A5

May 10, 2018

1 EECS 531: Computer Vision Assignment 5

David Fan
5/2/18

In this notebook we will be translating the geometric computer vision demo from MATLAB into Python and explaining what each section of the code does with math and relevant background.

1.1 Prerequisites

These are different from the demo since the ones in the demo are about setting up the Jupyter MATLAB kernel. Here we'll just be doing our standard imports.

```
In [21]: import numpy as np
         import matplotlib.pyplot as plt
         from mpl_toolkits.mplot3d import Axes3D
         import mpl_toolkits.mplot3d as a3
         from matplotlib import patches
```

1.2 Build a Simple Virtual World

1.2.1 Add something into the world

```
In [3]: def create_points():
            [Z, Y, X] = np.meshgrid([-0.5, 0, 0.5], [-0.5, 0, 0.5], [-0.5, 0., 0.5], indexing =

            # This could easily be wrong. MATLAB matrix to Numpy array conversion can get hair
            # The line below attempts to match the matlab line: `points = [X(:), Y(:), Z(:)];`
            # The difference between row major and column major may cause issues down the line
            points = np.column_stack((X.flatten(order='F'), Y.flatten(order='F'), Z.flatten(or

            # No clue how to replicate MATLAB's Jet function
            colors = None

            return (points, colors)
```

This function is appropriately named. It returns a set of coordinates for points to be plotted later on as well as a color value for each point from the Jet colormap.

1.2.2 Plot the points

```
In [14]: def plot_points(points, colors, size = 50):
             fig, ax = plt.subplots(projection='3d')
             ax.scatter(points[0], points[1], points[2], colors, size)
             return (fig, ax)
```

This is relatively straight forward. It simply creates a 3D scatter plot using a set of points and colors.

1.2.3 Set up a pair of cameras

```
In [5]: def preset_cameras():
            r = 5
            focal_length = 0.06
            width = 256
            height = 256
            film_width = 0.035
            film_height = 0.035

            alpha = np.pi/6
            beta = np.pi/6
            cam1 = {
                'position': [r * np.cos(beta) * np.cos(alpha) ,  r * np.cos(beta) * np.sin(alp
                'target': [0, 0, 0],
                'up': [0, 0, 1],
                'focal_length': focal_length,
                'film_width': film_width,
                'film_height': film_height,
                'width': width,
                'height': height
            }

            alpha = np.pi/3
            beta = np.pi/6
            cam2 = {
                'position': [r * np.cos(beta) * np.cos(alpha) ,  r * np.cos(beta) * np.sin(alp
                'target': [0, 0, 0],
                'up': [0, 0, 1],
                'focal_length': focal_length,
                'film_width': film_width,
                'film_height': film_height,
                'width': width,
                'height': height
            }

            return (cam1, cam2)
```

This function creates two dictionaries representing the two camera objects. The dictionaries store the cameras' values. Specifically: - the camera's position in space - the focal point (target) of the camera - the up direction of the camera, the focal length of the camera - the sensor height and width, and - the number of horizontal and vertical pixels (width and height).

1.2.4 Plot camera

```
In [6]: def camera_coordinate_system(cam):
            # The axis of camera coordinate system
            # prinicipal axis
            zcam = cam['target'] - cam['position']

            # x axis should pend to principal axis and up direction
            xcam = np.cross(zcam, cam['up'])

            # y axis should pend to principal axis and principal axis
            ycam = np.cross(zcam, xcam)

            # normalize to unit vector
            zcam = zcam / np.linalg.norm(zcam)
            xcam = xcam / np.linalg.norm(xcam)
            ycam = ycam / np.linalg.norm(ycam)

            origin = cam['position']

            return (xcam, ycam, zcam, origin)
```

This function computes the coordinate system based on the input camera. Z is along the principal axis, X is perpendicular to the principal axis and runs up, and Y is perpendicular to both. It also returns the origin value of the coordinate system.

```
In [11]: def plot_camera(fig, ax, cam, label = '', color=[0.75, 0.75, 0.75]):
             if label:
                 ax.text(cam['position'][0],cam['position'][1],cam['position'][2], label)

             # Compute the camera coordinate system
             xcam, ycam, zcam, origin = camera_coordinate_system(cam)

             # The four corners of the rectangle
             # on the plane through focal points
             d = np.linalg.norm(cam['target'] - cam['position'])
             x = 0.5 * cam['film_width'] * d / cam['focal_length']
             y = 0.5 * cam['film_height'] * d / cam['focal_length']

             P1 = origin + x * xcam + y * ycam + d * zcam
             P2 = origin + x * xcam - y * ycam + d * zcam
             P3 = origin - x * xcam - y * ycam + d * zcam
             P4 = origin - x * xcam + y * ycam + d * zcam
```

```
            # Function to draw a line segment (p1, p2)
            connect = lambda p1, p2: ax.plot([p1[0], p2[0]], [p1[1], p2[2]], color=color)

            # Plot line connect camera and target
            connect(cam['position'], cam['target'])

            # Plot line connect P1, P2, P3, P4
            Patch()...
```

This plotting function plots a visual representation of the input camera coordinate system. It plots the rectangle that the camera is viewing and draws lines leading back to the origin.

Unfortunately, there's really no good Pythonic equivalent of the matlab Patch function that does what the demo does so it seems my translation efforts end here. Maybe if I had more time I could try to hack a solution together, but unfortunately I don't have time for more than just a direct translation so the notebook won't actually run all together, but the idea should be there.

1.3 Show the Virtual World

```
In [19]: points, colors = create_points()
         cam1, cam2 = preset_cameras()
         # print('points: \n', points)
         # print('colors: \n', colors)
         print('camera 1: \n', cam1)
         print('\ncamera 2: \n', cam2)
```

```
camera 1:
 {'position': [3.7500000000000004, 2.1650635094610964, 2.4999999999999996], 'target': [0, 0, 0

camera 2:
 {'position': [2.1650635094610973, 3.75, 2.4999999999999996], 'target': [0, 0, 0], 'up': [0, 0
```

Not much to say here. Just calling the methods from earlier.

```
In [ ]: def lookthrough(cam):
            ...
```

Not actually sure what the MATLAB does here... It sets the axes to the various camera axes.

```
In [ ]: fig, ax = plot_points(points, colors, 50)
        plot_camera(fig, ax, cam1, 'Cam1', [1, 0, 0])
        plot_camera(fig, ax, cam2, 'Cam2', [0, 0, 1])
        ax.set_title("The virtual world")

        fig, ax = plot_points(points, colors, 50)
        plot_camera(fig, ax, cam1, '', [1, 0, 0])
        lookthrough(cam1)
        ax.set_title("Look through camera 1")
```

```
fig, ax = plot_points(points, colors, 50)
plot_camera(fig, ax, cam2, '', [0, 0, 1])
lookthrough(cam2)
ax.set_title("Look through camera 2")
```

Plots the virtual world and the view through each camera.

1.4 Camera Model

1.4.1 Euclidean transformation matrix

```
In [23]: def ExtrinsicsMtx(cam):
             xcam, ycam, zcam, origin = camera_coordinate_system(cam)
             # Rotation matrix
             R = np.stack([xcam, ycam, zcam])
             M = [R, np.dot(-origin, R)]
             return M
```

The first step in transforming the coordinate system from Camera to World is to perform a Euclidean transformation on the Camera coordinate system.

$$\begin{pmatrix} X_{cam} \\ Y_{cam} \\ Z_{cam} \\ 1 \end{pmatrix} = \begin{bmatrix} R & \mathbf{t} \\ \mathbf{O}^T & 1 \end{bmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

Here R is a 3×3 rotation matrix and $\mathbf{t}$ is a 3×1 translation vector.

1.4.2 Camera calibration matrix

```
In [24]: def IntrinsicsMtx(cam):
             cx = (cam['width'] + 1) * .5
             cy = (cam['height'] + 1) * .5

             fx = cam['focal_length'] * cam['width'] / cam['film_width']
             fy = cam['focal_length'] * cam['height'] / cam['film_height']

             K = np.asarray([[fx, 0, 0],[0, fy, 0],[cx, cy, 1]])
             return K
```

The step above constructs the camera calibration matrix:

$$K = \begin{bmatrix} \alpha_x & & x_0 \\ & \alpha_y & y_0 \\ & & 1 \end{bmatrix}$$

Where α_x and $alpha_y$ are the scaling parameters in the image x and y directions and (x_0, y_0) is the principal point, the point where the optic axis intersects the image plane.
Of note is that the aspect ratio of the image is equal to: α_y / α_x
The camera matrix is important because:

$$\mathbf{x} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \frac{1}{f} \begin{bmatrix} \alpha_x & & x_0 \\ & \alpha_y & y_0 \\ & & 1 \end{bmatrix} \begin{pmatrix} x_{cam} \\ y_{cam} \\ f \end{pmatrix} = K \begin{pmatrix} x_{cam} \\ y_{cam} \\ f \end{pmatrix}$$

1.4.3 Camera matrix

```
In [25]: def CameraMtx(cam):
             M = ExtrinsicsMtx(cam)
             K = IntrinsicsMtx(cam)
             P = np.dot(M, K)
             return P
```

Here we calculate the Camera Matrix P:

$$P = K \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} M$$

and

$$\mathbf{x} = P\mathbf{X}$$

Thus we have defined the $3x4$ projection matrix from Euclidean 3-space to an image.

1.4.4 Generate the image pair

```
In [26]: ...


          File "<ipython-input-26-82cca958dda9>", line 1
        def world2image(cam, points3d):
                                       ^

    SyntaxError: unexpected EOF while parsing
```

Here in the demo are a few functions that plot the projected points as images using the above functions. As my translation no longer actually works, there wasn't much point to translating this plotting code. If the above ever gets fixed then the plotting code should go here.

1.5 Triangulation

"Given the cameras and point pairs, reconstruct the 3D positions in world coordinates of the point pairs."

1.5.1 Linear triangulation method

```
In [27]: def triangulate(points1, points2, P1, P2):
             num_points = points1.size[0]
             points3d = np.zeros((num_points, 3))
```

```
        # iterate over point pairs
        for i in range(1, num_points):
            points3d[i] = triangulationOnePoint(points1(i).T, points2(i).T, P1.T, P2.T)
        return points3d
```

This function is relatively simple as the next helper function is the one that does the heavy lifting. This one just iterates over all point pairs in the two sets of points and passes them to the helper function. The return is used to set the 3D reconstruction coordinates.

The problem of triangulation can be formatted such that:

If we know: * P and P' * $\mathbf{x}$ and $\mathbf{x}'$ We compute the reconstruction, $\hat{x}$.

```
In [30]: def triangulationOnePoint(point1, point2, P1, P2):
             # Construct A
             A = np.zeros((4,4))
             A[0:2] = point1.dot(P1[2]) - P1[0:2]
             A[2:4] = point2.dot(P2[2]) - P2[0:2]

             # Solve the optimization problem: min_x ||Ax|| s.t. ||x||=1
             _,_,V = numpy.linalg.svd(A)
             X = V[:, -1]
             X = X/X[-1]

             # Homogenous -> Inhomogenous
             point3d = X[0:3]
             return point3d
```

Here we are calculating the actual reconstruction of each point for each point pair. We do this by solving the system of linear equations:

$$\begin{bmatrix} x\mathbf{p}^{3T} - \mathbf{p}^{1T} \\ y\mathbf{p}^{3T} - \mathbf{p}^{2T} \\ x'\mathbf{p}'^{3T} - \mathbf{p}'^{1T} \\ x\mathbf{p}'^{3T} - \mathbf{p}'^{2T} \end{bmatrix} \mathbf{x} = 0$$

```
In [ ]: ...
```

The rest is just plotting stuff to see the reconstruction!