Part 3: Single-View Geometry

Usage

This code snippet provides an overall code structure and some interactive plot interfaces for the *Single-View Geometry* section of Assignment 3. In main function, we outline the required functionalities step by step. Some of the functions which involves interactive plots are already provided, but the rest are left for you to implement.

Package installation

- In this code, we use `tkinter` package. Installation instruction can be found here.

Common imports

In [1]:
```
%matplotlib tk
import matplotlib.pyplot as plt
import numpy as np
from sympy import *
from sympy import solve
from PIL import Image
```

Provided functions

In [2]:
```
def get_input_lines(im, min_lines=3):
    """
    Allows user to input line segments; computes centers and directions.
    Inputs:
        im: np.ndarray of shape (height, width, 3)
        min_lines: minimum number of lines required
    Returns:
        n: number of lines from input
        lines: np.ndarray of shape (3, n)
            where each column denotes the parameters of the line equation
        centers: np.ndarray of shape (3, n)
            where each column denotes the homogeneous coordinates of the centers
    """
    n = 0
    lines = np.zeros((3, 0))
    centers = np.zeros((3, 0))

    plt.figure()
    plt.imshow(im)
    plt.show()
```

```python
    print('Set at least %d lines to compute vanishing point' % min_lines)
    while True:
        print('Click the two endpoints, use the right key to undo, and use the midd
        clicked = plt.ginput(2, timeout=0, show_clicks=True)
        if not clicked or len(clicked) < 2:
            if n < min_lines:
                print('Need at least %d lines, you have %d now' % (min_lines, n))
                continue
            else:
                # Stop getting lines if number of lines is enough
                break

        # Unpack user inputs and save as homogeneous coordinates
        pt1 = np.array([clicked[0][0], clicked[0][1], 1])
        pt2 = np.array([clicked[1][0], clicked[1][1], 1])
        # Get line equation using cross product
        # Line equation: line[0] * x + line[1] * y + line[2] = 0
        line = np.cross(pt1, pt2)
        lines = np.append(lines, line.reshape((3, 1)), axis=1)
        # Get center coordinate of the line segment
        center = (pt1 + pt2) / 2
        centers = np.append(centers, center.reshape((3, 1)), axis=1)

        # Plot line segment
        plt.plot([pt1[0], pt2[0]], [pt1[1], pt2[1]], color='b')

        n += 1

    return n, lines, centers
```

In [3]:
```python
def plot_lines_and_vp(im, lines, vp):
    """
    Plots user-input lines and the calculated vanishing point.
    Inputs:
        im: np.ndarray of shape (height, width, 3)
        lines: np.ndarray of shape (3, n)
            where each column denotes the parameters of the line equation
        vp: np.ndarray of shape (3, )
    """
    bx1 = min(1, vp[0] / vp[2]) - 10
    bx2 = max(im.shape[1], vp[0] / vp[2]) + 10
    by1 = min(1, vp[1] / vp[2]) - 10
    by2 = max(im.shape[0], vp[1] / vp[2]) + 10

    plt.figure()
    plt.imshow(im)
    for i in range(lines.shape[1]):
        if lines[0, i] < lines[1, i]:
            pt1 = np.cross(np.array([1, 0, -bx1]), lines[:, i])
            pt2 = np.cross(np.array([1, 0, -bx2]), lines[:, i])
        else:
            pt1 = np.cross(np.array([0, 1, -by1]), lines[:, i])
            pt2 = np.cross(np.array([0, 1, -by2]), lines[:, i])
        pt1 = pt1 / pt1[2]
        pt2 = pt2 / pt2[2]
        plt.plot([pt1[0], pt2[0]], [pt1[1], pt2[1]], 'g')
```

```
    plt.plot(vp[0] / vp[2], vp[1] / vp[2], 'ro')
    plt.show()
```

In [4]:
```
def get_top_and_bottom_coordinates(im, obj):
    """
    For a specific object, prompts user to record the top coordinate and the bottom
    Inputs:
        im: np.ndarray of shape (height, width, 3)
        obj: string, object name
    Returns:
        coord: np.ndarray of shape (3, 2)
            where coord[:, 0] is the homogeneous coordinate of the top of the objec
            coordinate of the bottom
    """
    plt.figure()
    plt.imshow(im)

    print('Click on the top coordinate of %s' % obj)
    clicked = plt.ginput(1, timeout=0, show_clicks=True)
    x1, y1 = clicked[0]
    # Uncomment this line to enable a vertical line to help align the two coordinat
    plt.plot([x1, x1], [0, im.shape[0]], 'b')
    print('Click on the bottom coordinate of %s' % obj)
    clicked = plt.ginput(1, timeout=0, show_clicks=True)
    x2, y2 = clicked[0]

    plt.plot([x1, x2], [y1, y2], 'b')

    return np.array([[x1, x2], [y1, y2], [1, 1]])
```

Your implementation

In [5]:
```
def get_vanishing_point(lines):
    """
    Solves for the vanishing point using the user-input lines.
    """
    intersect1 = np.cross(lines[:, 0], lines[:, 1])
    intersect2 = np.cross(lines[:, 1], lines[:, 2])
    intersect3 = np.cross(lines[:, 0], lines[:, 2])

    # convert to homogeneous coordinate
    intersect1 /= intersect1[-1]
    intersect2 /= intersect2[-1]
    intersect3 /= intersect3[-1]

    intersections = np.vstack((intersect1, intersect2, intersect3))
    vp = np.mean(intersections, axis=0)
    print('vanishing point:', vp)
    return vp
```

In [6]:
```
def get_horizon_line(vpts):
    """
```

```
    Calculates the ground horizon line.
    """
    horizon_line = np.cross(vpts[:, 0], vpts[:, 1])
    scale = np.linalg.norm([horizon_line[0], horizon_line[1]])
    horizon_line = horizon_line/scale
    return horizon_line
```

In [7]:
```
def plot_horizon_line(horizon_line,im):
    """
    Plots the horizon line.
    """
    col = im.shape[1]
    x_array = np.arange(0, col, 1)
    y_array = horizon_line[0]*x_array+horizon_line[2] / (-horizon_line[1])
    plt.figure()
    plt.imshow(im)
    plt.plot(x_array, y_array, 'g')
```

In [8]:
```
def get_camera_parameters(vpts):
    """
    Computes the camera parameters. Hint: The SymPy package is suitable for this.
    """
    vpt0 = vpts[:, 0][:, np.newaxis]
    vpt1 = vpts[:, 1][:, np.newaxis]
    vpt2 = vpts[:, 2][:, np.newaxis]

    #focal length and principal point
    focal_len, x_p, y_p= symbols('focal_len, x_p, y_p')
    CAM_MAT_T = Matrix([[1/focal_len, 0, 0], [0, 1/focal_len, 0], [-x_p/focal_len,
    CAM_MAT = Matrix([[1/focal_len, 0, -x_p/focal_len], [0, 1/focal_len, -y_p/focal

    eq1 = vpt0.T * CAM_MAT_T * CAM_MAT * vpt1
    eq2 = vpt0.T * CAM_MAT_T * CAM_MAT * vpt2
    eq3 = vpt1.T * CAM_MAT_T * CAM_MAT * vpt2

    focal_len, x_p, y_p = solve([eq1[0], eq2[0], eq3[0]], (focal_len, x_p, y_p))[0]

    return abs(focal_len), x_p, y_p
```

In [9]:
```
def get_rotation_matrix(f, u, v, vpts):
    """
    Computes the rotation matrix using the camera parameters.
    """
    vpt0 = vpts[:, 0][:, np.newaxis]
    vpt1 = vpts[:, 1][:, np.newaxis]
    vpt2 = vpts[:, 2][:, np.newaxis]

    K = np.array([[f, 0, u], [0, f, v], [0, 0, 1]]).astype(np.float64)
    K_inv = np.linalg.inv(K)

    r1 = K_inv.dot(vpt1)
    r2 = K_inv.dot(vpt2)
    r3 = K_inv.dot(vpt0)
```

```python
    r1 = r1 / np.linalg.norm(r1)
    r2 = r2 / np.linalg.norm(r2)
    r3 = r3 / np.linalg.norm(r3)

    R = np.concatenate((r1, r2, r3), axis=1)
    return R
```

In [10]:
```python
def estimate_height(coords, obj, person_coords, horizon_line, vpts, im, person_heig
    """
    Estimates height for a specific object using the recorded coordinates. You migh
your report.
    """
    horizon_line = horizon_line/np.linalg.norm([horizon_line[0], horizon_line[1]])

    person = person_coords
    person_top = person[:,0]
    person_bottom = person[:,1]

    object = coords[obj]
    object_top = object[:,0]
    object_bottom = object[:,1]

    bottom_line = np.cross(person_bottom, object_bottom)

    vanishing_point = np.cross(bottom_line, horizon_line)
    vanishing_point = vanishing_point/vanishing_point[-1]

    object_line = np.cross(object_bottom, object_top)
    person_vanish = np.cross(person_top, vanishing_point)
    target_point = np.cross(person_vanish, object_line)
    target_point = target_point/target_point[-1]

    infinite_vpt = vpts[:,2]
    p1_p3 = np.linalg.norm(object_bottom-object_top)
    p2_p4 = np.linalg.norm(infinite_vpt-target_point)
    p3_p4 = np.linalg.norm(object_top-infinite_vpt)
    p1_p2 = np.linalg.norm(object_bottom-target_point)
    ratio = p1_p3*p2_p4 / (p1_p2*p3_p4)

    plt.figure()
    plt.imshow(im)
    col = im.shape[1]
    x_array = np.arange(0, col, 1)
    y_array = horizon_line[0]*x_array+horizon_line[2] / (-horizon_line[1])
    plt.plot(x_array, y_array, 'g')
    plt.plot([vanishing_point[0], person_bottom[0]], [vanishing_point[1], person_bo
    plt.plot([vanishing_point[0], target_point[0]], [vanishing_point[1], target_poi
    plt.plot([vanishing_point[0], object_top[0]], [vanishing_point[1], object_top[1
    plt.plot([person_top[0], person_bottom[0]], [person_top[1], person_bottom[1]],
    plt.plot([object_bottom[0], object_top[0]], [object_bottom[1], object_top[1]],
    plt.plot(vanishing_point[0], vanishing_point[1], 'go')
    plt.show()

    obj_height = ratio * person_height
    return obj_height
```

Main function

In [11]:
```
im = np.asarray(Image.open('images/ECEB.jpg'))

# Part 1
# Get vanishing points for each of the directions
num_vpts = 3
vpts = np.zeros((3, num_vpts))
for i in range(num_vpts):
    print('Getting vanishing point %d' % i)
    # Get at least three lines from user input
    n, lines, centers = get_input_lines(im)
    print("number of lines:", n)
    print("lines values: ", lines)

    # <YOUR IMPLEMENTATION> Solve for vanishing point
    vpts[:, i] = get_vanishing_point(lines)
    # Plot the lines and the vanishing point
    plot_lines_and_vp(im, lines, vpts[:, i])

# <YOUR IMPLEMENTATION> Get the ground horizon line
horizon_line = get_horizon_line(vpts)
# <YOUR IMPLEMENTATION> Plot the ground horizon line
plot_horizon_line(horizon_line, im)
```

```
Getting vanishing point 0
Set at least 3 lines to compute vanishing point
Click the two endpoints, use the right key to undo, and use the middle key to stop i
nput
Click the two endpoints, use the right key to undo, and use the middle key to stop i
nput
Click the two endpoints, use the right key to undo, and use the middle key to stop i
nput
Click the two endpoints, use the right key to undo, and use the middle key to stop i
nput
number of lines: 3
lines values:  [[-5.83520470e+01 -1.39201269e+02 -2.57305650e+02]
 [-4.02840035e+02 -4.06355219e+02 -4.74014187e+02]
 [ 3.24881974e+05  3.17564375e+05  3.57842483e+05]]
vanishing point: [-129.54240386  825.66062852    1.        ]
Getting vanishing point 1
Set at least 3 lines to compute vanishing point
Click the two endpoints, use the right key to undo, and use the middle key to stop i
nput
Click the two endpoints, use the right key to undo, and use the middle key to stop i
nput
Click the two endpoints, use the right key to undo, and use the middle key to stop i
nput
Click the two endpoints, use the right key to undo, and use the middle key to stop i
nput
number of lines: 3
lines values:  [[-2.28920511e+02 -9.06158141e+01 -1.64844286e+01]
 [ 1.06374531e+03  6.53492148e+02  5.89165690e+02]
 [-1.55933544e+05 -2.41375030e+05 -4.15842470e+05]]
vanishing point: [2.97910163e+03 7.84403849e+02 1.00000000e+00]
Getting vanishing point 2
Set at least 3 lines to compute vanishing point
Click the two endpoints, use the right key to undo, and use the middle key to stop i
nput
Click the two endpoints, use the right key to undo, and use the middle key to stop i
nput
Click the two endpoints, use the right key to undo, and use the middle key to stop i
nput
Click the two endpoints, use the right key to undo, and use the middle key to stop i
nput
number of lines: 3
lines values:  [[ 9.35997480e+01  9.35997480e+01  9.42542917e+01]
 [ 1.30908738e+00  6.54543692e-01  0.00000000e+00]
 [-5.85393429e+04 -5.99616808e+04 -6.18758339e+04]]
vanishing point: [ 6.56256396e+02 -2.22047648e+03  1.00000000e+00]
```

In [12]:
```python
#print stuff
print("part 1.b")
print("above")
```

```
part 1.b
above
```

In [13]:
```python
print("part 1.c")
print("horizon_line: ", horizon_line)
print(f"horizon line normalized (horizon_line[0])**2 + (horizon_line[1])**2 = {(hor
```

```
part 1.c
horizon_line:  [ 1.32704632e-02  9.99911944e-01 -8.23868836e+02]
horizon line normalized (horizon_line[0])**2 + (horizon_line[1])**2 = 0.999999999999
9999
```

In [14]:
```
# Part 2
# <YOUR IMPLEMENTATION> Solve for the camera parameters (f, u, v)
f, u, v = get_camera_parameters(vpts)
print("part 2")
print(f"focal len={f}, principal point = ({u}, {v})")
```

```
part 2
focal len=1224.70493169743, principal point = (688.293191666695, 193.453913482437)
```

In [15]:
```
# Part 3
# <YOUR IMPLEMENTATION> Solve for the rotation matrix
R = get_rotation_matrix(f, u, v, vpts)
print("part 3")
print("Rotation matrix =")
print(R)
```

```
part 3
Rotation matrix =
[[ 0.85991118 -0.01183468 -0.51030648]
 [ 0.22182756 -0.8917279   0.39447926]
 [ 0.45972306  0.45241717  0.76418153]]
```

In [16]:
```
# Part 4
# Record image coordinates for each object and store in map
objects = ('person', 'leftside', 'rightside', 'door lamp post', 'right lamp post')
coords = dict()
# for obj in objects:
#     coords[obj] = get_top_and_bottom_coordinates(im, obj)


# since the top is specificed the instructions, it is too hard to exactly click,
# so coordinates are provided (x1 top and X2 bottom)
x1, x2, y1, y2 = 1319, 1332, 803, 988
coords['person'] = np.array(([[x1, x2], [y1, y2], [1, 1]]))

x1, x2, y1, y2 = 371, 358, 315, 867
coords['leftside'] = np.array(([[x1, x2], [y1, y2], [1, 1]]))

x1, x2, y1, y2 = 1870, 1901, 281, 813
coords['rightside'] = np.array(([[x1, x2], [y1, y2], [1, 1]]))

x1, x2, y1, y2 = 1525, 1535, 679, 850
coords['door lamp post'] = np.array(([[x1, x2], [y1, y2], [1, 1]]))

x1, x2, y1, y2 = 1928, 1935, 704, 830
coords['right lamp post'] = np.array(([[x1, x2], [y1, y2], [1, 1]]))
```

In [17]:
```
# <YOUR IMPLEMENTATION> Estimate heights
# 5.5 foot person
print("part 4")
heights = dict()
for obj in objects[1:]:
```

```
    print('Estimating height of %s' % obj)
    height = estimate_height(coords, obj, coords['person'], horizon_line, vpts, im,
    heights[obj] = height
    print(f"Height of {obj} = {height} feet")

avg_lamp_height = (heights['door lamp post'] + heights['right lamp post']) / 2
print(f"Average height of the lamp posts = {avg_lamp_height}")
```

```
part 4
Estimating height of leftside
Height of leftside = 74.53754455605917 feet
Estimating height of rightside
Height of rightside = 236.2277513208527 feet
Estimating height of door lamp post
Height of door lamp post = 20.692143698792133 feet
Estimating height of right lamp post
Height of right lamp post = 22.03910881554558 feet
Average height of the lamp posts = 21.365626257168856
```

In [18]:

```
# 6.0 foot person
print("part 5")
heights = dict()
for obj in objects[1:]:
    print('Estimating height of %s' % obj)
    height = estimate_height(coords, obj, coords['person'], horizon_line, vpts, im,
    heights[obj] = height
    print(f"Height of {obj} = {height} feet")

avg_lamp_height = (heights['door lamp post'] + heights['right lamp post']) / 2
print(f"Average height of the lamp posts = {avg_lamp_height}")
```

```
part 5
Estimating height of leftside
Height of leftside = 81.31368497024638 feet
Estimating height of rightside
Height of rightside = 257.7030014409302 feet
Estimating height of door lamp post
Height of door lamp post = 22.5732476714096 feet
Estimating height of right lamp post
Height of right lamp post = 24.04266416241336 feet
Average height of the lamp posts = 23.30795591691148
```

Extra credit

In [19]:

```
# coordinates from image (x1 top and X2 bottom)
objects_EC = ('person', 'Sculpture', 'Fire hydrant', 'Fur tree', 'Wall')
coords_EC = dict()

x1, x2, y1, y2 = 1319, 1332, 803, 988
coords_EC['person'] = np.array(([[x1, x2], [y1, y2], [1, 1]]))

x1, x2, y1, y2 = 1508, 1512, 697, 834
coords_EC['Sculpture'] = np.array(([[x1, x2], [y1, y2], [1, 1]]))

x1, x2, y1, y2 = 46, 45, 873, 907
coords_EC['Fire hydrant'] = np.array(([[x1, x2], [y1, y2], [1, 1]]))
```

```python
x1, x2, y1, y2 = 1876, 1883, 713, 826
coords_EC['Fur tree'] = np.array(([[x1, x2], [y1, y2], [1, 1]]))

x1, x2, y1, y2 = 734, 733, 913, 1016
coords_EC['Wall'] = np.array(([[x1, x2], [y1, y2], [1, 1]]))
```

In [20]:

```python
# <YOUR IMPLEMENTATION> Estimate heights
# 5.5 foot person
print("Extra credit")
heights = dict()
for obj in objects_EC[1:]:
    print('Estimating height of %s' % obj)
    height = estimate_height(coords_EC, obj, coords_EC['person'], horizon_line, vpt
    heights[obj] = height
    print(f"Height of {obj} = {height} feet")
```

```
Extra credit
Estimating height of Sculpture
Height of Sculpture = 25.405953523167295 feet
Estimating height of Fire hydrant
Height of Fire hydrant = 2.1575130914745078 feet
Estimating height of Fur tree
Height of Fur tree = 23.145805072327903 feet
Estimating height of Wall
Height of Wall = 2.664960654347753 feet
```