```python
import numpy as np
import matplotlib.pyplot as plt
import networkx as nx
from matplotlib.animation import FuncAnimation  # Unused in this code, but kept
from collections import deque
import random  # Unused in this code, but kept

class GraphVisualizer:
    """
    A comprehensive class for visualizing:
    1. Six Degrees of Separation
    2. Fractals (Sierpinski Triangle, Koch Snowflake, Tree)
    3. Mandelbrot Set
    """

    def __init__(self):
        pass  # Removed unused self.fig, self.ax

    # ==================== SIX DEGREES OF SEPARATION ====================

    def create_small_world_network(self, n=100, k=6, p=0.1):
        """
        Create a Watts-Strogatz small-world network
        n: number of nodes
        k: each node connected to k nearest neighbors
        p: probability of rewiring
        """
        G = nx.watts_strogatz_graph(n, k, p)
        return G

    def visualize_six_degrees(self, start_node=0, target_node=50):
        """
        Visualize the six degrees of separation concept
        """
        # Create small-world network
        G = self.create_small_world_network(n=100, k=6, p=0.1)

        # Validate nodes
        if start_node not in G or target_node not in G:
            raise ValueError("Start or target node not in graph")

        # Find shortest path
        try:
            path = nx.shortest_path(G, start_node, target_node)
            path_length = len(path) - 1
        except nx.NetworkXNoPath:
            print("No path exists between nodes")
            return

        # Create figure
        fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 7))

        # Left plot: Full network with highlighted path
        pos = nx.spring_layout(G, seed=42)

        # Draw all nodes and edges
        nx.draw_networkx_edges(G, pos, alpha=0.1, ax=ax1)
        nx.draw_networkx_nodes(G, pos, node_color='lightblue',
                              node_size=100, ax=ax1)

        # Highlight the path
        path_edges = list(zip(path[:-1], path[1:]))
        nx.draw_networkx_edges(G, pos, edgelist=path_edges,
                              edge_color='red', width=3, ax=ax1)
        nx.draw_networkx_nodes(G, pos, nodelist=path,
                              node_color='red', node_size=300, ax=ax1)

        # Highlight start and end
        nx.draw_networkx_nodes(G, pos, nodelist=[start_node],
                              node_color='green', node_size=500, ax=ax1)
        nx.draw_networkx_nodes(G, pos, nodelist=[target_node],
                              node_color='orange', node_size=500, ax=ax1)

        ax1.set_title(f'Six Degrees of Separation\nPath Length: {path_length} degrees',
                     fontsize=14, fontweight='bold')
        ax1.axis('off')

        # Right plot: BFS visualization showing degree levels
        self._visualize_bfs_levels(G, start_node, ax2)

        plt.tight_layout()
        plt.savefig('six_degrees_separation.png', dpi=300, bbox_inches='tight')
        plt.show()

        # Print statistics
        self._print_network_stats(G, path_length)

    def _visualize_bfs_levels(self, G, start_node, ax):
        """
        Visualize BFS levels from start node
        """
        # Perform BFS to get levels
        levels = {start_node: 0}
        queue = deque([start_node])

        while queue:
            node = queue.popleft()
            for neighbor in G.neighbors(node):
                if neighbor not in levels:
                    levels[neighbor] = levels[node] + 1
                    queue.append(neighbor)

        # Create radial layout for better level visualization
        pos = nx.spring_layout(G, seed=42)  # Kept for consistency; could switch to nx.shell_layout for radial

        # Color nodes by level
        max_level = max(levels.values()) if levels else 0
        colors = [levels.get(node, 0) for node in G.nodes()]

        nx.draw_networkx_edges(G, pos, alpha=0.2, ax=ax)
        nodes = nx.draw_networkx_nodes(G, pos, node_color=colors,
                                       cmap=plt.cm.viridis,
                                       node_size=100, ax=ax)

        # Add colorbar
        plt.colorbar(nodes, ax=ax, label='Degrees of Separation')

        ax.set_title(f'BFS Levels from Node {start_node}\nMax Separation: {max_level} degrees',
                    fontsize=12, fontweight='bold')
        ax.axis('off')

    def _print_network_stats(self, G, path_length):
        """
        Print network statistics
        """
        avg_path_length = nx.average_shortest_path_length(G)
        clustering = nx.average_clustering(G)

        print("\n" + "="*50)
        print("NETWORK STATISTICS")
        print("="*50)
        print(f"Number of nodes: {G.number_of_nodes()}")
        print(f"Number of edges: {G.number_of_edges()}")
        print(f"Sample path length: {path_length} degrees")
        print(f"Average path length: {avg_path_length:.2f}")
        print(f"Clustering coefficient: {clustering:.3f}")
        print(f"Network diameter: {nx.diameter(G)}")  # Note: Expensive for large graphs
        print("="*50 + "\n")

    # ==================== FRACTALS ====================

    def draw_sierpinski_triangle(self, order=5):
        """
        Draw Sierpinski Triangle fractal
        """
        fig, ax = plt.subplots(figsize=(10, 10))

        def sierpinski(ax, order, points):
            if order == 0:
                triangle = plt.Polygon(points, fill=True,
                                      edgecolor='blue', facecolor='cyan', alpha=0.7)
                ax.add_patch(triangle)
            else:
                # Calculate midpoints
                mid1 = [(points[0][0] + points[1][0])/2,
                       (points[0][1] + points[1][1])/2]
                mid2 = [(points[1][0] + points[2][0])/2,
                       (points[1][1] + points[2][1])/2]
                mid3 = [(points[2][0] + points[0][0])/2,
                       (points[2][1] + points[0][1])/2]

                # Recursively draw three smaller triangles
                sierpinski(ax, order-1, [points[0], mid1, mid3])
                sierpinski(ax, order-1, [mid1, points[1], mid2])
                sierpinski(ax, order-1, [mid3, mid2, points[2]])

        # Initial triangle points
        points = [[0, 0], [1, 0], [0.5, np.sqrt(3)/2]]
        sierpinski(ax, order, points)

        ax.set_aspect('equal')
        ax.set_title(f'Sierpinski Triangle (Order {order})',
                    fontsize=16, fontweight='bold')
        ax.axis('off')

        plt.tight_layout()
        plt.savefig('sierpinski_triangle.png', dpi=300, bbox_inches='tight')
        plt.show()

    def draw_koch_snowflake(self, order=4):
        """
        Draw Koch Snowflake fractal
        """
        fig, ax = plt.subplots(figsize=(10, 10))

        def koch_curve(p1, p2, order):
            if order == 0:
                return [p1, p2]
            else:
                # Divide line into three parts
                dx = p2[0] - p1[0]
                dy = p2[1] - p1[1]

                p3 = [p1[0] + dx/3, p1[1] + dy/3]
                p5 = [p1[0] + 2*dx/3, p1[1] + 2*dy/3]

                # Calculate peak point
                angle = np.pi/3
                p4 = [p3[0] + (p5[0]-p3[0])*np.cos(angle) - (p5[1]-p3[1])*np.sin(angle),
                      p3[1] + (p5[0]-p3[0])*np.sin(angle) + (p5[1]-p3[1])*np.cos(angle)]

                # Recursively generate curve
                curve = []
                curve.extend(koch_curve(p1, p3, order-1)[:-1])
                curve.extend(koch_curve(p3, p4, order-1)[:-1])
                curve.extend(koch_curve(p4, p5, order-1)[:-1])
                curve.extend(koch_curve(p5, p2, order-1))

                return curve

        # Create initial triangle
        size = 1
        p1 = [0, 0]
        p2 = [size, 0]
        p3 = [size/2, size*np.sqrt(3)/2]

        # Generate Koch snowflake
        points = []
        points.extend(koch_curve(p1, p2, order)[:-1])
        points.extend(koch_curve(p2, p3, order)[:-1])
        points.extend(koch_curve(p3, p1, order)[:-1])

        # Plot
        points = np.array(points)
        ax.plot(points[:, 0], points[:, 1], 'b-', linewidth=0.5)
        ax.fill(points[:, 0], points[:, 1], 'cyan', alpha=0.5)

        ax.set_aspect('equal')
        ax.set_title(f'Koch Snowflake (Order {order})',
                     fontsize=16, fontweight='bold')
        ax.axis('off')

        plt.tight_layout()
        plt.savefig('koch_snowflake.png', dpi=300, bbox_inches='tight')
        plt.show()

    def draw_fractal_tree(self, order=10, angle=30):
        """
        Draw a fractal tree
        """
        fig, ax = plt.subplots(figsize=(10, 12))

        def draw_branch(x, y, length, angle_deg, order):
            if order > 0:
                # Calculate end point
                angle_rad = np.radians(angle_deg)
                x_end = x + length * np.cos(angle_rad)
                y_end = y + length * np.sin(angle_rad)

                # Draw branch with color based on order
                color = plt.cm.YlGn(order / 10)
                width = order * 0.5
                ax.plot([x, x_end], [y, y_end], color=color, linewidth=width)

                # Recursively draw smaller branches (use passed angle)
                new_length = length * 0.7
                draw_branch(x_end, y_end, new_length, angle_deg + angle, order - 1)
                draw_branch(x_end, y_end, new_length, angle_deg - angle, order - 1)

        # Start drawing from bottom center
        draw_branch(0, 0, 1, 90, order)

        ax.set_aspect('equal')
        ax.set_xlim(-2, 2)  # Added for better framing
        ax.set_ylim(0, 3)
        ax.set_title(f'Fractal Tree (Order {order}, Angle {angle}°)',
                     fontsize=16, fontweight='bold')
        ax.axis('off')

        plt.tight_layout()
        plt.savefig('fractal_tree.png', dpi=300, bbox_inches='tight')
        plt.show()

    # ==================== MANDELBROT SET ====================

    def draw_mandelbrot(self, width=800, height=800, max_iter=100,
                        xmin=-2.5, xmax=1.5, ymin=-2, ymax=2):
        """
        Draw the Mandelbrot Set
        """
        # Create coordinate arrays
        x = np.linspace(xmin, xmax, width)
        y = np.linspace(ymin, ymax, height)
        X, Y = np.meshgrid(x, y)
        C = X + 1j * Y

        # Initialize output array
        mandelbrot_set = np.zeros((height, width))

        # Calculate Mandelbrot set (fixed iteration counting)
        Z = np.zeros_like(C)
        for i in range(max_iter):
            mask = np.abs(Z) <= 2
            Z[mask] = Z[mask]**2 + C[mask]
            mandelbrot_set[mask] += 1  # Increment for each bounded iteration

        # Create figure
        fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 7))

        # Plot 1: Standard Mandelbrot
        im1 = ax1.imshow(mandelbrot_set, extent=[xmin, xmax, ymin, ymax],
                         cmap='hot', interpolation='bilinear', origin='lower')
        ax1.set_title('Mandelbrot Set', fontsize=16, fontweight='bold')
        ax1.set_xlabel('Real axis')
        ax1.set_ylabel('Imaginary axis')
        plt.colorbar(im1, ax=ax1, label='Iterations to divergence')

        # Plot 2: Zoomed region (interesting area)
        zoom_xmin, zoom_xmax = -0.8, -0.4
        zoom_ymin, zoom_ymax = -0.2, 0.2

        x_zoom = np.linspace(zoom_xmin, zoom_xmax, width)
        y_zoom = np.linspace(zoom_ymin, zoom_ymax, height)
        X_zoom, Y_zoom = np.meshgrid(x_zoom, y_zoom)
        C_zoom = X_zoom + 1j * Y_zoom

        mandelbrot_zoom = np.zeros((height, width))
        Z_zoom = np.zeros_like(C_zoom)

        for i in range(max_iter):
            mask = np.abs(Z_zoom) <= 2
            Z_zoom[mask] = Z_zoom[mask]**2 + C_zoom[mask]
            mandelbrot_zoom[mask] += 1  # Fixed iteration counting here too

        im2 = ax2.imshow(mandelbrot_zoom, extent=[zoom_xmin, zoom_xmax, zoom_ymin, zoom_ymax],
                        cmap='twilight', interpolation='bilinear', origin='lower')
        ax2.set_title('Mandelbrot Set (Zoomed)', fontsize=16, fontweight='bold')
        ax2.set_xlabel('Real axis')
        ax2.set_ylabel('Imaginary axis')
        plt.colorbar(im2, ax=ax2, label='Iterations to divergence')

        plt.tight_layout()
        plt.savefig('mandelbrot_set.png', dpi=300, bbox_inches='tight')
        plt.show()

    def draw_julia_set(self, c=-0.7+0.27015j, width=800, height=800,
                      max_iter=100, xmin=-2, xmax=2, ymin=-2, ymax=2):
        """
        Draw the Julia Set for a given complex constant c
        """
        # Create coordinate arrays
        x = np.linspace(xmin, xmax, width)
        y = np.linspace(ymin, ymax, height)
        X, Y = np.meshgrid(x, y)
        Z = X + 1j * Y

        # Initialize output array
        julia_set = np.zeros((height, width))

        # Calculate Julia set (fixed iteration counting)
        for i in range(max_iter):
            mask = np.abs(Z) <= 2
            Z[mask] = Z[mask]**2 + c
            julia_set[mask] += 1  # Increment for each bounded iteration

        # Create figure
        fig, ax = plt.subplots(figsize=(10, 10))

        im = ax.imshow(julia_set, extent=[xmin, xmax, ymin, ymax],
                       cmap='hot', interpolation='bilinear', origin='lower')
        ax.set_title(f'Julia Set (c = {c})', fontsize=16, fontweight='bold')
        ax.set_xlabel('Real axis')
        ax.set_ylabel('Imaginary axis')
        plt.colorbar(im, ax=ax, label='Iterations to divergence')

        plt.tight_layout()
        plt.savefig('julia_set.png', dpi=300, bbox_inches='tight')
        plt.show()

    def run_examples(self):
        """
        Run example visualizations for all methods
        """
        print("Running Six Degrees of Separation...")
        self.visualize_six_degrees()

        print("Running Sierpinski Triangle...")
        self.draw_sierpinski_triangle()

        print("Running Koch Snowflake...")
        self.draw_koch_snowflake()

        print("Running Fractal Tree...")
        self.draw_fractal_tree()

        print("Running Mandelbrot Set...")
        self.draw_mandelbrot()

        print("Running Julia Set...")
        self.draw_julia_set()

# Example usage
if __name__ == "__main__":
    gv = GraphVisualizer()
    gv.run_examples()

```