Computer Graphics Lab - CS352

Weekly Assignment Report

Question:

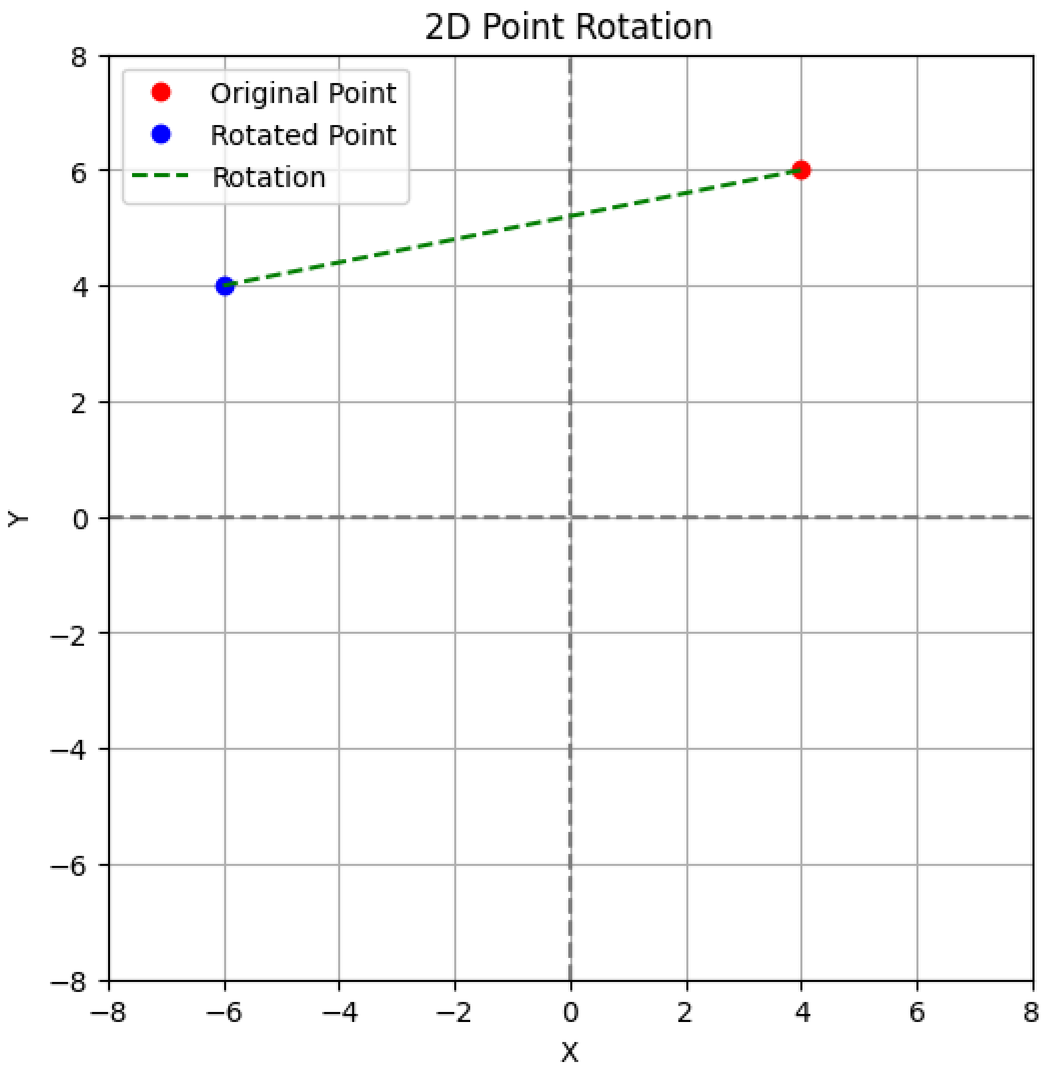
Python

```python
# Q1: Write a Python program to rotate a given point (4,6) by 90 degrees using the 2D rotation

# matrix. Visualize the original and rotated points using Matplotlib. Explain how the

# rotation matrix is applied to achieve the transformation.



import numpy as np

import matplotlib.pyplot as plt



def rotate_point(point, angle_degrees):

  angle_radians = np.radians(angle_degrees)

  rotation_matrix = np.array([

      [np.cos(angle_radians), -np.sin(angle_radians)],

      [np.sin(angle_radians), np.cos(angle_radians)]

  ])

  rotated_point = np.dot(rotation_matrix, point)

  return tuple(rotated_point)



# Original point
```

```
original_point = (4, 6)


# Rotate the point by 90 degrees

rotated_point = rotate_point(original_point, 90)




# Visualization

plt.figure(figsize=(6, 6))

plt.plot(original_point[0], original_point[1], 'ro', label='Original Point')

plt.plot(rotated_point[0], rotated_point[1], 'bo', label='Rotated Point')


# Draw lines to represent the x and y axis

plt.axhline(0, color='gray', linestyle='--')

plt.axvline(0, color='gray', linestyle='--')


# Draw a line connecting the original point to the rotated point

plt.plot([original_point[0],rotated_point[0]], [original_point[1],
rotated_point[1]], 'g--', label='Rotation')




plt.xlabel('X')

plt.ylabel('Y')

plt.title('2D Point Rotation')
```

```
plt.grid(True)

plt.legend()

plt.xlim(-8,8) #Adjust as needed

plt.ylim(-8,8) #Adjust as needed

plt.gca().set_aspect('equal', adjustable='box')

plt.show()


print(f"Original Point: {original_point}")

print(f"Rotated Point: {rotated_point}")




#Explanation:


#A 2D rotation matrix helps to rotate points around the origin in a 2D plane.

#The general form for rotation counterclockwise by angle θ is:

# | cos(θ)  -sin(θ) |

# | sin(θ)   cos(θ) |
```

Output:


2D Point Rotation
Original Point
Rotated Point
Rotation
8
6
4
2
0
−2
−4
−6
−8
Y
−8
−6
−4
−2
0
2
4
6
8
X

Question 2:

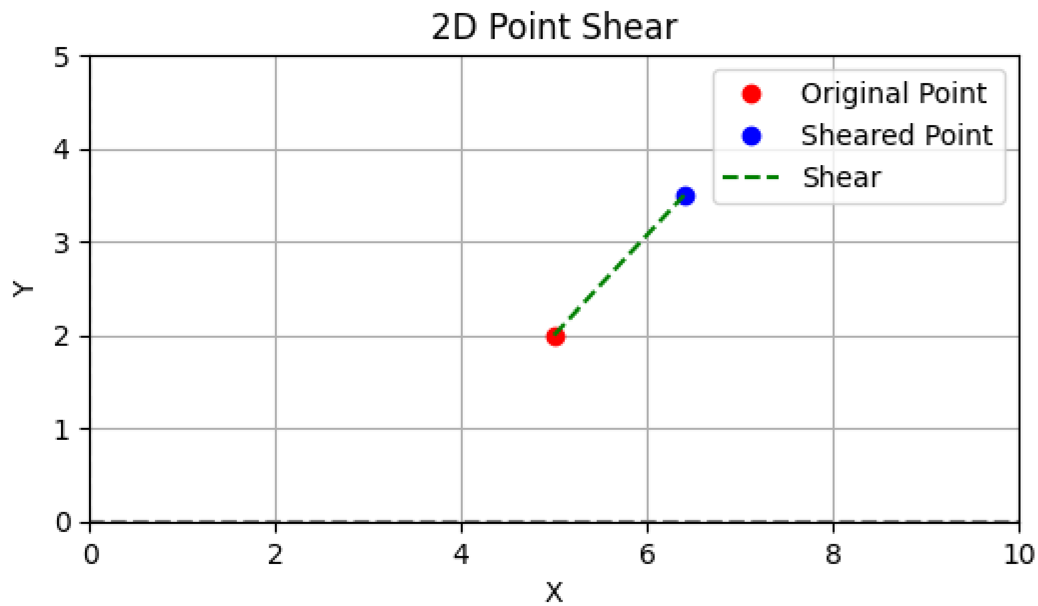
Python

```python
# Q2: Write a Python program to apply a 2D shear transformation on a given point (5,2)
# using shear factors of 0.7 along the X-axis and 0.3 along the Y-axis. Visualize the
# original and sheared points using Matplotlib. Explain how the shear matrix is applied
# to achieve the transformation.

import numpy as np
import matplotlib.pyplot as plt

def shear_point(point, shear_x, shear_y):
  shear_matrix = np.array([[1, shear_x],
                           [shear_y, 1]])
  point_matrix = np.array([[point[0]], [point[1]]])
  sheared_point_matrix = np.dot(shear_matrix, point_matrix)
  return (sheared_point_matrix[0, 0], sheared_point_matrix[1, 0])

# Original point
original_point = (5, 2)

# Shear factors
shear_x = 0.7
```

```
shear_y = 0.3


# Apply shear transformation

sheared_point = shear_point(original_point, shear_x, shear_y)


# Visualization

plt.figure(figsize=(6, 6))

plt.plot(original_point[0], original_point[1], 'ro', label='Original Point')

plt.plot(sheared_point[0], sheared_point[1], 'bo', label='Sheared Point')


# Draw lines to represent the x and y axis

plt.axhline(0, color='gray', linestyle='--')

plt.axvline(0, color='gray', linestyle='--')




plt.plot([original_point[0], sheared_point[0]], [original_point[1],
sheared_point[1]], 'g--', label='Shear')


plt.xlabel('X')

plt.ylabel('Y')

plt.title('2D Point Shear')

plt.grid(True)

plt.legend()
```

```
plt.xlim(0, 10)

plt.ylim(0, 5)

plt.gca().set_aspect('equal', adjustable='box')

plt.show()


print(f"Original Point: {original_point}")

print(f"Sheared Point: {sheared_point}")


# Explanation:

# A 2D shear matrix transforms a point by shifting its coordinates along one axis in proportion

# to its position along the other axis.


# The general form of a 2D shear matrix with shear factor sh_x along the x-axis and sh_y along the y-axis is:

# | 1    sh_x |

# | sh_y  1    |
```

Output:


2D Point Shear
Original Point
Sheared Point
Shear
Y
X
0
1
2
3
4
5
0
2
4
6
8
10

Question 3:

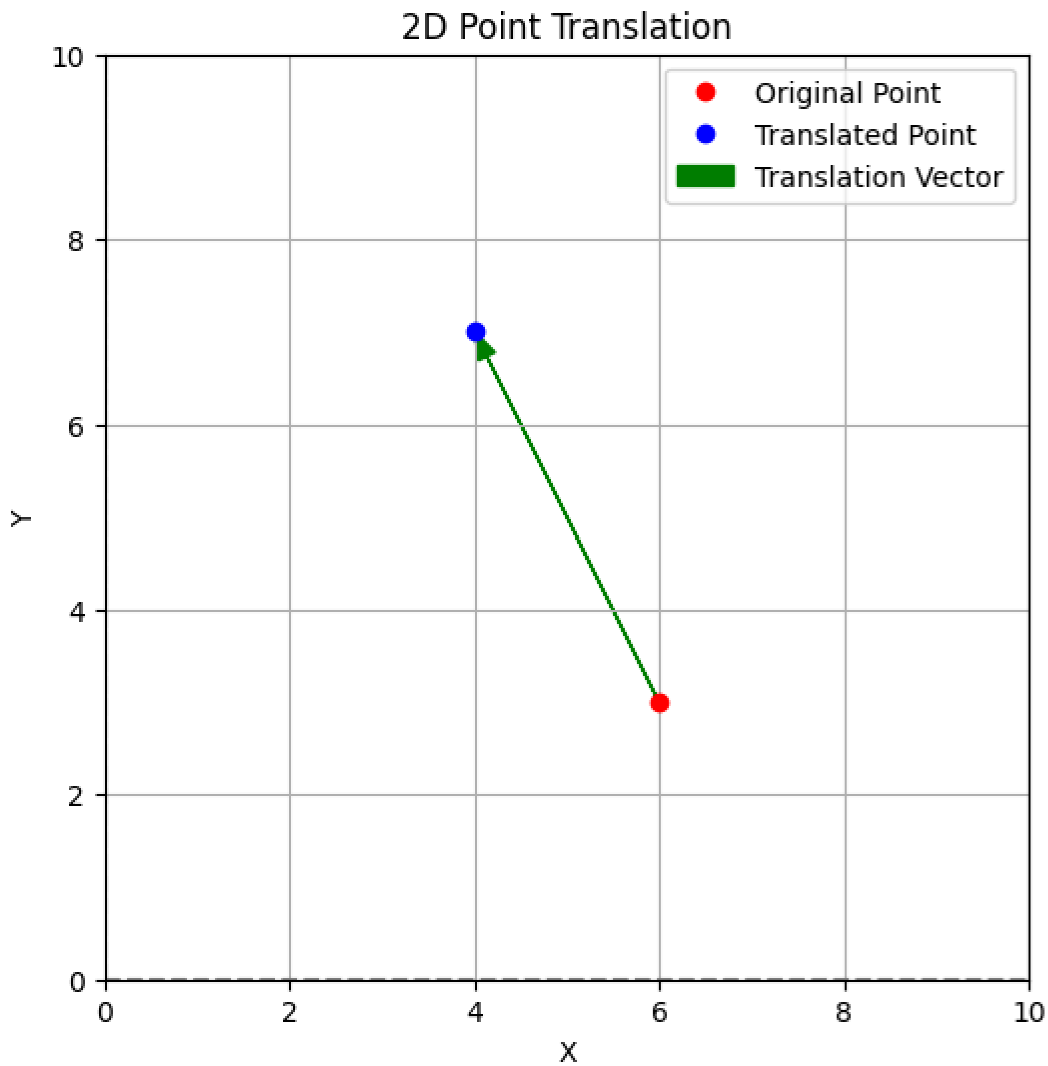
Python

```python
# Q3: Write a Python program to apply a 2D translation transformation on a given point
# (6,3) using a translation vector of (-2,4). Visualize the original and translated points
# using Matplotlib. Explain how the translation vector is applied to shift the point.


import numpy as np

import matplotlib.pyplot as plt


def translate_point(point, translation_vector):

  translated_point = (point[0] + translation_vector[0], point[1] + translation_vector[1])

  return translated_point


# Original point

original_point = (6, 3)


# Translation vector

translation_vector = (-2, 4)


# Apply translation

translated_point = translate_point(original_point, translation_vector)
```

```
# Visualization
plt.figure(figsize=(6, 6))
plt.plot(original_point[0], original_point[1], 'ro', label='Original Point')
plt.plot(translated_point[0], translated_point[1], 'bo', label='Translated
Point')


# Draw lines to represent the x and y axis
plt.axhline(0, color='gray', linestyle='--')
plt.axvline(0, color='gray', linestyle='--')


# Draw an arrow to represent the translation vector
plt.arrow(original_point[0], original_point[1], translation_vector[0],
translation_vector[1],
          head_width=0.2, head_length=0.3, fc='g', ec='g',
length_includes_head=True, label='Translation Vector')


plt.xlabel('X')
plt.ylabel('Y')
plt.title('2D Point Translation')
plt.grid(True)
plt.legend()
plt.xlim(0, 10)  # Adjust as needed
plt.ylim(0, 10)  # Adjust as needed
```

```python
plt.gca().set_aspect('equal', adjustable='box')

plt.show()



print(f"Original Point: {original_point}")

print(f"Translated Point: {translated_point}")



# Explanation:

# A 2D translation shifts a point by adding the translation vector's components to the point's coordinates.

# In this case, the original point (6,3) is shifted by (-2, 4).

# New x-coordinate: 6 + (-2) = 4

# New y-coordinate: 3 + 4 = 7

# So the translated point is (4, 7)
```

Output:


2D Point Translation
Original Point
Translated Point
Translation Vector
10
8
6
4
2
0
0
2
4
6
8
10
X
Y

Question 4:

Python

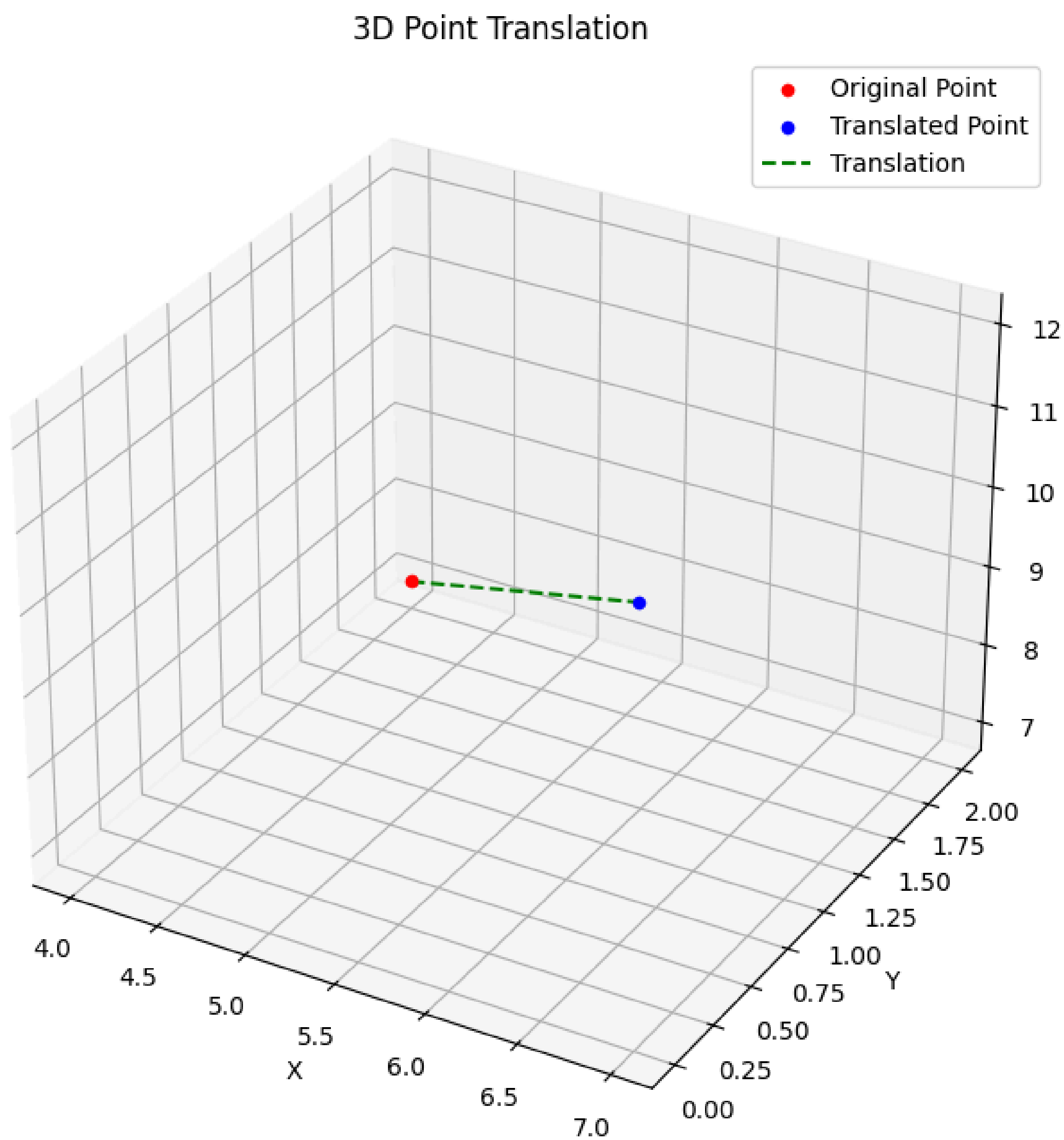
```
# Q4: Write a Python program to apply a 3D translation transformation using a

# homogeneous matrix on a point (4,2,7,1) using translation values of (3,-2,5)
along the

# X, Y, and Z axes, respectively. Visualize the original and translated points
using

# Matplotlib's 3D plotting capabilities. Explain how the homogeneous coordinates
and

# the 4x4 translation matrix are used to perform the transformation.


import numpy as np

import matplotlib.pyplot as plt

from mpl_toolkits.mplot3d import Axes3D


def translate_3d_point(point, translation_vector):

    # Create the 4x4 translation matrix

    translation_matrix = np.array([

        [1, 0, 0, translation_vector[0]],

        [0, 1, 0, translation_vector[1]],

        [0, 0, 1, translation_vector[2]],

        [0, 0, 0, 1]

    ])


    # Convert the point to a homogeneous coordinate
```

```
    point_homogeneous = np.array(point).reshape(4, 1)

    # Apply the translation
    translated_point_homogeneous = np.dot(translation_matrix, point_homogeneous)

    # Convert back to Cartesian coordinates
    translated_point = tuple(translated_point_homogeneous[:3].flatten())

    return translated_point


# Original point
original_point = (4, 2, 7, 1)

# Translation vector
translation_vector = (3, -2, 5)

# Apply translation
translated_point = translate_3d_point(original_point, translation_vector)

# Visualization
fig = plt.figure(figsize=(8, 8))
ax = fig.add_subplot(111, projection='3d')
```

```python
ax.scatter(original_point[0], original_point[1], original_point[2], color='r', label='Original Point')

ax.scatter(translated_point[0], translated_point[1], translated_point[2], color='b', label='Translated Point')


ax.plot([original_point[0], translated_point[0]],

        [original_point[1], translated_point[1]],

        [original_point[2], translated_point[2]], 'g--', label='Translation')


ax.set_xlabel('X')

ax.set_ylabel('Y')

ax.set_zlabel('Z')

ax.set_title('3D Point Translation')

ax.legend()

plt.show()


print(f"Original Point: {original_point[:3]}")  # Print without the homogeneous coordinate

print(f"Translated Point: {translated_point}")


#Explanation:
```

```
#Homogeneous coordinates represent a point in 3D space using four coordinates
(x, y, z, w).

# When w = 1, the Cartesian coordinates are (x/w, y/w, z/w). A 4x4
transformation matrix

# enables combining translations with rotations and scaling in 3D space.

# The translation matrix shifts a point along each axis.  Applying a 4x4
translation matrix to a point

# in homogeneous coordinates effectively translates the point in 3D space.
```

Output:



3D Point Translation
Original Point
Translated Point
Translation
12
11
10
9
8
7
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
Y
4.0
4.5
5.0
5.5
6.0
6.5
7.0
X

Question 5:

Python

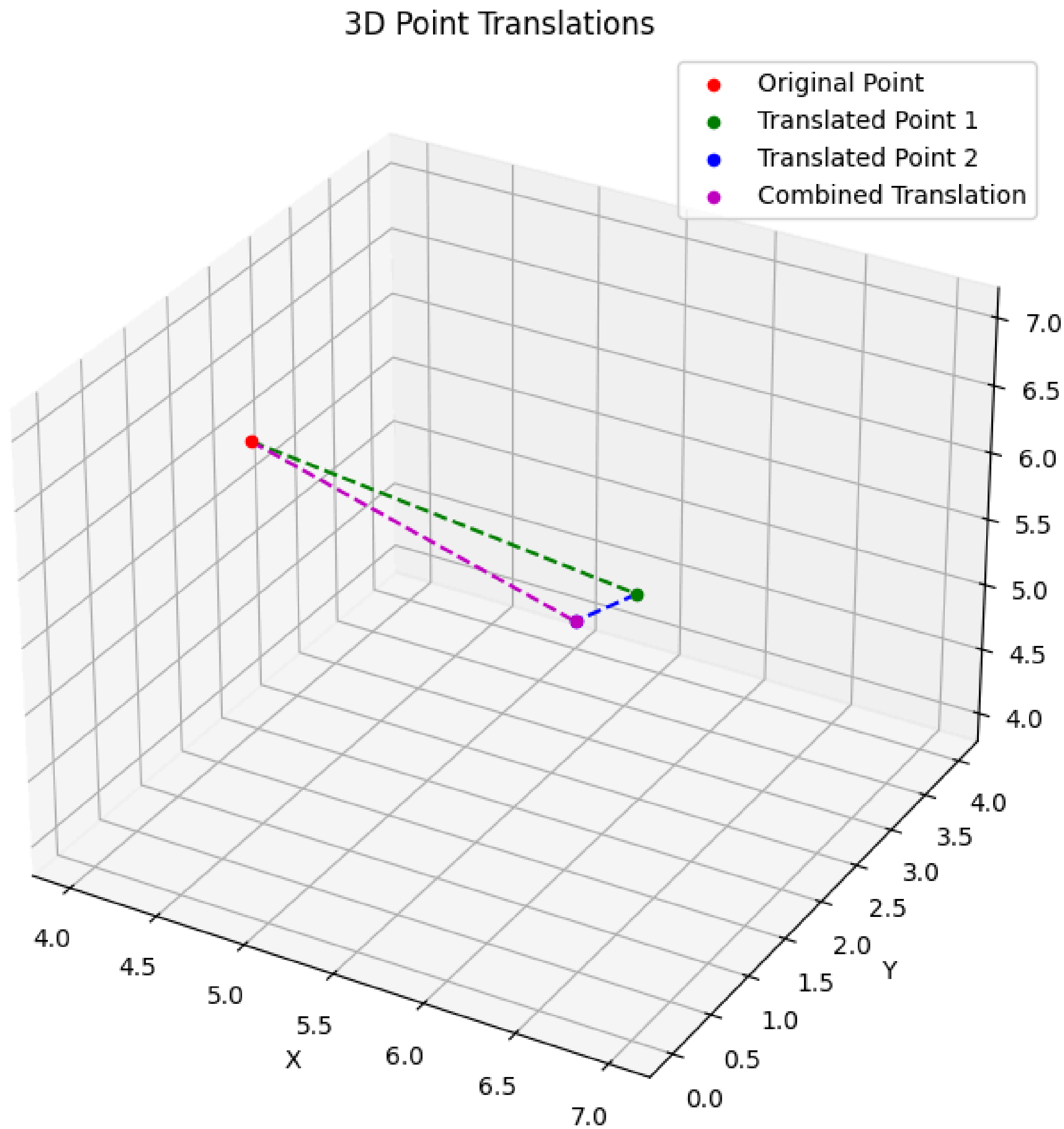
```python
# Q5: Write a Python program to apply two sequential 3D translations using homogeneous
# matrices on a point (4,2,6,1) using translation vectors (3,-2,1) and (-2,4,-3)Then,
# apply a combined translation using the sum of these vectors and verify if the result
# matches the sequential translation. Visualize all points using Matplotlib's 3D plotting
# capabilities and explain how homogeneous matrices perform these transformations.


import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D


def translate_3d_point(point, translation_vector):
    translation_matrix = np.array([
        [1, 0, 0, translation_vector[0]],
        [0, 1, 0, translation_vector[1]],
        [0, 0, 1, translation_vector[2]],
        [0, 0, 0, 1]
    ])
    point_homogeneous = np.array(point).reshape(4, 1)
```

```
    translated_point_homogeneous = np.dot(translation_matrix, point_homogeneous)

    translated_point = tuple(translated_point_homogeneous[:3].flatten())

    return translated_point


# Original point

original_point = (4, 2, 6, 1)


# Translation vectors

translation_vector1 = (3, -2, 1)

translation_vector2 = (-2, 4, -3)


# Sequential translations

translated_point1 = translate_3d_point(original_point, translation_vector1)

translated_point2 = translate_3d_point(translated_point1 + (1,),
translation_vector2)




# Combined translation

combined_translation_vector = (translation_vector1[0] + translation_vector2[0],

                               translation_vector1[1] + translation_vector2[1],

                               translation_vector1[2] + translation_vector2[2])
```

```
translated_point_combined = translate_3d_point(original_point,
combined_translation_vector)


# Visualization

fig = plt.figure(figsize=(10, 8))

ax = fig.add_subplot(111, projection='3d')


ax.scatter(original_point[0], original_point[1], original_point[2], color='r',
label='Original Point')

ax.scatter(translated_point1[0], translated_point1[1], translated_point1[2],
color='g', label='Translated Point 1')

ax.scatter(translated_point2[0], translated_point2[1], translated_point2[2],
color='b', label='Translated Point 2')

ax.scatter(translated_point_combined[0], translated_point_combined[1],
translated_point_combined[2], color='m', label='Combined Translation')



ax.plot([original_point[0], translated_point1[0]], [original_point[1],
translated_point1[1]], [original_point[2], translated_point1[2]], 'g--')

ax.plot([translated_point1[0], translated_point2[0]], [translated_point1[1],
translated_point2[1]], [translated_point1[2], translated_point2[2]], 'b--')

ax.plot([original_point[0], translated_point_combined[0]], [original_point[1],
translated_point_combined[1]], [original_point[2],
translated_point_combined[2]], 'm--')
```

```
ax.set_xlabel('X')

ax.set_ylabel('Y')

ax.set_zlabel('Z')

ax.set_title('3D Point Translations')

ax.legend()

plt.show()


print(f"Original Point: {original_point[:3]}")

print(f"Translated Point 1: {translated_point1}")

print(f"Translated Point 2: {translated_point2}")

print(f"Combined Translated Point: {translated_point_combined}")


# Explanation:

# Homogeneous coordinates and transformation matrices are used for applying 3D
transformations.

# Homogeneous coordinates represent a 3D point as a 4D vector.  A 4x4
translation matrix

# translates the point in 3D space. Sequential translations can be combined by
summing the translation vectors.
```

Output:


3D Point Translations
Original Point
Translated Point 1
Translated Point 2
Combined Translation
X
Y
4.0
4.5
5.0
5.5
6.0
6.5
7.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.0
4.5
5.0
5.5
6.0
6.5
7.0

Question 6:

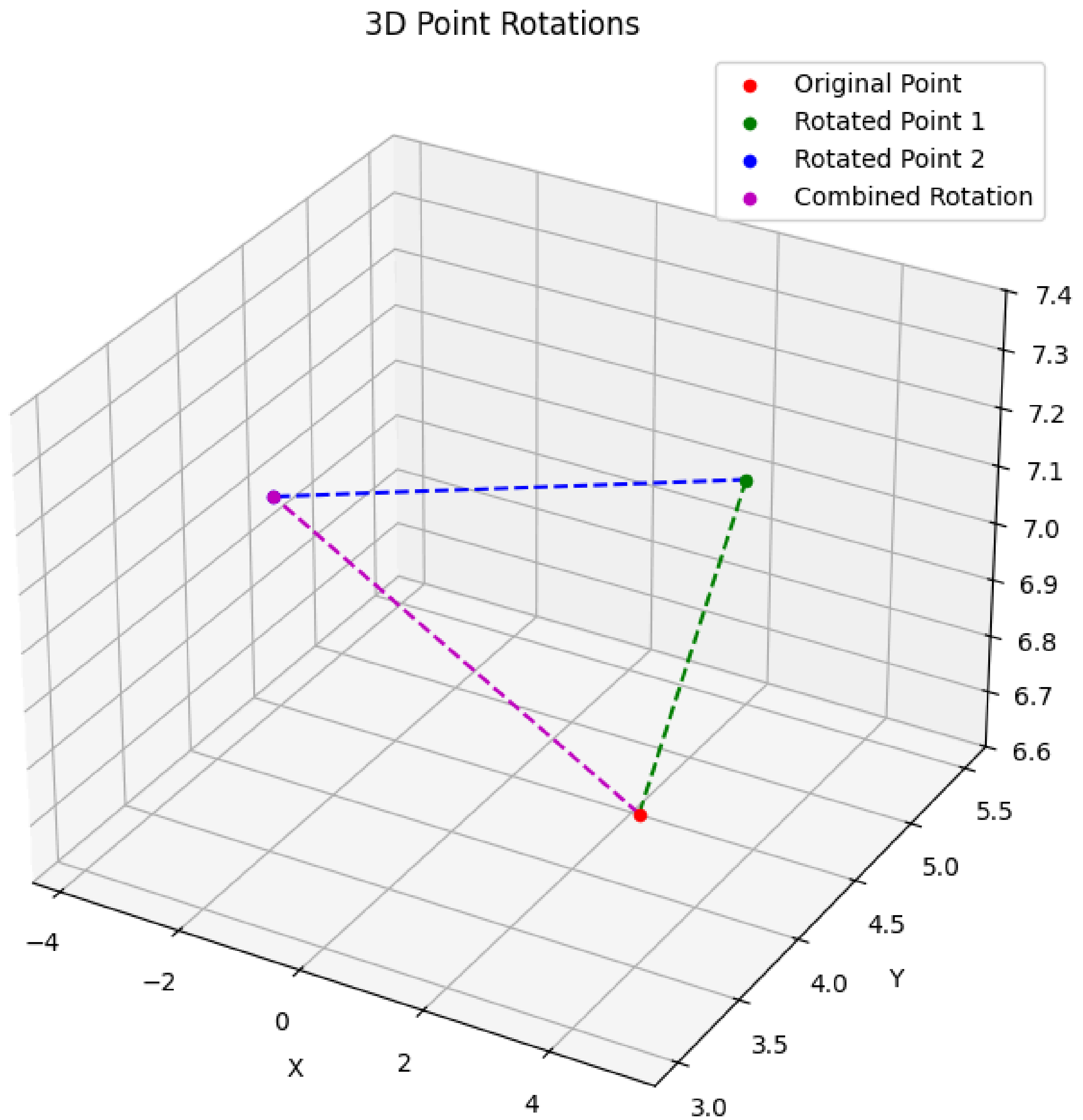
```python
# Q6: Write a Python program to apply two sequential 3D rotations using homogeneous
# matrices on a point (5,3,7,1) using rotation angles 40◦ and 60◦ about the z-axis. Then,
# apply a combined rotation using the sum of these angles and verify if the result matches
# the sequential rotation. Visualize all points using Matplotlib's 3D plotting capabilities
# and explain how homogeneous matrices perform these transformations.


import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D


def rotate_z(point, angle_degrees):
    """Rotates a 3D point around the z-axis."""
    angle_radians = np.radians(angle_degrees)
    rotation_matrix = np.array([
        [np.cos(angle_radians), -np.sin(angle_radians), 0, 0],
        [np.sin(angle_radians), np.cos(angle_radians), 0, 0],
        [0, 0, 1, 0],
        [0, 0, 0, 1]
    ])
```

```
    point_homogeneous = np.array(point).reshape(4, 1)

    rotated_point_homogeneous = np.dot(rotation_matrix, point_homogeneous)

    rotated_point = tuple(rotated_point_homogeneous[:3].flatten())

    return rotated_point


# Original point

original_point = (5, 3, 7, 1)


# Rotation angles

angle1 = 40

angle2 = 60


# Sequential rotations

rotated_point1 = rotate_z(original_point, angle1)

rotated_point2 = rotate_z(rotated_point1 + (1,), angle2)


# Combined rotation

combined_angle = angle1 + angle2

rotated_point_combined = rotate_z(original_point, combined_angle)


# Visualization

fig = plt.figure(figsize=(10, 8))

ax = fig.add_subplot(111, projection='3d')
```

```
ax.scatter(original_point[0], original_point[1], original_point[2], color='r',
label='Original Point')

ax.scatter(rotated_point1[0], rotated_point1[1], rotated_point1[2], color='g',
label='Rotated Point 1')

ax.scatter(rotated_point2[0], rotated_point2[1], rotated_point2[2], color='b',
label='Rotated Point 2')

ax.scatter(rotated_point_combined[0], rotated_point_combined[1],
rotated_point_combined[2], color='m', label='Combined Rotation')


ax.plot([original_point[0], rotated_point1[0]], [original_point[1],
rotated_point1[1]], [original_point[2], rotated_point1[2]], 'g--')

ax.plot([rotated_point1[0], rotated_point2[0]], [rotated_point1[1],
rotated_point2[1]], [rotated_point1[2], rotated_point2[2]], 'b--')

ax.plot([original_point[0], rotated_point_combined[0]], [original_point[1],
rotated_point_combined[1]], [original_point[2], rotated_point_combined[2]],
'm--')


ax.set_xlabel('X')

ax.set_ylabel('Y')

ax.set_zlabel('Z')

ax.set_title('3D Point Rotations')

ax.legend()

plt.show()


print(f"Original Point: {original_point[:3]}")
```

```python
print(f"Rotated Point 1: {rotated_point1}")

print(f"Rotated Point 2: {rotated_point2}")

print(f"Combined Rotated Point: {rotated_point_combined}")



# Explanation:

# Homogeneous coordinates represent points in 3D space as 4D vectors (x, y, z,
w).

# The rotation matrices are 4x4 to allow for combined transformations.

# Sequential rotations are not commutative - the order matters.

# A combined rotation matrix (sum of angles) is not equal to sequential
rotations.
```

Output:


3D Point Rotations
Original Point
Rotated Point 1
Rotated Point 2
Combined Rotation
X
Y
−4
−2
0
2
4
3.0
3.5
4.0
4.5
5.0
5.5
6.6
6.7
6.8
6.9
7.0
7.1
7.2
7.3
7.4

Question 7:

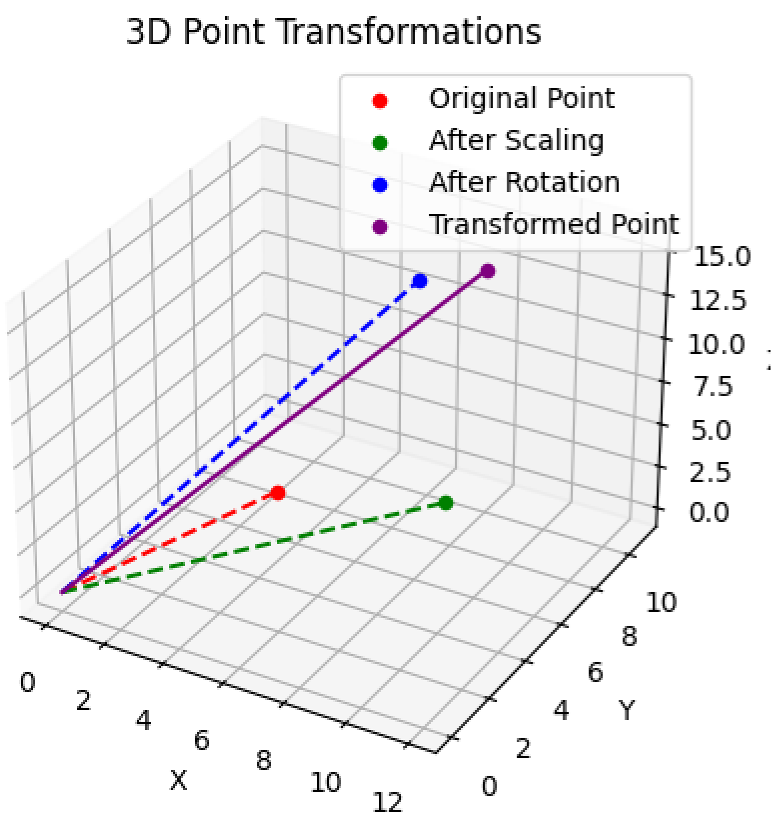
```python
# Q7: Given a point (6, 2, 7) in homogeneous coordinates [6, 2, 7, 1], perform the following

# transformations:

# 1. Scaling: Factors (2.0, 0.5, 1.5) along x, y, and z axes.

# 2. Rotation: 60° about the z-axis.

# 3. Translation: By (4, -3, 5) along x, y, and z.

# Tasks:

# 1. Create 4x4 matrices for scaling, rotation, and translation using NumPy.

# 2. Compute the composite transformation matrix.

# 3. Apply it to the point and print the result.

# 4. Visualize the original and transformed points using Matplotlib with dashed

# lines from the origin.

# 5. Plot intermediate points after each transformation.


import numpy as np

import matplotlib.pyplot as plt

from mpl_toolkits.mplot3d import Axes3D


# Define the point in homogeneous coordinates

point = np.array([6, 2, 7, 1])


# 1. Scaling matrix
```

```python
scaling_factors = (2.0, 0.5, 1.5)

scaling_matrix = np.diag([scaling_factors[0], scaling_factors[1],
scaling_factors[2], 1])


# 2. Rotation matrix (60° about the z-axis)

angle_rad = np.radians(60)

rotation_matrix = np.array([

    [np.cos(angle_rad), -np.sin(angle_rad), 0, 0],

    [np.sin(angle_rad), np.cos(angle_rad), 0, 0],

    [0, 0, 1, 0],

    [0, 0, 0, 1]

])


# 3. Translation matrix

translation_vector = (4, -3, 5)

translation_matrix = np.eye(4)

translation_matrix[:3, 3] = translation_vector


# 4. Composite transformation matrix

composite_matrix = translation_matrix @ rotation_matrix @ scaling_matrix


# 5. Apply the transformation to the point

transformed_point = composite_matrix @ point
```

```python
# Print the result
print("Transformed Point:", transformed_point)

# Visualization
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Plot the original point
ax.scatter(point[0], point[1], point[2], color='r', label='Original Point')
ax.plot([0, point[0]], [0, point[1]], [0, point[2]], 'r--')

# Intermediate points and transformations visualization
intermediate_point1 = scaling_matrix @ point
ax.scatter(intermediate_point1[0], intermediate_point1[1],
intermediate_point1[2], color='g', label="After Scaling")
ax.plot([0, intermediate_point1[0]], [0, intermediate_point1[1]], [0,
intermediate_point1[2]], 'g--')

intermediate_point2 = rotation_matrix @ intermediate_point1
ax.scatter(intermediate_point2[0], intermediate_point2[1],
intermediate_point2[2], color='b', label="After Rotation")
ax.plot([0, intermediate_point2[0]], [0, intermediate_point2[1]], [0,
intermediate_point2[2]], 'b--')
```

```python
# Plot the transformed point

ax.scatter(transformed_point[0], transformed_point[1], transformed_point[2],
color='purple', label='Transformed Point')

ax.plot([0, transformed_point[0]], [0, transformed_point[1]], [0,
transformed_point[2]], 'purple')



ax.set_xlabel('X')

ax.set_ylabel('Y')

ax.set_zlabel('Z')

ax.set_title('3D Point Transformations')

ax.legend()

plt.show()
```

Output:



Colab link for code is here.