línea 38: v0 = b[:2] - a[:2]

- Dentro de la función barycentric_coords: Calcula el vector 2D v0 restando las componentes X e Y de a de las de b. (Se usa [:2] porque a, b, c son 3D, pero los cálculos baricéntricos se hacen en el plano 2D de la pantalla).

Línea 39: v1 = c[:2] - a[:2]

- Dentro de la función barycentric_coords: Calcula el vector 2D v1 restando las componentes X e Y de a de las de c.

Línea 40: v2 = p - a[:2]

- Dentro de la función barycentric_coords: Calcula el vector 2D v2 restando las componentes X e Y de a de las de p.

Línea 41: d00 = np.dot(v0, v0)

- Dentro de la función barycentric_coords: Calcula el producto punto de v0 consigo mismo (v0 · v0).

Línea 42: d01 = np.dot(v0, v1)

- Dentro de la función barycentric_coords: Calcula el producto punto de v0 y v1 (v0 · v1).

Línea 43: d11 = np.dot(v1, v1)

- Dentro de la función barycentric_coords: Calcula el producto punto de v1 consigo mismo (v1 · v1).

Línea 44: d20 = np.dot(v2, v0)

- Dentro de la función barycentric_coords: Calcula el producto punto de v2 y v0 (v2 · v0).

Línea 45: d21 = np.dot(v2, v1)

- Dentro de la función barycentric_coords: Calcula el producto punto de v2 y v1 (v2 · v1).

Línea 46: denom = d00 * d11 - d01 * d01

- Dentro de la función barycentric_coords: Calcula el denominador para las coordenadas baricéntricas. Esto representa el doble del área del triángulo en 2D.

Línea 47: if denom == 0:

- Dentro de la función barycentric_coords: Condicional que verifica si el denominador es cero. Si es cero, el triángulo es degenerado (colineal).

Línea 48: return -1, -1, -1

- Dentro de la función barycentric_coords: Si el triángulo es degenerado, la función retorna (-1, -1, -1) como valores inválidos para las coordenadas baricéntricas.

Línea 49: v = (d11 * d20 - d01 * d21) / denom

- Dentro de la función barycentric_coords: Calcula la coordenada baricéntrica v (peso del vértice b).

Línea 50: w = (d00 * d21 - d01 * d20) / denom

- Dentro de la función barycentric_coords: Calcula la coordenada baricéntrica w (peso del vértice c).

Línea 51: u = 1 - v - w

- Dentro de la función barycentric_coords: Calcula la coordenada baricéntrica u (peso del vértice a), utilizando la propiedad de que la suma de las tres coordenadas debe ser 1.

Línea 52: return u, v, w

- Dentro de la función barycentric_coords: Retorna los tres valores u, v, w como una tupla.

Línea 53: (En blanco)

- Línea en blanco: Separa la definición de la función barycentric_coords de la función de cálculo de iluminación Phong.

Línea 54: # Phong lighting per pixel

- Comentario que indica el inicio de la sección que define la función para calcular el color de un píxel usando el modelo de iluminación de Phong.

Línea 55: def compute_phong(normal, position):

- Define una función llamada compute_phong que acepta dos parámetros: normal (el vector normal 3D interpolado para el píxel) y position (la posición 3D interpolada del píxel).

Línea 56: normal = normal / np.linalg.norm(normal)

- Dentro de la función compute_phong: Normaliza el vector normal recibido. Esto asegura que su longitud sea 1, lo cual es crucial para los cálculos de iluminación.

Línea 57: light_dir = light_pos - position

- Dentro de la función compute_phong: Calcula el vector light_dir (dirección de la luz) restando la position del píxel de la light_pos de la fuente de luz.

Línea 58: light_dir = light_dir / np.linalg.norm(light_dir)

- Dentro de la función compute_phong: Normaliza el vector light_dir para que su longitud sea 1.

Línea 59: view_dir = np.array([0, 0, 1])

- Dentro de la función compute_phong: Define el vector view_dir (dirección de la vista u observador). En este ejemplo, se asume que el observador siempre mira a lo largo del eje Z positivo, desde el infinito (o, de forma simplificada, es una proyección ortográfica con la cámara en Z+).

Línea 60: (En blanco)

- Dentro de la función compute_phong: Línea en blanco para separar las definiciones de direcciones de los cálculos de componentes de iluminación.

Línea 61: # Ambient

- Dentro de la función compute_phong: Comentario que indica el cálculo de la componente ambiental.

Línea 62: ambient_color = ambient * light_color

- Dentro de la función compute_phong: Calcula el color de la componente ambiental. Es la intensidad ambiental (ambient) multiplicada por el color de la luz (light_color).

Línea 63: (En blanco)

- Dentro de la función compute_phong: Línea en blanco para separar el componente ambiental del difuso.

Línea 64: # Diffuse

- Dentro de la función compute_phong: Comentario que indica el cálculo de la componente difusa.

Línea 65: diff = max(np.dot(normal, light_dir), 0)

- Dentro de la función compute_phong: Calcula la intensidad de la luz difusa según la ley de Lambert. Es el producto punto del vector normal y light_dir, asegurándose de que el resultado no sea negativo (se toma el máximo entre 0 y el producto punto).

Línea 66: diffuse_color = diffuse_strength * diff * light_color

- Dentro de la función compute_phong: Calcula el color de la componente difusa. Es la intensidad difusa del material (diffuse_strength) multiplicada por la intensidad calculada (diff) y por el color de la luz.

Línea 67: (En blanco)

- Dentro de la función compute_phong: Línea en blanco para separar el componente difuso del especular.

Línea 68: # Specular

- Dentro de la función compute_phong: Comentario que indica el cálculo de la componente especular.

Línea 69: reflect_dir = 2 * np.dot(normal, light_dir) * normal - light_dir

- Dentro de la función compute_phong: Calcula el vector de reflexión (reflect_dir) utilizando la fórmula estándar: 2 * (N·L) * N - L, donde N es la normal y L es la dirección de la luz.

Línea 70: spec = np.power(max(np.dot(view_dir, reflect_dir), 0), shininess)

- Dentro de la función compute_phong: Calcula la intensidad del brillo especular. Es el producto punto del vector view_dir y reflect_dir, clampeado a 0, y luego elevado a la potencia shininess.

Línea 71: specular_color = specular_strength * spec * light_color

- Dentro de la función compute_phong: Calcula el color de la componente especular. Es la intensidad especular del material (specular_strength) multiplicada por la intensidad calculada (spec) y por el color de la luz.

Línea 72: (En blanco)

- Dentro de la función compute_phong: Línea en blanco para separar los cálculos de componentes de la combinación final.

Línea 73: result = ambient_color + diffuse_color + specular_color

- Dentro de la función compute_phong: Suma las tres componentes de color (ambiental, difusa, especular) para obtener el color final del píxel.

Línea 74: return np.clip(result * 255, 0, 255).astype(int)

- Dentro de la función compute_phong: Retorna el color final, ajustado a valores de 0 a 255 y convertido a enteros.

- result * 255: Escala los valores de color de [0, 1] a [0, 255].
- np.clip(..., 0, 255): Limita los valores de color a este rango, por si la suma excediera 255.
- .astype(int): Convierte los valores flotantes a enteros, que es el formato esperado por Pygame para los colores.

Línea 75: (En blanco)

- Línea en blanco: Separa la función de cálculo de Phong de la función de dibujo de triángulos.

Línea 76: # Dibuja un triángulo con sombreado Phong

- Comentario que indica el inicio de la sección que define la función para rasterizar y sombrear un triángulo usando el modelo Phong.

Línea 77: def draw_phong_triangle(v1, v2, v3, n1, n2, n3):

- Define una función draw_phong_triangle que acepta los tres vértices 3D del triángulo (v1, v2, v3) y los tres vectores normales 3D asociados a esos vértices (n1, n2, n3).

Línea 78: min_x = int(min(v1[0], v2[0], v3[0]))

- Dentro de la función draw_phong_triangle: Calcula la coordenada X mínima del bounding box del triángulo, tomando el mínimo de las coordenadas X de los vértices 2D proyectados (se usa v[0] para X).

Línea 79: max_x = int(max(v1[0], v2[0], v3[0]))

- Dentro de la función draw_phong_triangle: Calcula la coordenada X máxima del bounding box.

Línea 80: min_y = int(min(v1[1], v2[1], v3[1]))

- Dentro de la función draw_phong_triangle: Calcula la coordenada Y mínima del bounding box, tomando el mínimo de las coordenadas Y de los vértices 2D proyectados (se usa v[1] para Y).

Línea 81: max_y = int(max(v1[1], v2[1], v3[1]))

- Dentro de la función draw_phong_triangle: Calcula la coordenada Y máxima del bounding box.

Línea 82: (En blanco)

- Dentro de la función draw_phong_triangle: Línea en blanco para separar el cálculo del bounding box del bucle de píxeles.

Línea 83: for x in range(min_x, max_x + 1):

- Dentro de la función draw_phong_triangle: Inicia el bucle exterior for, que iterará sobre las coordenadas x de los píxeles dentro del bounding box (desde min_x hasta max_x, inclusive).

Línea 84: for y in range(min_y, max_y + 1):

- Dentro del bucle for x: Inicia el bucle interior for, que iterará sobre las coordenadas y de los píxeles (desde min_y hasta max_y, inclusive). Cada par (x, y) representa un píxel a considerar.

Línea 85: p = np.array([x, y])

- Dentro del bucle anidado: Crea un array NumPy p que representa las coordenadas 2D del píxel actual [x, y].

Línea 86: u, v, w = barycentric_coords(p, v1, v2, v3)

- Dentro del bucle anidado: Llama a la función barycentric_coords para obtener las coordenadas baricéntricas del píxel p con respecto al triángulo (v1, v2, v3). Se usan solo las componentes XY de v1, v2, v3 para el cálculo.

Línea 87: if u >= 0 and v >= 0 and w >= 0:

- Dentro del bucle anidado: Condicional que verifica si el píxel p está dentro o en el borde del triángulo. Esto es True si y solo si todas las coordenadas baricéntricas (u, v, w) son mayores o iguales a cero.

Línea 88: interpolated_normal = u * n1 + v * n2 + w * n3

- Dentro del bloque if: Calcula el vector normal 3D interpolado para el píxel actual. Esto se hace mediante una combinación lineal de las normales de los vértices (n1, n2, n3), ponderada por las coordenadas baricéntricas (u, v, w). **Este es el paso clave del sombreado Phong.**

Línea 89: interpolated_pos = u * v1 + v * v2 + w * v3

- Dentro del bloque if: Calcula la posición 3D interpolada del píxel actual. Esto se hace de manera similar, combinando las posiciones 3D de los vértices (v1, v2, v3) con las coordenadas baricéntricas. Aunque v1, v2, v3 tienen Z=0 en este ejemplo, esta interpolación de posición es crucial para que el vector de

luz y vista se calculen correctamente por píxel si el objeto tuviera profundidad o estuviera en 3D real.

Línea 90: color = compute_phong(interpolated_normal, interpolated_pos)

- Dentro del bloque if: Llama a la función compute_phong para calcular el color RGB final del píxel. Se le pasa la interpolated_normal y la interpolated_pos del píxel. Esto significa que el cálculo de iluminación se realiza *por píxel* (a diferencia del sombreado Gouraud, donde se interpola el color ya calculado).

Línea 91: screen.set_at((x, y), color)

- Dentro del bloque if: Dibuja el píxel individual en la superficie de la pantalla de Pygame en las coordenadas (x, y) con el color calculado.

Línea 92: (En blanco)

- Línea en blanco: Separa la definición de la función de dibujo de triángulos del bucle principal del programa.

Línea 93: # Main loop

- Comentario que indica el inicio del bucle principal del programa, que gestiona el renderizado y los eventos.

Línea 94: running = True

- Define una variable booleana running y la inicializa en True. Esta variable controla la ejecución del bucle principal.

Línea 95: while running:

- Inicia un bucle while que continuará ejecutándose mientras running sea True.

Línea 96: screen.fill((0, 0, 0))

- Dentro del bucle while: Rellena toda la superficie de la pantalla con el color negro (0, 0, 0). Esto borra el contenido del fotograma anterior.

Línea 97: (En blanco)

- Dentro del bucle while: Línea en blanco para separar la limpieza de la pantalla del bucle de dibujo del pentágono.

Línea 98: for i in range(num_vertices):

- Dentro del bucle while: Inicia un bucle for que iterará num_vertices (5) veces, una por cada segmento del pentágono.

Línea 99: v1 = center

- Dentro del bucle for i: Asigna la posición 3D del center a v1.

Línea 100: v2 = vertices_3d[i]

- Dentro del bucle for i: Asigna la posición 3D del vértice i del pentágono a v2.

Línea 101: v3 = vertices_3d[(i + 1) % num_vertices]

- Dentro del bucle for i: Asigna la posición 3D del vértice siguiente (envolviendo al primero si es el último) a v3.
- Así, cada iteración define un triángulo que va del centro a dos vértices adyacentes del perímetro del pentágono.

Línea 102: n1 = center_normal

- Dentro del bucle for i: Asigna la normal del center a n1.

Línea 103: n2 = normals[i]

- Dentro del bucle for i: Asigna la normal del vértice i a n2.

Línea 104: n3 = normals[(i + 1) % num_vertices]

- Dentro del bucle for i: Asigna la normal del siguiente vértice a n3.

Línea 105: (En blanco)

- Dentro del bucle for i: Línea en blanco.

Línea 106: draw_phong_triangle(v1, v2, v3, n1, n2, n3)

- Dentro del bucle for i: Llama a la función draw_phong_triangle para rasterizar y sombrear el triángulo actual con sus vértices 3D y normales asociadas.

Línea 107: (En blanco)

- Dentro del bucle while: Línea en blanco para separar el bucle de dibujo de la actualización de la pantalla.

Línea 108: pygame.display.flip()

- Dentro del bucle while: Actualiza toda la ventana de Pygame para mostrar los gráficos que se han dibujado en la superficie screen.

Línea 109: (En blanco)

- Dentro del bucle while: Línea en blanco para separar la actualización de la pantalla del manejo de eventos.

Línea 110: for event in pygame.event.get():

- Dentro del bucle while: Itera sobre todos los eventos que Pygame ha detectado (clics de ratón, pulsaciones de teclado, cierre de ventana, etc.).

Línea 111: if event.type == pygame.QUIT:

- Dentro del bucle for event: Condicional que verifica si el evento actual es del tipo pygame.QUIT, que se genera cuando el usuario intenta cerrar la ventana.

Línea 112: running = False

- Dentro del bloque if: Si el evento es pygame.QUIT, se establece la variable running en False, lo que provocará que el bucle principal finalice.

Línea 113: (En blanco)

- Dentro del bucle while: Línea en blanco para separar el manejo de eventos del control de velocidad.

Línea 114: clock.tick(60)

- Dentro del bucle while: Limita la velocidad de fotogramas del bucle principal a 60 FPS. Esto asegura que la aplicación no consuma CPU innecesariamente y se ejecute a una velocidad consistente.

Línea 115: (En blanco)

- Línea en blanco: Separa el bucle principal de la finalización de Pygame.

Línea 116: pygame.quit()

- Llama a la función quit() del módulo pygame. Esta función desinicializa los módulos de Pygame y libera los recursos del sistema antes de que el programa termine.

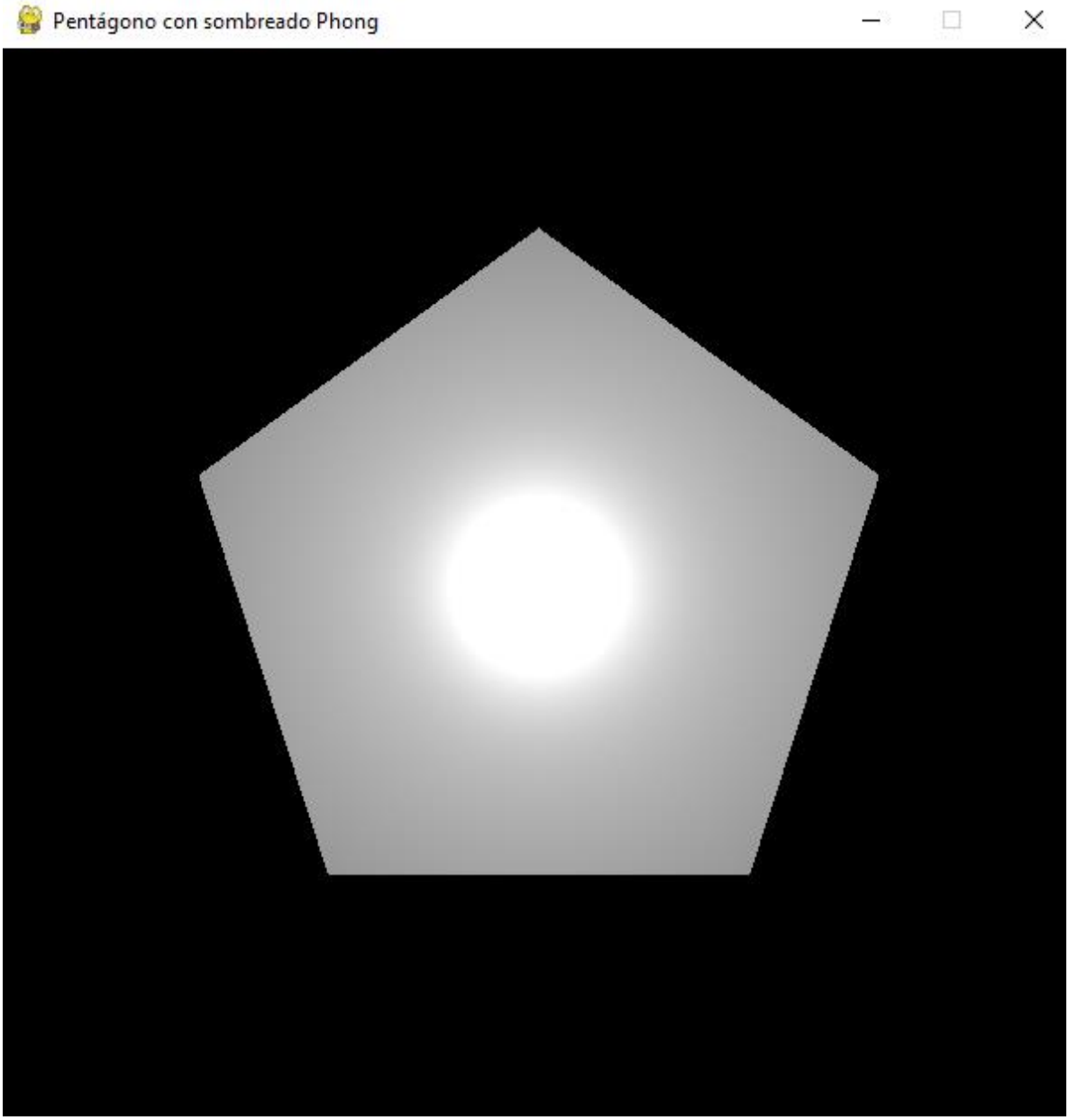
Resumen del Código

Este script de Python implementa un renderizador básico que dibuja un pentágono en 2D utilizando Pygame y aplica un sombreado por píxel basado en el modelo de iluminación de **Phong**.

1. **Inicialización:** Configura Pygame, crea una ventana de 600x600 píxeles y un objeto Clock para controlar la velocidad de fotogramas.
2. **Parámetros de Iluminación y Materiales:** Define las propiedades de una fuente de luz (posición 3D, color) y los coeficientes del material del objeto (ambiental, difuso, especular y el exponente de brillo o shininess).
3. **Geometría del Pentágono:**
    - Calcula las coordenadas 2D de los 5 vértices de un pentágono regular centrado en la pantalla.
    - Convierte estas coordenadas 2D a 3D, colocándolas en el plano Z=0.
    - Define vectores normales para cada vértice y para el centro del pentágono, simplificando que todos apuntan directamente "hacia afuera de la pantalla" (eje Z positivo).
4. **Función de Coordenadas Baricéntricas:** Incluye una función barycentric_coords para calcular las coordenadas baricéntricas de un punto 2D dentro de un triángulo. Estas coordenadas son cruciales para interpolar atributos a través de la superficie del triángulo.
5. **Función de Sombreado Phong (compute_phong):**
    - Esta es la parte central del modelo Phong. Toma un vector normal 3D y una posición 3D (ambos típicamente interpolados para un píxel específico).
    - Calcula las direcciones normalizadas hacia la luz y hacia el observador.
    - Determina las contribuciones de tres componentes de iluminación:
        - **Ambiental:** Un color base uniforme en todo el objeto.
        - **Difuso:** Depende del ángulo entre la normal de la superficie y la dirección de la luz (cuanto más perpendicular la luz, más intensa).
        - **Especular:** Crea un "brillo" reflejado que depende del ángulo entre la dirección de la vista y el vector de reflexión de la luz.
    - Suma estas tres componentes para obtener el color RGB final del píxel.
    - Escala y recorta el color a un rango de 0-255 y lo convierte a enteros.

6. **Función de Dibujo de Triángulos Phong (draw_phong_triangle):**
    - Toma los 3 vértices 3D de un triángulo y sus 3 normales 3D asociadas.
    - Calcula un "bounding box" 2D para el triángulo en la pantalla.
    - Itera sobre cada píxel dentro de este bounding box.
    - Para cada píxel:
        - Calcula sus coordenadas baricéntricas.
        - Si el píxel está dentro del triángulo, **interpola los vectores normales 3D y las posiciones 3D** de los vértices usando las coordenadas baricéntricas.
        - Llama a compute_phong con estas **normales y posiciones interpoladas** para obtener el color del píxel.
        - Dibuja el píxel en la pantalla.
7. **Bucle Principal:**
    - En cada fotograma, borra la pantalla.
    - Divide el pentágono en 5 triángulos (cada uno formado por el centro y dos vértices adyacentes del perímetro).
    - Para cada uno de estos 5 triángulos, llama a draw_phong_triangle para dibujarlo con sombreado Phong.
    - Actualiza la pantalla de Pygame.
    - Maneja eventos (como cerrar la ventana).
    - Limita la velocidad de fotogramas a 60 FPS.

El resultado es un pentágono dibujado en la ventana de Pygame con un sombreado que simula una fuente de luz y reflejos especulares, dando una apariencia mucho más realista y tridimensional que el sombreado Gouraud, ya que los cálculos de iluminación se realizan individualmente para cada píxel.


Pentágono con sombreado Phong