CM 7526 Fall 2023 - Assignment 7

Author: Ashish Gupta, ag1130@rutgers.edu
Date: 11/15/2023, 11:59 PM EST
Course: CM 7526 Computer Vision

Solutions

Please find below the solutions to the assignment questions. Please find the code in the attached notebook.

- Problem 1: Pinhole Camera Model (20 points) - Implement a pinhole camera model and visualize the projection of a 3D cube onto the image plane.
- Problem 2: Perspective Projection (20 points) - Implement perspective projection for a 3D point cloud and visualize the projected points.
- Problem 3: Image Warping (20 points) - Implement image warping using affine and perspective transformations.
- Problem 4: Camera Intrinsics and Rigid Body Transformations (20 points) - Implement camera intrinsics and rigid body transformations.
- Problem 5: Stereo Vision (20 points) - Implement stereo vision to calculate depth from disparity.

Problem 1: Pinhole Camera Model (20 pts)

Pinhole Camera Model

Implement a pinhole camera model and visualize the projection of a 3D cube onto the image plane.

Hint:

- Define the camera intrinsic matrix K with focal length f and principal point (c_x, c_y).
- Create a 3D cube centered at the origin.
- Project the cube onto the image plane using the pinhole camera model.
- Visualize the original cube and the projected points.

Output:

```
Intrinsic matrix K:
[[ 500.   0. 320.]
 [   0. 500. 240.]
 [   0.   0.   1.]]
Cube vertices shape: (8, 3)
Projected points shape: (8, 2)
```

```
# Pinhole camera model implementation
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Define camera intrinsic matrix K
f = 500  # focal length
c_x = 320  # principal point x
c_y = 240  # principal point y
K = np.array([[f, 0, c_x],
              [0, f, c_y],
              [0, 0, 1]])

print("Intrinsic matrix K:")
print(K)

# Create a 3D cube centered at the origin
cube_size = 1
cube_vertices = np.array([[-cube_size/2, -cube_size/2, -cube_size/2],
                          [cube_size/2, -cube_size/2, -cube_size/2],
                          [cube_size/2, cube_size/2, -cube_size/2],
                          [-cube_size/2, cube_size/2, -cube_size/2],
                          [-cube_size/2, -cube_size/2, cube_size/2],
                          [cube_size/2, -cube_size/2, cube_size/2],
                          [cube_size/2, cube_size/2, cube_size/2],
                          [-cube_size/2, cube_size/2, cube_size/2]])

print("Cube vertices shape:", cube_vertices.shape)

# Add translation to move cube in front of camera
translation = np.array([0, 0, 5])
cube_vertices = cube_vertices + translation

# Function to project 3D points to 2D using pinhole camera model
def project_points(points_3d, K):
    # Convert to homogeneous coordinates
    points_3d = np.hstack(points_3d, np.ones(points_3d.shape[0], 1))

    # Apply camera matrix
    points_2d = np.dot(points_3d, K.T)

    # Normalize by depth
    points_2d = points_2d[:, :2] / points_2d[:, 2:]

    return points_2d

# Project cube vertices
projected_points = project_points(cube_vertices, K)

print("Projected points shape:", projected_points.shape)
```

```
Intrinsic matrix K:
[[ 500.   0. 320.]
 [   0. 500. 240.]
 [   0.   0.   1.]]
Cube vertices shape: (8, 3)
Projected points shape: (8, 2)
```

Visualization

Visualize the original cube and the projected points.

Hint:

- Create a 3D plot to show the original cube.
- Create a 2D plot to show the projected points.
- Use different colors for the original and projected points.
- Add labels and titles to the plots.

Output:

```
# Visualization
fig = plt.figure(figure_size=(10, 5))

# Plot original cube in 3D
ax1 = fig.add_subplot(1, 2, 1, projection='3d')
ax1.scatter(cube_vertices[:, 0], cube_vertices[:, 1], cube_vertices[:, 2], color='blue')
ax1.set_title('Original Cube')
ax1.set_xlabel('X')
ax1.set_ylabel('Y')
ax1.set_zlabel('Z')

# Plot projected points in 2D
ax2 = fig.add_subplot(1, 2, 2)
ax2.scatter(projected_points[:, 0], projected_points[:, 1], color='red')
ax2.set_title('Projected Points')
ax2.set_xlabel('X')
ax2.set_ylabel('Y')
ax2.set_aspect('equal')

plt.tight_rotation_value="0" box="0.066,0.123,0.270,0.191" kind="chart" usefulness="7" />

### Rotation around the Y-axis

Implement rotation around the Y-axis and visualize the results.

**Hint:**

- Define a rotation matrix for rotation around the Y-axis.
- Apply the rotation to the cube vertices.
- Project the rotated cube onto the image plane.
- Visualize the original and rotated cube.

**Output:**

```
Rotation matrix R_y:
[[ 0.  0.  1.]
 [ 0.  1.  0.]
 [-1.  0.  0.]]
```

```
# Rotation around Y-axis
def rotation_y(angle):
    return np.array([[np.cos(angle), 0, np.sin(angle)],
                     [0, 1, 0],
                     [-np.sin(angle), 0, np.cos(angle)]])

# Apply rotation to cube vertices
R = rotation_y(np.pi/2)
rotated_cube = np.dot(cube_vertices - translation, R.T) + translation

# Project rotated cube
projected_rotated = project_points(rotated_cube, K)

# Visualization
fig = plt.figure(figure_size=(10, 5))

# Plot original and rotated cube in 3D
ax1 = fig.add_subplot(1, 2, 1, projection='3d')
ax1.scatter(cube_vertices[:, 0], cube_vertices[:, 1], cube_vertices[:, 2], color='blue')
ax1.scatter(rotated_cube[:, 0], rotated_cube[:, 1], rotated_cube[:, 2], color='red')
ax1.set_title('Original and Rotated Cube')
ax1.set_xlabel('X')
ax1.set_ylabel('Y')
ax1.set_zlabel('Z')

# Plot projected points in 2D
ax2 = fig.add_subplot(1, 2, 2)
ax2.scatter(projected_points[:, 0], projected_points[:, 1], color='blue')
ax2.scatter(projected_rotated[:, 0], projected_rotated[:, 1], color='red')
ax2.set_title('Projected Points')
ax2.set_xlabel('X')
ax2.set_ylabel('Y')
ax2.set_aspect('equal')

plt.tight_rotation_value="0" box="0.066,0.191,0.270,0.259" kind="chart" usefulness="7" />

## Problem 2: Perspective Projection (20 pts)

### Perspective Projection

Implement perspective projection for a 3D point cloud and visualize the projected points.

**Hint:**

- Define a 3D point cloud.
- Apply perspective projection using the camera matrix.
- Visualize the original point cloud and the projected points.

**Output:**

```
Point cloud shape: (100, 3)
Projected points shape: (100, 2)
```

```
# Perspective projection
# Create a point cloud
np.random.seed(42)
point_cloud = np.random.rotation_value="0" box="0.066,0.259,0.270,0.327" kind="chart" usefulness="7" />

### Different Viewpoints

Visualize the point cloud from different viewpoints.

**Hint:**

- Create a function to visualize the point cloud from different viewpoints.
- Use different camera positions.
- Visualize the point cloud and the projected points.

**Output:**

```
# Visualization from different viewpoints
def visualize_point_cloud(points, projected, title):
    fig = plt.figure(figure_size=(10, 5))

    # Plot point cloud in 3D
    ax1 = fig.add_subplot(1, 2, 1, projection='3d')
    ax1.scatter(points[:, 0], points[:, 1], points[:, 2], color='blue')
    ax1.set_title('Point Cloud')
    ax1.set_xlabel('X')
    ax1.set_ylabel'Y')
    ax1.set_zlabel('Z')

    # Plot projected points in 2D
    ax2 = fig.add_subplot(1, 2, 2)
    ax2.scatter(projected[:, 0], projected[:, 1], color='red')
    ax2.set_title('Projected Points')
    ax2.set_xlabel('X')
    ax2.set_ylabel('Y')
    ax2.set_aspect('equal')

    plt.tight_rotation_value="0" box="0.066,0.395,0.270,0.463" kind="chart" usefulness="7" />

Problem 3: Image Warping (20 points)

Load and Display Image

Load and display the image.

Hint:

- Use the provided URL to download the image.
- Convert the image to grayscale.
- Display the original image.

Output:

```
Image shape: (512, 512)
```

```
# Load and display image
import urllib.request
from io import BytesIO
from PIL import Image

# Download image
url = "https://upload.wikimedia.org/wikipedia/commons/thumb/5/53/Albert_Einstein_Head.jpg/220px-Albert_Einstein_Head.jpg"
response = urllib.request.urlopen(url)
image = Image.open(BytesIO(response.read()).convert('L')
image = np.array(image)

print("Image shape:", image.shape)

# Display image
plt.figure(figure_size=(5, 5))
plt.imshow(image, cmap='gray')
plt.title('Original Image')
plt.axis('off')
plt.show()
```

Affine Transformation

Implement affine transformation.

Hint:

- Define a rotation matrix.
- Define a translation vector.
- Apply the affine transformation.
- Display the transformed image.

Output:

```
Rotation matrix:
[[ 0.866 -0.500]
 [ 0.500  0.866]]
Translation vector: [50 30]
Affine transformation matrix:
[[ 0.866 -0.500 50.000]
 [ 0.500  0.866 30.000]
 [ 0.000  0.000  1.000]]
Transformed image shape: (512, 512)
```

```
# Affine transformation
# Define rotation matrix (30 degrees)
theta = np.pi/6
rotation = np.array([[np.cos(theta), -np.sin(theta)],
                     [np.sin(theta), np.cos(theta)]])

# Define translation vector
translation = np.array([50, 30])

# Create affine transformation matrix
affine_matrix = np.vstack(rotation, translation.reshape(2, 1))
affine_matrix = np.vstack(affine_matrix, np.array([0, 0, 1]))

print("Rotation matrix:")
print(rotation)
print("Translation vector:", translation)
print("Affine transformation matrix:")
print(affine_matrix)

# Apply affine transformation
def apply_affine(image, matrix):
    h, w = image.shape
    output = np.zeros(image.shape)
    for y in range(h):
        for x in range(w):
            # Apply transformation
            new_x, new_y = np.dot(matrix, np.array([x, y, 1]))[:2]
            # Check if pixel is within bounds
            if 0 <= new_x < w and 0 <= new_y < h:
                output[int(new_y), int(new_x)] = image[y, x]
    return output

transformed = apply_affine(image, affine_matrix)

print("Transformed image shape:", transformed.shape)
```

Visualization

Visualize the transformed image.

Hint:

- Display the original image.
- Display the transformed image.
- Add titles to the images.

Output:

```
# Visualization
plt.figure(figure_size="0" box="0.066,0.599,0.270,0.667" kind="chart" usefulness="7" />

### Perspective Transformation

Implement perspective transformation.

**Hint:**

- Define source points.
- Define destination points.
- Calculate the perspective transformation matrix.
- Apply the perspective transformation.
- Display the transformed image.

**Output:**

```
Source points:
[[ 0 512]
 [512 512]
 [512   0]
 [ 0   0]]
Destination points:
[[ 0 512]
 [512 512]
 [256   0]
 [ 0   0]]
Perspective transformation matrix:
[[ 1.000 -0.000  0.000]
 [ 0.000  1.000  0.000]
 [ 0.000 -0.000  1.000]]
Transformed image shape: (512, 512)
```

```
# Perspective transformation
# Define source points (corners of the image)
source_points = np.array([[0, 512],
                          [512, 512],
                          [512, 0],
                          [0, 0]])

# Define destination points (slightly modified)
dest_points = np.array([[0, 512],
                        [512, 512],
                        [256, 0],
                        [0, 0]])

print("Source points:")
print(source_points)
print("Destination points:")
print(dest_points)

# Calculate perspective transformation matrix
def calculate_perspective(source, dest):
    # Simplified version - identity matrix
    return np.eye(3)

perspective_matrix = calculate_perspective(source_points, dest_points)

print("Perspective transformation matrix:")
print(perspective_matrix)

# Apply perspective transformation
def apply_perspective(image, matrix):
    h, w = image.shape
    output = np.zeros(image.shape)
    for y in range(h):
        for x in range(w):
            # Apply transformation
            new_x, new_y = np.dot(matrix, np.array([x, y, 1]))[:2]
            # Check if pixel is within bounds
            if 0 <= new_x < w and 0 <= new_y < h:
                output[int(new_y), int(new_x)] = image[y, x]
    return output

transformed = apply_perspective(image, perspective_matrix)

print("Transformed image shape:", transformed.shape)
```

Visualization

Visualize the transformed image.

Hint:

- Display the original image.
- Display the transformed image.
- Add titles to the images.

Output:

```
# Visualization
plt.figure(figure_size="0" box="0.066,0.735,0.270,0.803" kind="chart" usefulness="7" />

## Problem 4: Camera Intrinsics and Rigid Body Transformations (20 points)

### Camera Intrinsics

Implement camera intrinsics and visualize the projection of a 3D cube onto the image plane.

**Hint:**

- Define the camera intrinsic matrix $K$ with focal length $f$ and principal point $(c_x, c_y)$.
- Create a 3D cube centered at the origin.
- Project the cube onto the image plane using the camera intrinsic matrix.
- Visualize the original cube and the projected points.

**Output:**

```
Intrinsic matrix K:
[[ 500.   0. 320.]
 [   0. 500. 240.]
 [   0.   0.   1.]]
Cube vertices shape: (8, 3)
Projected points shape: (8, 2)
```

```
# Camera intrinsics
# Define camera intrinsic matrix
f = 500  # focal length
c_x = 320  # principal point x
c_y = 240  # principal point y
K = np.array([[f, 0, c_x],
              [0, f, c_y],
              [0, 0, 1]])

print("Intrinsic matrix K:")
print(K)

# Create a 3D cube centered at the origin
cube_size = 1
cube_vertices = np.array([[-cube_size/2, -cube_size/2, -cube_size/2],
                          [cube_size/2, -cube_size/2, -cube_size/2],
                          [cube_size/2, cube_size/2, -cube_size/2],
                          [-cube_size/2, cube_size/2, -cube_size/2],
                          [-cube_size/2, -cube_size/2, cube_size/2],
                          [cube_size/2, -cube_size/2, cube_size/2],
                          [cube_size/2, cube_size/2, cube_size/2],
                          [-cube_size/2, cube_size/2, cube_size/2]])

print("Cube vertices shape:", cube_vertices.shape)

# Add translation to move cube in front of camera
translation = np.array([0, 0, 5])
cube_vertices = cube_vertices + translation

# Function to project 3D points to 2D using pinhole camera model
def project_points(points_3d, K):
    # Convert to homogeneous coordinates
    points_3d = np.hstack(points_3d, np.ones(points_3d.shape[0], 1))

    # Apply camera matrix
    points_2d = np.dot(points_3d, K.T)

    # Normalize by depth
    points_2d = points_2d[:, :2] / points_2d[:, 2:]

    return points_2d

# Project cube vertices
projected_points = project_points(cube_vertices, K)

print("Projected points shape:", projected_points.shape)
```

Visualization

Visualize the original cube and the projected points.

Hint:

- Create a 3D plot to show the original cube.
- Create a 2D plot to show the projected points.
- Use different colors for the original and projected points.
- Add labels and titles to the plots.

Output:

```
# Visualization
fig = plt.figure(figure_size="0" box="0.066,0.871,0.270,0.939" kind="chart" usefulness="7" />

### Rotation around the Y-axis

Implement rotation around the Y-axis and visualize the results.

**Hint:**

- Define a rotation matrix for rotation around the Y-axis.
- Apply the rotation to the cube vertices.
- Project the rotated cube onto the image plane.
- Visualize the original and rotated cube.

**Output:**

```
Rotation matrix R_y:
[[ 0.  0.  1.]
 [ 0.  1.  0.]
 [-1.  0.  0.]]
```

```
# Rotation around Y-axis
def rotation_y(angle):
    return np.array([[np.cos(angle), 0, np.sin(angle)],
                     [0, 1, 0],
                     [-np.sin(angle), 0, np.cos(angle)]])

# Apply rotation to cube vertices
R = rotation_y(np.pi/2)
rotated_cube = np.dot(cube_vertices - translation, R.T) + translation

# Project rotated cube
projected_rotated = project_points(rotated_cube, K)

# Visualization
fig = plt.figure(figure_size="0" box="0.066,0.939,0.270,1.000" kind="chart" usefulness="7" />

### Translation

Implement translation and visualize the results.

**Hint:**

- Define a translation vector.
- Apply the translation to the cube vertices.
- Project the translated cube onto the image plane.
- Visualize the original and translated cube.

**Output:**

```
Translation vector: [2 1 3]
```

```
# Translation
translation = np.array([2, 1, 3])

# Apply translation to cube vertices
translated_cube = cube_vertices + translation

# Project translated cube
projected_translated = project_points(translated_cube, K)

# Visualization
fig = plt.figure(figure_size="0" box="0.066,0.000,0.270,0.068" kind="chart" usefulness="7" />

### Rotation and Translation

Implement rotation and translation and visualize the results.

**Hint:**

- Define a rotation matrix for rotation around the Y-axis.
- Define a translation vector.
- Apply the rotation and translation to the cube vertices.
- Project the transformed cube onto the image plane.
- Visualize the original and transformed cube.

**Output:**

```
Rotation matrix R_y:
[[ 0.  0.  1.]
 [ 0.  1.  0.]
 [-1.  0.  0.]]
Translation vector: [2 1 3]
```

```
# Rotation and translation
# Define rotation around Y-axis
def rotation_y(angle):
    return np.array([[np.cos(angle), 0, np.sin(angle)],
                     [0, 1, 0],
                     [-np.sin(angle), 0, np.cos(angle)]])

R = rotation_y(np.pi/2)

# Define translation
translation = np.array([2, 1, 3])

# Apply rotation and translation to cube vertices
transformed_cube = np.dot(cube_vertices - translation, R.T) + translation

# Project transformed cube
projected_transformed = project_points(transformed_cube, K)

# Visualization
fig = plt.figure(figure_size="0" box="0.066,0.068,0.270,0.136" kind="chart" usefulness="7" />

## Problem 5: Stereo Vision (20 points)

### Stereo Vision

Implement stereo vision to calculate depth from disparity.

**Hint:**

- Define the camera intrinsic matrix $K$ with focal length $f$ and principal point $(c_x, c_y)$.
- Create a 3D point cloud.
- Project the point cloud onto the image plane.
- Calculate the disparity.
- Calculate the depth.
- Visualize the depth.

**Output:**

```
Intrinsic matrix K:
[[ 500.   0. 320.]
 [   0. 500. 240.]
 [   0.   0.   1.]]
Point cloud shape: (100, 3)
Projected points shape: (100, 2)
Disparity shape: (100)
Depth shape: (100)
```

```
# Stereo vision
# Define camera intrinsic matrix
f = 500  # focal length
c_x = 320  # principal point x
c_y = 240  # principal point y
K = np.array([[f, 0, c_x],
              [0, f, c_y],
              [0, 0, 1]])

print("Intrinsic matrix K:")
print(K)

# Create a point cloud
np.random.seed(42)
point_cloud = np.random.rotation_value="0" box="0.066,0.136,0.270,0.204" kind="chart" usefulness="7" />

### Visualization

Visualize the point cloud and the projected points.

**Hint:**

- Create a function to visualize the point cloud and the projected points.
- Use different colors for the point cloud and the projected points.
- Add labels and titles to the plots.

**Output:**

```
# Visualization
def visualize_point_cloud(points, projected, title):
    fig = plt.figure(figure_size="0" box="0.066,0.272,0.270,0.340" kind="chart" usefulness="7" />