```python
import numpy as np
from scipy.spatial.transform import Rotation
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

class SpatialTracker:
    def __init__(self):
        # Position in 3D space (x, y, z)
        self.position = np.zeros(3)

        # Orientation as a rotation object (more robust than Euler angles)
        self.rotation = Rotation.identity()

        # Alternative: Store as quaternion
        self.quaternion = np.array([1.0, 0.0, 0.0, 0.0])  # [w, x, y, z]

        # Optional: Store Euler angles for easy access (roll, pitch, yaw)
        self.orientation = np.zeros(3)

        # History for plotting
        self.position_history = [self.position.copy()]
        self.orientation_history = [self.orientation.copy()]
        self.quaternion_history = [self.quaternion.copy()]
        self.time_history = [0.0]

    def update_motion(self, linear_velocity, angular_velocity, dt):
        """
        Update 6-DOF motion given linear and angular velocities

        Parameters:
        -----------
        linear_velocity : array-like, shape (3,)
            Linear velocity in m/s [vx, vy, vz]
        angular_velocity : array-like, shape (3,)
            Angular velocity in rad/s [wx, wy, wz]
        dt : float
            Time step in seconds
        """
        linear_velocity = np.array(linear_velocity)
        angular_velocity = np.array(angular_velocity)

        # Update position (linear motion)
        self.position += linear_velocity * dt

        # Update orientation (angular motion)
        angle = np.linalg.norm(angular_velocity) * dt
        if angle > 1e-10:  # Avoid division by zero
            axis = angular_velocity / np.linalg.norm(angular_velocity)
            delta_rotation = Rotation.from_rotvec(axis * angle)
            self.rotation = delta_rotation * self.rotation

        # Update quaternion representation
        self.quaternion = self.rotation.as_quat()  # Returns [x, y, z, w] format
        # Convert to [w, x, y, z] format
        self.quaternion = np.array([self.quaternion[3], self.quaternion[0],
                                    self.quaternion[1], self.quaternion[2]])

        # Update Euler angles (roll, pitch, yaw)
        self.orientation = self.rotation.as_euler('xyz', degrees=False)

        # Store history
        self.position_history.append(self.position.copy())
        self.orientation_history.append(self.orientation.copy())
        self.quaternion_history.append(self.quaternion.copy())
        self.time_history.append(self.time_history[-1] + dt)

    def get_state(self):
```

```python
        """Get the complete 6-DOF state"""
        return {
            'position': self.position.copy(),
            'orientation_euler': self.orientation.copy(),
            'orientation_euler_deg': np.degrees(self.orientation),
            'quaternion': self.quaternion.copy(),
            'rotation_matrix': self.rotation.as_matrix()
        }

    def plot_trajectory_3d(self, show_orientation=True, orientation_interval=10):
        """Plot 3D trajectory with orientation frames"""
        fig = plt.figure(figsize=(12, 10))
        ax = fig.add_subplot(111, projection='3d')

        # Convert history to arrays
        positions = np.array(self.position_history)

        # Plot trajectory
        ax.plot(positions[:, 0], positions[:, 1], positions[:, 2],
                'b-', linewidth=2, label='Trajectory')

        # Mark start and end
        ax.scatter(*positions[0], color='green', s=100, marker='o', label='Start')
        ax.scatter(*positions[-1], color='red', s=100, marker='s', label='End')

        # Show orientation frames
        if show_orientation:
            for i in range(0, len(self.position_history), orientation_interval):
                pos = self.position_history[i]
                rot = Rotation.from_euler('xyz', self.orientation_history[i])
                R = rot.as_matrix()

                # Draw coordinate frame (RGB = XYZ)
                scale = 0.5
                colors = ['r', 'g', 'b']
                for j in range(3):
                    direction = R[:, j] * scale
                    ax.quiver(pos[0], pos[1], pos[2],
                             direction[0], direction[1], direction[2],
                             color=colors[j], arrow_length_ratio=0.3, linewidth=1.5)

        ax.set_xlabel('X (m)')
        ax.set_ylabel('Y (m)')
        ax.set_zlabel('Z (m)')
        ax.set_title('6-DOF Trajectory in 3D Space')
        ax.legend()
        ax.grid(True)

        # Equal aspect ratio
        max_range = np.array([positions[:, 0].max()-positions[:, 0].min(),
                              positions[:, 1].max()-positions[:, 1].min(),
                              positions[:, 2].max()-positions[:, 2].min()]).max() / 2.0
        mid_x = (positions[:, 0].max()+positions[:, 0].min()) * 0.5
        mid_y = (positions[:, 1].max()+positions[:, 1].min()) * 0.5
        mid_z = (positions[:, 2].max()+positions[:, 2].min()) * 0.5
        ax.set_xlim(mid_x - max_range, mid_x + max_range)
        ax.set_ylim(mid_y - max_range, mid_y + max_range)
        ax.set_zlim(mid_z - max_range, mid_z + max_range)

        plt.tight_layout()
        return fig

    def plot_position_vs_time(self):
        """Plot position components vs time"""
        fig, axes = plt.subplots(3, 1, figsize=(12, 8))

        positions = np.array(self.position_history)
```

```python
        times = np.array(self.time_history)

        labels = ['X Position', 'Y Position', 'Z Position']
        colors = ['r', 'g', 'b']

        for i, (ax, label, color) in enumerate(zip(axes, labels, colors)):
            ax.plot(times, positions[:, i], color=color, linewidth=2)
            ax.set_ylabel(f'{label} (m)')
            ax.grid(True, alpha=0.3)
            ax.set_title(label)

        axes[-1].set_xlabel('Time (s)')
        plt.suptitle('Position vs Time', fontsize=14, fontweight='bold')
        plt.tight_layout()
        return fig

    def plot_orientation_vs_time(self):
        """Plot orientation (Euler angles) vs time"""
        fig, axes = plt.subplots(3, 1, figsize=(12, 8))

        orientations = np.array(self.orientation_history)
        times = np.array(self.time_history)

        labels = ['Roll', 'Pitch', 'Yaw']
        colors = ['r', 'g', 'b']

        for i, (ax, label, color) in enumerate(zip(axes, labels, colors)):
            ax.plot(times, np.degrees(orientations[:, i]), color=color, linewidth=2)
            ax.set_ylabel(f'{label} (deg)')
            ax.grid(True, alpha=0.3)
            ax.set_title(label)

        axes[-1].set_xlabel('Time (s)')
        plt.suptitle('Orientation (Euler Angles) vs Time', fontsize=14, fontweight='bold')
        plt.tight_layout()
        return fig

    def plot_all(self):
        """Generate all plots"""
        self.plot_trajectory_3d()
        self.plot_position_vs_time()
        self.plot_orientation_vs_time()
        plt.show()


# Main execution
if __name__ == "__main__":
    # Create tracker instance
    tracker = SpatialTracker()

    # Simulation parameters
    dt = 0.05
    duration = 10.0
    steps = int(duration / dt)

    # Simulate helical motion
    for i in range(steps):
        t = i * dt

        # Circular motion in XY plane, linear in Z
        linear_vel = np.array([
            -2.0 * np.sin(t),  # X velocity
            2.0 * np.cos(t),   # Y velocity
            0.5                # Z velocity (upward)
        ])
```

```python
            # Rotating around all axes
            angular_vel = np.array([
                0.1 * np.sin(t),    # Roll
                0.1 * np.cos(t),    # Pitch
                1.0                 # Yaw (spinning)
            ])

            tracker.update_motion(linear_vel, angular_vel, dt)

        # Print final state
        print(f"Final position: {tracker.position}")
        print(f"Final orientation (deg): {np.degrees(tracker.orientation)}")

        # Show just the 3D plot (recommended)
        tracker.plot_trajectory_3d()
        plt.show()

        # Or show all plots
        # tracker.plot_all()
```