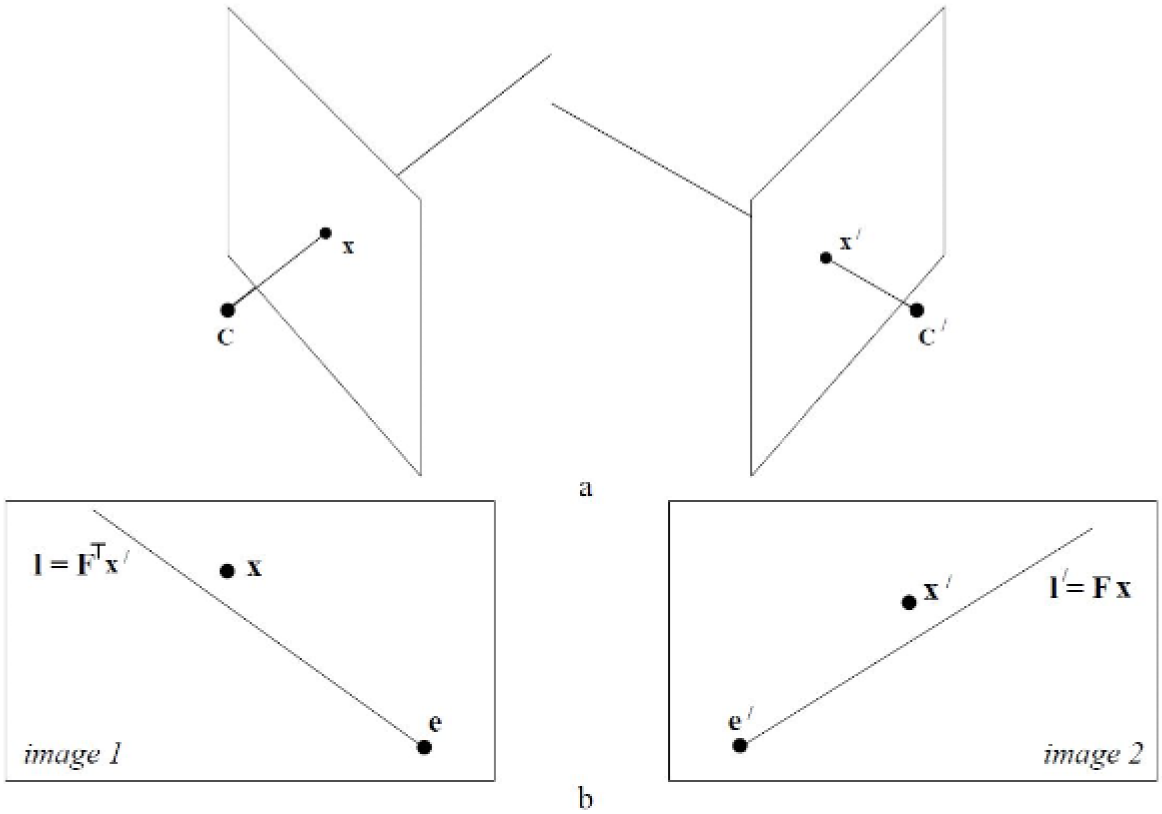
Xiangyi Cheng (xxc283)

Concept and Background

Triangulation will be discussed in this exercise. In trigonometry and geometry, triangulation is the process of determining the location of a point by forming triangles to it form know points. In computer vision, we are using this principle in order to determine the spatial dimensions and the geometry of an item. Specically, two sensors, usually digital cameras, are used to observe the item. The projection centers of the cameras and the target on the surface of the object define a spatial triangle. By determing the angles between the projection ray of the cameras and the base, the 3D coordinates would be calculated based on the triangle relation as shown below.



The principle to estimate a 3D point $\hat{X}$ by two 2D points $\hat{x}$ and $\hat{x}'$ are:

$$\hat{x} = P\hat{X}$$
$$\hat{x}' = P\hat{X}'$$

matrix P is the camera matrix solved in Exercise 2. We will use linear triangulation method to solve the problem. The equations shown above can be combied into a form $AX = 0$, which is an equation linear in X. A is in the form as shown below. With know A, we are able to solve X which represents the estimated 3D coordinates of the object.

$$A = \begin{pmatrix} x\mathbf{P}^{\mathbf{3}\intercal} - \mathbf{P}^{\mathbf{1}\intercal} \\ y\mathbf{P}^{\mathbf{3}\intercal} - \mathbf{P}^{\mathbf{2}\intercal} \\ x'\mathbf{P'}^{\mathbf{3}\intercal} - \mathbf{P'}^{\mathbf{1}\intercal} \\ x'\mathbf{P'}^{\mathbf{3}\intercal} - \mathbf{P'}^{\mathbf{2}\intercal} \end{pmatrix}$$

Implementation

Import the code from Exercise 2 as the preparation part.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from mpl_toolkits.mplot3d.art3d import Poly3DCollection


def define_3Dpoints():

    x=[-0.5, 0, 0.5]
    y=[-0.5, 0, 0.5]
    z=[-0.5, 0, 0.5]
    X, Z, Y = np.meshgrid(x, y, z)

    dimension=X.shape[0]*Y.shape[0]*Z.shape[0]

    colors=np.zeros((dimension,3))
    for i in range (0, dimension):
        colors[i,:]=np.random.rand(3)

    return X,Y,Z,colors


def camera_specification():
    # position parameters
    r= 5
    alpha1= np.pi/6
    beta= np.pi/6

    # camera 1 parameters
    cam1_pos= [r*np.cos(beta)*np.cos(alpha1),
               r*np.cos(beta)*np.sin(alpha1), r*np.sin(beta)]
    target= np.array([0,0,0])
    up= np.array([0,0,1])
    focal_length= 0.06
    film_width= 0.035
    film_height= 0.035
    width= 256
    height= 256

    # camera 2 parameters, others are the same as the camera 1
    alpha2= np.pi/3
    cam2_pos= [r*np.cos(beta)*np.cos(alpha2),
               r*np.cos(beta)*np.sin(alpha2), r*np.sin(beta)]
    return cam1_pos,cam2_pos,target,up,focal_length,
film_height,film_width,width,height


def camera_view_unit(target,cam_pos,up):
    zcam= target-cam_pos
    xcam= np.cross(zcam,up)
    ycam= np.cross(zcam,xcam)
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from mpl_toolkits.mplot3d.art3d import Poly3DCollection
```

```
def define_3Dpoints():

    x=[-0.5, 0, 0.5]
    y=[-0.5, 0, 0.5]
    z=[-0.5, 0, 0.5]
    X, Z, Y = np.meshgrid(x, y, z)

    dimension=X.shape[0]*Y.shape[0]*Z.shape[0]

    colors=np.zeros((dimension,3))
    for i in range (0, dimension):
        colors[i,:]=np.random.rand(3)

    return X,Y,Z,colors


def camera_specification():
    # position parameters
    r= 5
    alpha1= np.pi/6
    beta= np.pi/6

    # camera 1 parameters
    cam1_pos= [r*np.cos(beta)*np.cos(alpha1),
               r*np.cos(beta)*np.sin(alpha1), r*np.sin(beta)]
    target= np.array([0,0,0])
    up= np.array([0,0,1])
    focal_length= 0.06
    film_width= 0.035
    film_height= 0.035
    width= 256
    height= 256

    # camera 2 parameters, others are the same as the camera 1
    alpha2= np.pi/3
    cam2_pos= [r*np.cos(beta)*np.cos(alpha2),
               r*np.cos(beta)*np.sin(alpha2), r*np.sin(beta)]
    return cam1_pos,cam2_pos,target,up,
focal_length,film_height,film_width,width,height


def camera_view_unit(target,cam_pos,up):
    zcam= target-cam_pos
    xcam= np.cross(zcam,up)
    ycam= np.cross(zcam,xcam)

    # normalization
    if np.linalg.norm(xcam)!=0:
        xcam= xcam/np.linalg.norm(xcam)

    if np.linalg.norm(ycam)!=0:
        ycam= ycam/np.linalg.norm(ycam)

    if np.linalg.norm(zcam)!=0:
        zcam= zcam/np.linalg.norm(zcam)
```

```
    return xcam,ycam,zcam


def extrinsic_matrix(camera):
    cam1_pos,cam2_pos,target,up,_,_,_,_,_=camera_specification()

    if camera==1:
        cam_pos=cam1_pos
    elif camera==2:
        cam_pos=cam2_pos
    else:
        print 'camera is not defined.'

    xcam,ycam,zcam=camera_view_unit(target,cam_pos,up)

    rotation_matrix=np.column_stack((xcam,ycam,zcam))
    add=np.dot(np.dot(-1,cam_pos),rotation_matrix)

    extrinsic_matrix=np.vstack([rotation_matrix,add])

    return extrinsic_matrix


def intrinsic_matrix(camera):
    _,_,_,_,focal_length,film_height,film_width,
    width,height=camera_specification()
    cx= 0.5* (width +1)
    cy= 0.5* (height+1)

    fx= focal_length* width /film_width
    fy= focal_length* height/film_height

    intrinsic_matrix= [[fx,0,0],[0,fy,0],[cx,cy,1]]

    return intrinsic_matrix

def camera_matrix(camera,extrinsic_matrix,intrinsic_matrix):
    extrinsic_matrix=extrinsic_matrix(camera)
    intrinsic_matrix=intrinsic_matrix(camera)
    camera_matrix= np.dot(extrinsic_matrix, intrinsic_matrix)

    return camera_matrix


def conv_2Dimage(camera,camera_matrix):
    X,Y,Z,colors=define_3Dpoints()
    X_reshape=X.reshape(1,-1)
    Y_reshape=Y.reshape(1,-1)
    Z_reshape=Z.reshape(1,-1)
    _,dimension=X_reshape.shape

    point=np.column_stack((X_reshape,Y_reshape,
                           Z_reshape,np.ones((1,dimension))))
    point_reshape=np.transpose(point.reshape(4,-1))
    pt=np.dot(point_reshape,camera_matrix)
    object_x=pt[:,0]/pt[:,2]
    object_y=pt[:,1]/pt[:,2]
```

```
    object_2D=np.column_stack((object_x,object_y))

    object_x= np.transpose(object_x)
    object_y= np.transpose(object_y)

    return object_2D,object_x,object_y,colors
    # normalization
    if np.linalg.norm(xcam)!=0:
        xcam= xcam/np.linalg.norm(xcam)

    if np.linalg.norm(ycam)!=0:
        ycam= ycam/np.linalg.norm(ycam)

    if np.linalg.norm(zcam)!=0:
        zcam= zcam/np.linalg.norm(zcam)

    return xcam,ycam,zcam


def extrinsic_matrix(camera):
    cam1_pos,cam2_pos,target,up,_,_,_,_,_=camera_specification()

    if camera==1:
        cam_pos=cam1_pos
    elif camera==2:
        cam_pos=cam2_pos
    else:
        print 'camera is not defined.'

    xcam,ycam,zcam=camera_view_unit(target,cam_pos,up)

    rotation_matrix=np.column_stack((xcam,ycam,zcam))
    add=np.dot(np.dot(-1,cam_pos),rotation_matrix)

    extrinsic_matrix=np.vstack([rotation_matrix,add])

    return extrinsic_matrix


def intrinsic_matrix(camera):
    _,_,_,_,focal_length,film_height,
    film_width,width,height=camera_specification()
    cx= 0.5* (width +1)
    cy= 0.5* (height+1)

    fx= focal_length* width /film_width
    fy= focal_length* height/film_height

    intrinsic_matrix= [[fx,0,0],[0,fy,0],[cx,cy,1]]

    return intrinsic_matrix

def camera_matrix(camera,extrinsic_matrix,intrinsic_matrix):
    extrinsic_matrix=extrinsic_matrix(camera)
    intrinsic_matrix=intrinsic_matrix(camera)
    camera_matrix= np.dot(extrinsic_matrix, intrinsic_matrix)
```

```
    return camera_matrix


def conv_2Dimage(camera,camera_matrix):
    X,Y,Z,colors=define_3Dpoints()
    X_reshape=X.reshape(1,-1)
    Y_reshape=Y.reshape(1,-1)
    Z_reshape=Z.reshape(1,-1)
    _,dimension=X_reshape.shape

    point=np.column_stack((X_reshape,Y_reshape,
                           Z_reshape,np.ones((1,dimension))))
    point_reshape=np.transpose(point.reshape(4,-1))
    pt=np.dot(point_reshape,camera_matrix)
    object_x=pt[:,0]/pt[:,2]
    object_y=pt[:,1]/pt[:,2]

    object_2D=np.column_stack((object_x,object_y))

    object_x= np.transpose(object_x)
    object_y= np.transpose(object_y)

    return object_2D,object_x,object_y,colors
```

For the triangulation at each point, a function called triangulation was developed. A matrix is as we described before,

$$A = \begin{pmatrix} x\mathbf{P}^{\mathbf{3}\intercal} - \mathbf{P}^{\mathbf{1}\intercal} \\ y\mathbf{P}^{\mathbf{3}\intercal} - \mathbf{P}^{\mathbf{2}\intercal} \\ x'\mathbf{P'}^{\mathbf{3}\intercal} - \mathbf{P'}^{\mathbf{1}\intercal} \\ x'\mathbf{P'}^{\mathbf{3}\intercal} - \mathbf{P'}^{\mathbf{2}\intercal} \end{pmatrix}$$

Then $AX = 0$ where X is the 3D coordinates is solved with the known A by applying the SVD (Singular Value Decomposition).

The 3D coordinates of the point is returned in the end of this function.

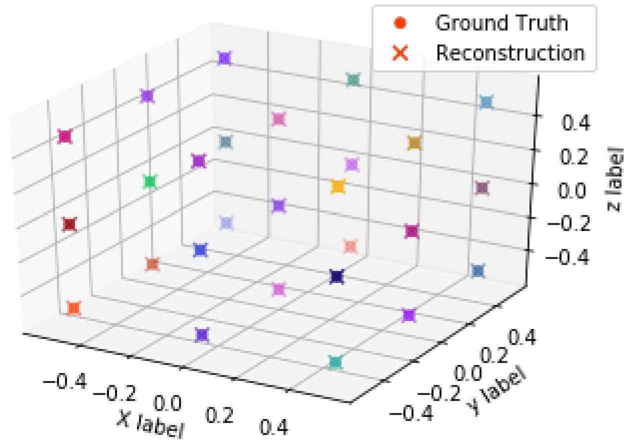
In [2]:
```
def triangulation(object1_2D,object2_2D,camera1_matrix,camera2_matrix):
    A= np.zeros((4,4))

    object1_2D= object1_2D.reshape((2,1))
    object2_2D= object2_2D.reshape((2,1))
    camera1_matrix_r= camera1_matrix[2,:].reshape((1,-1))
    camera2_matrix_r= camera2_matrix[2,:].reshape((1,-1))

    A[0:2,:]=np.dot(object1_2D,camera1_matrix_r) -
    camera1_matrix[0:2,:]
    A[2:4,:]=np.dot(object2_2D,camera2_matrix_r) -
    camera2_matrix[0:2,:]

    _,_,v= np.linalg.svd(A)
    w=v.shape[1]
    X= np.transpose(v)[:,w-1]/np.transpose(v)[w-1,w-1]

    object_3D= X[0:3]

    return object_3D
```

Call the triangulation() at each 2D point of the object in order to reconstruct the 3D location. After recovering each point, plot the result to see if the reconstruction match the true position.

In [14]:
```
def reconstruction3D(object1_2D,object2_2D,camera1_matrix,
                     camera2_matrix):
    object_number=object1_2D.shape[0]
    object_3D= np.zeros((object_number,3))

    for i in range (0,object_number):
        object_3D[i,:]= triangulation(object1_2D[i,:],
            object2_2D[i,:],np.transpose(camera1_matrix),
                                     np.transpose(camera2_matrix))

    X,Y,Z,colors= define_3Dpoints()
    fig=plt.figure()
    ax=fig.add_subplot(111,projection='3d')
    ax.scatter(X,Y,Z,marker='.',c=colors,s=100, label='Ground Truth')

    ax.set_xlabel('X label')
    ax.set_ylabel('y label')
    ax.set_zlabel('z label')
    ax.set_xlim([-0.6,0.6])
    ax.set_ylim([-0.6,0.6])
    ax.set_zlim([-0.6,0.6])

    ax.scatter(object_3D[:,0],object_3D[:,1],object_3D[:,2],

               marker='x',c=colors,s=50, label='Reconstruction')
    ax.legend()
    plt.show()
```

Camera 1 and 2 are given into the function to do the triangulation. The result shows how close the estimated 3D coordinates to the ture coordinates.

In [18]:
```
camera=1
camera1_matrix=camera_matrix(camera,extrinsic_matrix,intrinsic_matrix)
object1_2D,_,_,_=conv_2Dimage(camera,camera1_matrix)

camera=2
camera2_matrix=camera_matrix(camera,extrinsic_matrix,intrinsic_matrix)
object2_2D,_,_,_=conv_2Dimage(camera,camera2_matrix)
reconstruction3D(object1_2D,object2_2D,camera1_matrix,camera2_matrix)
```

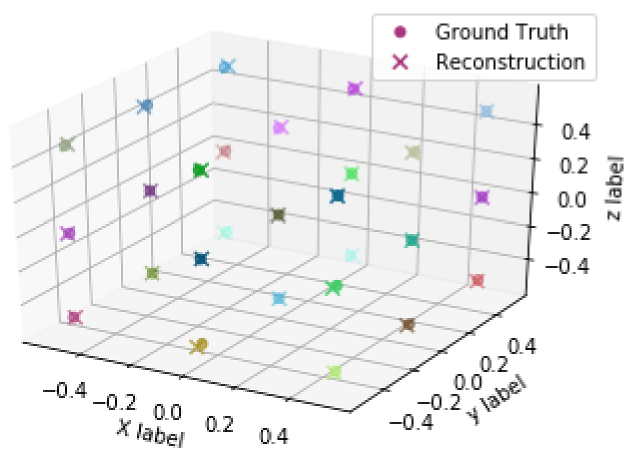


Exploration

In many cases, noises exist in the system so that we want to simulate what the result looks like if there are some noises. The sigma represents the value of the noise as 0.8.

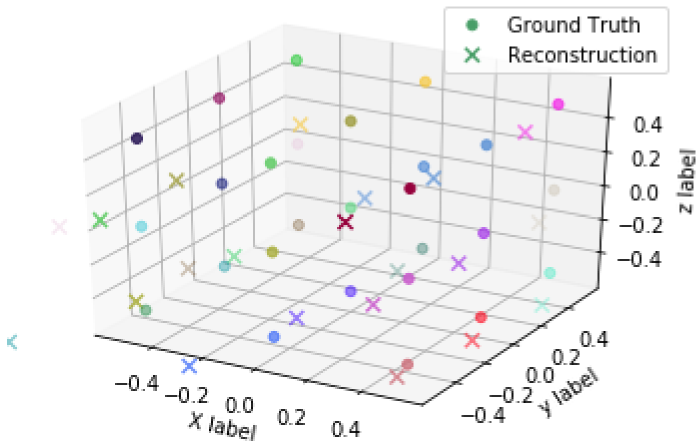
In one aspect, noise is added into the 2D coordinates.

In [16]:
```
sigma=0.8
object1_2D_noise= object1_2D+ sigma* np.random.rand(
    object1_2D.shape[0],object1_2D.shape[1])
object2_2D_noise= object2_2D+ sigma* np.random.rand(
    object1_2D.shape[0],object1_2D.shape[1])
reconstruction3D(object1_2D_noise,object2_2D_noise,
                 camera1_matrix,camera2_matrix)
```



In another aspect, we could add noise into the camera matrix.

In [17]:
```
camera1_matrix_noise= camera1_matrix+ sigma* np.random.rand(
    camera1_matrix.shape[0],camera1_matrix.shape[1])
camera2_matrix_noise= camera2_matrix+ sigma* np.random.rand(
    camera2_matrix.shape[0],camera2_matrix.shape[1])
reconstruction3D(object1_2D,object2_2D,camera1_matrix_noise,
                 camera2_matrix_noise)
```



Conclusion and Discussion

Compared these three results to each other, we could say that the estimated 3D coordinates are really close to the true coordinates if there is no noise into the system, as shown in the first plot. The dots are covered by the sign x which means the locations are realy close that we cannot tell any offsets in naked eyes. Therefore, triangulation is a reasonable approach to consturct objects in 3D world based on several 2D images. However, once there exists noise, the results are different. If the noise is in 2D coordinates, the error which shows in the second plot is acceptable. Some errors can be told but they are still good. But if the noise is in the camera matrix, the error is relatively high compared to the previous two.

To have a reconstructed 3D coordinates with high accuracy, having a good model and reducing the noise are definitely the criticle process to be done.

Reference

[https://en.wikipedia.org/wiki/Triangulation (https://en.wikipedia.org/wiki/Triangulation)](https://en.wikipedia.org/wiki/Triangulation)

Richard Hartley, Andrew Zisserman, "Multiple View Geometry in Computer View", second edition, 2004