Assignment 3 B : Calibration part B (/30pts)

Please edit the cell below to include your name and student ID #

name: Tzu Hsuan Huang

SID: 67913387

NOTE: In this assignment we will build on top of your **project** and **triangulate** functions from previous assignments. If you weren't happy with your implementation from assignment 2, please consult with the TAs or your classmates to get things cleaned up before starting this assignment.

1. Parameterizing 3D Rotations

In order to optimize over the camera rotation during calibration, we need a way to parameterize the space of 3D rotations. There are many different ways to do this and each comes with different tradeoffs, but for our purposes we will adopt a simple approach of building a rotation by a sequence of rotations around the X, Y and Z axes (so called *Tait-Bryan angles*, see https://en.wikipedia.org/wiki/Euler_angles for more discussion)

1.1 Implement [15pts]

Write a function **makerotation** which takes as input three angles **rx,ry,rz** and returns a rotation matrix corresponding to rotating by **rx** degrees around the x-axis, followed by a rotation of **ry** degrees around the y-axis, followed by a rotation of **rz** degrees around the z-axis.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.optimize
import matplotlib.patches as patches
from mpl_toolkits.mplot3d import Axes3D
import visutils
%matplotlib inline
```

In [2]:
```
def makerotation(rx,ry,rz):
    """
    Generate a rotation matrix

    Parameters
    ----------
    rx,ry,rz : floats
        Amount to rotate around x, y and z axes in degrees

    Returns
    -------
    R : 2D numpy.array (dtype=float)
        Rotation matrix of shape (3,3)
    """
```

```
    rx = np.pi*rx/180.0
    ry = np.pi*ry/180.0
    rz = np.pi*rz/180.0
    Rx = np.array([[1,0,0],
                   [0, np.cos(rx), -np.sin(rx)],
                   [0, np.sin(rx), np.cos(rx)]
                  ])
    Ry = np.array([[np.cos(ry), 0, np.sin(ry)],
                   [0,1,0],
                   [-np.sin(ry),0, np.cos(ry)]
                  ])
    Rz = np.array([[np.cos(rz), -np.sin(rz),0],
                   [np.sin(rz), np.cos(rz), 0],
                   [0,0,1]
                  ])
    R = (Rz @ Ry @ Rx)
    return R
```

1.2 Testing [5pts]

Work out by hand what a 90 degree rotation should look like around each axis. Then execute the test examples below and add some tests (asserts) to make sure your code passes.

Find a way to achieve the same rotation as **makerotation(90,90,0)** but without using rotation around the x-axis. That is, determine some angles so that **makerotation(0,?,?) == makerotation(90,90,0)**

In [3]:

```
#
# test your function on some simple examples
#
np.set_printoptions(precision=4,suppress=True)

print(makerotation(90,0,0))

print(makerotation(0,90,0))

print(makerotation(0,0,90))

print(makerotation(90,90,0))

ry = 90
rz = -90
print(makerotation(0,ry,rz))

# figure out what ry,rz values are needed in order to pass this test
assert((makerotation(90,90,0)-makerotation(0,ry,rz)<1e-9).all())
```

```
[[ 1.  0.  0.]
 [ 0.  0. -1.]
 [ 0.  1.  0.]]
[[ 0.  0.  1.]
 [ 0.  1.  0.]
 [-1.  0.  0.]]
[[ 0. -1.  0.]
 [ 1.  0.  0.]
 [ 0.  0.  1.]]
[[ 0.  1.  0.]
 [ 0.  0. -1.]
 [-1.  0.  0.]]
[[ 0.  1.  0.]
 [-0.  0. -1.]
 [-1.  0.  0.]]
```

2. Reprojection Error [20pts]

We will now specify a function which computes the reprojection error. This is the function that we will later optimize when calibrating the camera extrinsic parameters. Take a look at the documentation for **scipy.optimize.leastsq**. The optimizer expects that our function should take a vector of parameters and return a vector of residuals which it will square and sum up to get the total error. For this reason, we will structure our code in the following way.

First, write a member function for the Camera class called **update_extrinsics** which takes a vector of 6 parameters (rx,ry,rz,tx,ty,tz). The function should keep the same intrinsic parameters (f,c) but update the extrinsic parameters (R,t) based on the entries in the parameter vector.

Second, implement a function named **residuals** which computes the difference between a provided set of 2D point coordinates and the projection of 3D point coordinates by specified camera. The residuals function takes as input the 3D points, the target 2D points, a camera with specified intrinsic parameters, and an extrinsic parameter vector. You should use **update_extrinsics** to update the extrinsic parameters, compute the projection of the 3D points with the updated camera and return a 1D vector containing the differences of all the x and y coordinates.

In [4]:
```
class Camera:
    """
    A simple data structure describing camera parameters

    The parameters describing the camera
    cam.f : float   --- camera focal length (in units of pixels)
    cam.c : 2x1 vector  --- offset of principle point
    cam.R : 3x3 matrix --- camera rotation
    cam.t : 3x1 vector --- camera translation

    """

    def __init__(self,f,c,R,t):
        self.f = f
        self.c = c
        self.R = R
        self.t = t

    def __str__(self):
        return f'Camera : \n f={self.f} \n c={self.c.T} \n R={self.R} \n t = {self.t.T}'

    def project(self,pts3):
        """
        Project the given 3D points in world coordinates into the specified camera

        Parameters
        ----------
        pts3 : 2D numpy.array (dtype=float)
            Coordinates of N points stored in a array of shape (3,N)

        Returns
        -------
        pts2 : 2D numpy.array (dtype=float)
            Image coordinates of N points stored in an array of shape (2,N)

        """
        assert(pts3.shape[0]==3)
```

```
        #inverse R
        R_inv = np.linalg.inv(self.R)
        # invR(P world - t)
        pts3_cam = pts3 - self.t
        pts3_cam = R_inv @ pts3_cam
        #divide by camZ and multiply f
        pts3_pixel = pts3_cam[:2, :] / pts3_cam[2, :]
        pts3_pixel = pts3_pixel * self.f
        pts2 = pts3_pixel + self.c
        assert(pts2.shape[1]==pts3.shape[1])
        assert(pts2.shape[0]==2)

        return pts2


    def update_extrinsics(self,params):
        """
        Given a vector of extrinsic parameters, update the camera
        to use the provided parameters.

        Parameters
        ----------
        params : 1D numpy.array of shape (6,) (dtype=float)
            Camera parameters we are optimizing over stored in a vector
            params[:3] are the rotation angles, params[3:] are the translation

        """
        rx = params[0]
        ry = params[1]
        rz = params[2]
        tx = params[3]
        ty = params[4]
        tz = params[5]
        R = makerotation(rx, ry, rz)
        assert(R.shape == (3,3))
        self.R = R
        T = np.array([[tx],[ty],[tz]])
        self.t = T
```

In [5]:

```
def residuals(pts3,pts2,cam,params):
    """
    Compute the difference between the projection of 3D points by the camera
    with the given parameters and the observed 2D locations

    Parameters
    ----------
    pts3 : 2D numpy.array (dtype=float)
        Coordinates of N points stored in a array of shape (3,N)

    pts2 : 2D numpy.array (dtype=float)
        Coordinates of N points stored in a array of shape (2,N)

    params : 1D numpy.array (dtype=float)
        Camera parameters we are optimizing stored in a vector of shape (6,)

    Returns
    -------
    residual : 1D numpy.array (dtype=float)
        Vector of residual 2D projection errors of size 2*N

    """
    #... make sure to flatten into  vector before returning, so it works with scipy.optimize.
    cam.update_extrinsics(params)
```

```
    newpts2 = cam.project(pts3)
    result = pts2 - newpts2
    result = result.flatten('C')

    return result
```

In [6]:

```
#
# Test the residual function to make sure it is doing the right thing.
#

# create two cameras with same intrinsic but slightly different extrinsic parameters
c = [50,50]
camA = Camera(f=200,c=np.array([c]).T,t=np.array([[0,0,0]]).T, R=makerotation(0,0,0))
camB = Camera(f=200,c=np.array([c]).T,t=np.array([[0,0,0]]).T, R=makerotation(0,0,0))

paramsA = np.array([0,0,0,0.5,0.5,-2.5])
paramsB = np.array([0,0,5,0.5,0.5,-3])
camA.update_extrinsics(paramsA)
camB.update_extrinsics(paramsB)

print(camA)
print(camB)

# create a test object (corners of a 3D cube)
pts3 = np.array([[0,0,0],[0,0,1],[0,1,1],[0,1,0],[1,1,0],[1,0,0],[1,0,1],[1,1,1]]).T
print(pts3)
# visualize the two projections
pts2A = camA.project(pts3)
pts2B = camB.project(pts3)

plt.plot(pts2A[0,:],pts2A[1,:],'r')
plt.plot(pts2B[0,:],pts2B[1,:],'b')
#visualize the sensor frame assuming our sensor is centered around the principal point
plt.gca().add_patch(patches.Rectangle((0,0),2*c[0],2*c[1],color='g',fill=False,ls='--'))
plt.plot(c[0],c[1],'g+')
plt.axis('square')
plt.show()

# double check that the residuals are the same as the difference in the reprojected coordinat
print("\n residuals of camB relative to camA")
print(residuals(pts3,pts2A,camB,paramsB))
print(pts2A-pts2B)

print("\n residuals of camA relative to camB")
print(residuals(pts3,pts2B,camA,paramsA))
print(pts2B-pts2A)
```

```
Camera :
 f=200
 c=[[50 50]]
 R=[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
 t = [[ 0.5  0.5 -2.5]]
Camera :
 f=200
 c=[[50 50]]
 R=[[ 0.9962 -0.0872  0.    ]
 [ 0.0872  0.9962  0.    ]
 [ 0.      0.      1.    ]]
 t = [[ 0.5  0.5 -3. ]]
[[0 0 0 0 1 1 1 1]
 [0 0 1 1 1 0 0 1]
 [0 1 1 0 0 0 1 1]]
```

```
 residuals of camB relative to camA
[-3.8883 -1.4877 -5.8455 -9.6987  3.8883  9.6987  5.8455  1.4877 -9.6987
 -5.8455  1.4877  3.8883  9.6987 -3.8883 -1.4877  5.8455]
[[-3.8883 -1.4877 -5.8455 -9.6987  3.8883  9.6987  5.8455  1.4877]
 [-9.6987 -5.8455  1.4877  3.8883  9.6987 -3.8883 -1.4877  5.8455]]

 residuals of camA relative to camB
[ 3.8883  1.4877  5.8455  9.6987 -3.8883 -9.6987 -5.8455 -1.4877  9.6987
  5.8455 -1.4877 -3.8883 -9.6987  3.8883  1.4877 -5.8455]
[[ 3.8883  1.4877  5.8455  9.6987 -3.8883 -9.6987 -5.8455 -1.4877]
 [ 9.6987  5.8455 -1.4877 -3.8883 -9.6987  3.8883  1.4877 -5.8455]]
```

3. Camera Pose Estimation

We are now ready to estimate camera pose using optimize. Implement a function **calibratePose** which takes as input the 3D coordinates of a calibration object, the observed 2D coordinates in the image, and an initial guess of the camera. Your function should use **scipy.optimize.leastsq** to optimize the extrinsic parameters in order to minimize the reprojection error. Since the **residuals** function takes additional arguments and **leastsq** expects a function which only takes the parameter vector as input, you should use Python's **lambda** function to wrap **residuals**, subistituting in the parameters that are fixed during the optimization. Once you have determined the optimum parameters, update the extrinsic parameters to the optimum and return the resulting camera.

3.1 Implementation [30pts]

In [7]:
```
def calibratePose(pts3,pts2,cam_init,params_init):
    """
    Calibrate the provided camera by updating R,t so that pts3 projects
    as close as possible to pts2

    Parameters
    ----------
```

```
    pts3 : 2D numpy.array (dtype=float)
        Coordinates of N points stored in a array of shape (3,N)

    pts2 : 2D numpy.array (dtype=float)
        Coordinates of N points stored in a array of shape (2,N)

    cam : Camera
        Initial estimate of camera

    params_init : 1D numpy.array (dtype=float)
        Initial estimate of camera extrinsic parameters ()
        params[0:3] are the rotation angles, params[3:6] are the translation

    Returns
    -------
    cam : Camera
        Refined estimate of camera with updated R,t parameters

    """

    # use scipy.optimize.leastsq to optimize for params and return the new camera ...
    func = lambda x: residuals(pts3,pts2,cam_init,x)
    optim_params, _ = scipy.optimize.leastsq(func, params_init)
    assert(optim_params.shape==(6,))
    cam_init.update_extrinsics(optim_params)
    return cam_init
```

3.2 Synthetic Test Example and Failure Cases [10pts]

Use the code below to check that your calibrate function works. Add some code to also visualize the point locations in 3D and the location and orientation of the camera (i.e., using the 3D plotting functions from Assignment 2)

Once you are confident that your calibration function is behaving correctly, you should experiment with changing the initial parameters. Find a set of initial parameters which yields a ***wrong*** solution (i.e. where the Final Camera is not similar to the True Camera). In the text box below indicate what bad initialization you used and the resulting set of camera parameters after the optimization. Give a brief explanation of where this bad camera is located and what direction it is oriented in.

In [8]:
```
# 3D calibration object
pts3 = np.array([[0,0,0],[0,0,1],[0,1,1],[0,1,0],[1,1,0],[1,0,0],[1,0,1],[1,1,1]]).T

# true camera
c = [50,50]
cam_true = Camera(f=50,c=np.array([c]).T,t=np.array([[-0.25,-0.25,-2]]).T, R=makerotation(10,

print("\n True Camera")
print(cam_true)

# image of calibration object with some simulated noise in the 2D locations
pts2 = cam_true.project(pts3)
noiselevel = 0.5
pts2 = pts2 + noiselevel*np.random.randn(pts2.shape[0],pts2.shape[1])

# initial guess of camera params
cam = Camera(f=50,c=np.array([[50,50]]).T,t=np.array([[0,0,0]]).T, R=makerotation(0,0,0))
```

```
params_init = np.array([0,0,0,0,0,-2])
cam.update_extrinsics(params_init)

print("\n Initial Camera")
print(cam)
pts2init = cam.project(pts3)

# now run calibration
cam = calibratePose(pts3,pts2,cam,params_init)

print("\n Final Camera")
print(cam)
pts2final = cam.project(pts3)

#
# Plot the true, initial and final reprojections
# The final reprojection should be on top of the true image
#
plt.plot(pts2[0,:],pts2[1,:],'bo')
plt.plot(pts2init[0,:],pts2init[1,:],'r')
plt.plot(pts2final[0,:],pts2final[1,:],'k')
plt.gca().add_patch(patches.Rectangle((0,0),2*c[0],2*c[1],color='g',fill=False,ls='--'))
plt.plot(c[0],c[1],'g+')
plt.axis('square')
plt.show()

#
# Add some additional visualiztion here to show the points in 3D and the locations and orient
# of cam_true and cam.  You can either use a 3D plot or show multiple 2D plots (e.g. overhead
# and side views).  Play with this! Maybe try the visualization tools in Open3D if you want e
#
# generate coordinates of a line segment running from the center
# of the camera to 2 units in front of the camera
look_true = np.hstack((cam_true.t, cam_true.t +cam_true.R @ np.array([[0,0,2]]).T))
look = np.hstack((cam.t,cam.t+cam.R@ np.array([[0,0,2]]).T))

# visualize the image overlaid
fig = plt.figure()
ax = fig.add_subplot(2,2,1)
ax.plot(pts2[0,:],pts2[1,:],'bo')
ax.plot(pts2init[0,:],pts2init[1,:],'r')
ax.plot(pts2final[0,:],pts2final[1,:],'k')
plt.axis('equal')
plt.legend(('pts2','pts2init','pts2final'),loc=1)

#visualize 3D layout of points, camera positions
ax = fig.add_subplot(2,2,2,projection='3d')
ax.plot(pts3[0,:],pts3[1,:],pts3[2,:],'r')
ax.plot(cam_true.t[0],cam_true.t[1],cam_true.t[2],'ro')
ax.plot(cam.t[0],cam.t[1],cam.t[2],'bo')
ax.plot(look_true[0,:],look_true[1,:],look_true[2,:],'r')
ax.plot(look[0,:],look[1,:],look[2,:],'b')
visutils.set_axes_equal_3d(ax)
visutils.label_axes(ax)
plt.title('scene 3D view')

# overhead view showing points, camera
# positions, and direction camera is pointed
ax = fig.add_subplot(2,2,3)
ax.plot(pts3[0,:],pts3[2,:],'r')
ax.plot(cam_true.t[0],cam_true.t[2],'ro')
ax.plot(look_true[0,:],look_true[2,:],'r')
ax.plot(cam.t[0],cam.t[2],'bo')
ax.plot(look[0,:],look[2,:],'b')
```

```
plt.axis('equal')
plt.grid()
plt.xlabel('x')
plt.ylabel('z')
plt.title('scene overhead view')
```

```
 True Camera
Camera :
 f=50
 c=[[50 50]]
 R=[[ 1.      0.      0.    ]
 [ 0.      0.9848 -0.1736]
 [ 0.      0.1736  0.9848]]
 t = [[-0.25 -0.25 -2.  ]]

 Initial Camera
Camera :
 f=50
 c=[[50 50]]
 R=[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
 t = [[ 0  0 -2]]

 Final Camera
Camera :
 f=50
 c=[[50 50]]
 R=[[ 0.9997  0.0047 -0.024 ]
 [-0.0094  0.9796 -0.2009]
 [ 0.0225  0.201   0.9793]]
 t = [[-0.1977 -0.1702 -2.037 ]]
```

Out[8]:

```
Text(0.5, 1.0, 'scene overhead view')
```

In [9]:

```
#
# Now repeat the calibration but with a setting for params_init that results
# in the optimization finding a poor solution (a bad local minima)
#

#
# Visualize the resulting bad solution.
#
# 3D calibration object
pts3 = np.array([[0,0,0],[0,0,1],[0,1,1],[0,1,0],[1,1,0],[1,0,0],[1,0,1],[1,1,1]]).T

# true camera
c = [50,50]
cam_true = Camera(f=50,c=np.array([c]).T,t=np.array([[-0.25,-0.25,-2]]).T, R=makerotation(10,

print("\n True Camera")
print(cam_true)

# image of calibration object with some simulated noise in the 2D locations
pts2 = cam_true.project(pts3)
noiselevel = 0
pts2 = pts2 + noiselevel*np.random.randn(pts2.shape[0],pts2.shape[1])

# initial guess of camera params
cam = Camera(f=50,c=np.array([[50,50]]).T,t=np.array([[0,0,0]]).T, R=makerotation(0,0,0))
params_init = np.array([1,2,3,-4,5,6])
cam.update_extrinsics(params_init)

print("\n Initial Camera")
print(cam)
pts2init = cam.project(pts3)

# now run calibration
cam = calibratePose(pts3,pts2,cam,params_init)
```

```
print("\n Final Camera")
print(cam)
pts2final = cam.project(pts3)

#
# Plot the true, initial and final reprojections
# The final reprojection should be on top of the true image
#
plt.plot(pts2[0,:],pts2[1,:],'bo')
plt.plot(pts2init[0,:],pts2init[1,:],'r')
plt.plot(pts2final[0,:],pts2final[1,:],'k')
plt.gca().add_patch(patches.Rectangle((0,0),2*c[0],2*c[1],color='g',fill=False,ls='--'))
plt.plot(c[0],c[1],'g+')
plt.axis('square')
plt.show()

#
# Add some additional visualiztion here to show the points in 3D and the locations and orient
# of cam_true and cam.  You can either use a 3D plot or show multiple 2D plots (e.g. overhead
# and side views).  Play with this! Maybe try the visualization tools in Open3D if you want e
#

# generate coordinates of a line segment running from the center
# of the camera to 2 units in front of the camera
look_true = np.hstack((cam_true.t, cam_true.t +cam_true.R @ np.array([[0,0,2]]).T))
look = np.hstack((cam.t,cam.t+cam.R@ np.array([[0,0,2]]).T))

# visualize the image overlaid
fig = plt.figure()
ax = fig.add_subplot(2,2,1)
ax.plot(pts2[0,:],pts2[1,:],'bo')
ax.plot(pts2init[0,:],pts2init[1,:],'r')
ax.plot(pts2final[0,:],pts2final[1,:],'k')
plt.axis('equal')
plt.legend(('pts2','pts2init','pts2final'),loc=1)

#visualize 3D layout of points, camera positions
ax = fig.add_subplot(2,2,2,projection='3d')
ax.plot(pts3[0,:],pts3[1,:],pts3[2,:],'r')
ax.plot(cam_true.t[0],cam_true.t[1],cam_true.t[2],'ro')
ax.plot(cam.t[0],cam.t[1],cam.t[2],'bo')
ax.plot(look_true[0,:],look_true[1,:],look_true[2,:],'r')
ax.plot(look[0,:],look[1,:],look[2,:],'b')
visutils.set_axes_equal_3d(ax)
visutils.label_axes(ax)
plt.title('scene 3D view')

# overhead view showing points, camera
# positions, and direction camera is pointed
ax = fig.add_subplot(2,2,3)
ax.plot(pts3[0,:],pts3[2,:],'r')
ax.plot(cam_true.t[0],cam_true.t[2],'ro')
ax.plot(look_true[0,:],look_true[2,:],'r')
ax.plot(cam.t[0],cam.t[2],'bo')
ax.plot(look[0,:],look[2,:],'b')
plt.axis('equal')
plt.grid()
plt.xlabel('x')
plt.ylabel('z')
plt.title('scene overhead view')
```

```
 True Camera
Camera :
 f=50
 c=[[50 50]]
 R=[[ 1.      0.      0.    ]
 [ 0.      0.9848 -0.1736]
 [ 0.      0.1736  0.9848]]
 t = [[-0.25 -0.25 -2.  ]]

 Initial Camera
Camera :
 f=50
 c=[[50 50]]
 R=[[ 0.998  -0.0517  0.0358]
 [ 0.0523  0.9985 -0.0156]
 [-0.0349  0.0174  0.9992]]
 t = [[-4  5  6]]

 Final Camera
Camera :
 f=50
 c=[[50 50]]
 R=[[-0.7949  0.3127  0.5199]
 [ 0.2181 -0.6523  0.7259]
 [ 0.5661  0.6904  0.4503]]
 t = [[1.5076 1.6129 2.8529]]
```

Out[9]: Text(0.5, 1.0, 'scene overhead view')

Explain one of the failures including where the initial camera was that led to failure.

I changed params_init from [0,0,0,1,0,-2] to a different rotataion [1,2,3,-4,5,6], which the initial camera is at (-4 5 6) and final camera is located at (1.5076 1.6129 2.8529). The new params make the camera not face toward the object, which leads to failure.

4. Calibration from real images [30pts]

There is a provided set of calibration images (images of a planar checkerboard) along with stereo pair depicting an object. In order to calibrate the intrinsic camera parameters we will use the OpenCV library which includes functionality for automatically detecting corners of the checkerboard and estimating intrinsic parameters.

1. Install OpenCV python libraries in your Anaconda environment. You can do this from the terminal via the command **conda install opencv** or via the Anconda Navigator gui.

2. Read through the provided standalone script **calibrate.py** and modify as necessary in order to compute the intrinsic camera parameters from the set of provided calibration images.

3. Load in the intrinsic parameter calibration data saved by the script in *calibration.pickle*. Since our camera model assumes that the focal length is the same in the x and y axes, you can set your f to be the average of the two estimated by the script.

4. Load in the test images *Left.jpg* and *Right.jpg*

5. Use the **cv2.findChessboardCorners** function in order to automatically get the 2D coordinates of the corners in the image.

6. Specify the 3D coordinates of the 6x8 grid of checkerboard corners. The squares are 2.8cm x 2.8cm.

7. Use your **calibratePose** function to estimate the R,t for each camera. You will like need to experiment with selecting the initial parameters in order to get a good solution (e.g., translate so the cameras have positive z coordinates and rotate so they are looking down on the checkerboard).

8. Finally, once you have the calibrated pose for each camera, use your triangulate function to estimate the 3D coordinates of the checkerboard corners based on the 2D points in the left and right camera.

4.1 Implementation

In [10]:
```
def triangulate(pts2L,camL,pts2R,camR):
    """
    Triangulate the set of points seen at location pts2L / pts2R in the
    corresponding pair of cameras. Return the 3D coordinates relative
    to the global coordinate system


    Parameters
    ----------
    pts2L : 2D numpy.array (dtype=float)
        Coordinates of N points stored in a array of shape (2,N) seen from camL camera

    pts2R : 2D numpy.array (dtype=float)
        Coordinates of N points stored in a array of shape (2,N) seen from camR camera

    camL : Camera
        The first "left" camera view

    camR : Camera
        The second "right" camera view

    Returns
    -------
    pts3 : 2D numpy.array (dtype=float)
        (3,N) array containing 3D coordinates of the points in global coordinates

    """


    #
    # Your code goes here.  I recommend adding assert statements to check the
    # sizes of the inputs and outputs to make sure they are correct
    #
    assert(pts2L.shape[0]==2)
    assert(pts2R.shape[0]==2)

    N= pts2L.shape[1]
    #to get the qL and qR
    new_pts2L = pts2L - camL.c
    new_pts2L /=  camL.f
    oneArr = np.ones(N)
    qL = np.vstack((new_pts2L, oneArr))
    new_pts2R = pts2R - camR.c
    new_pts2R /=  camR.f
    qR = np.vstack((new_pts2R, oneArr))
    pts3 = np.zeros((3,N))
    for i in range(N):
        #Get A matrix
        a = camL.R @ qL[:,i].reshape(3,1)
```

```
        b = -(camR.R @ qR[:,i].reshape(3,1))
        A = np.hstack((a,b))
        assert(A.shape == (3,2))
        #caculate ZR and Zl
        Z = np.linalg.lstsq(A, camR.t-camL.t, rcond=None)[0]
        assert(Z.shape == (2,1))
        tmp_pts3L = qL[:,i].reshape(3,1)*Z[0,0]
        tmp_pts3R = qR[:,i].reshape(3,1)*Z[1,0]
        P1 = camL.R@tmp_pts3L + camL.t
        P2 = camR.R@tmp_pts3R + camR.t
        mean = (P1+P2)/2.0
        pts3[:,i] = mean.flatten()
    assert(pts3.shape[0]==3)



    return pts3
```

In [11]:

```
import cv2
import pickle
# load in the intrinsic camera parameters

fid = open('calibration.pickle','rb')
calib = pickle.load(fid)
fid.close()

# create Camera objects representing the left and right cameras
# use the known intrinsic parameters you loaded in.
getf  =(calib["fx"]+calib["fy"])/2
camL = Camera(f=getf,c=np.array([[calib["cx"],calib["cy"]]]).T,t=np.array([[0,0,0]]).T, R=mak
camR = Camera(f=getf,c=np.array([[calib["cx"],calib["cy"]]]).T,t=np.array([[0,0,0]]).T, R=mak

# load in the left and right images and find the coordinates of
# the chessboard corners using OpenCV
imgL = plt.imread('calib1/Left.jpg')
ret, cornersL = cv2.findChessboardCorners(imgL, (8,6), None)
pts2L = cornersL.squeeze().T

imgR = plt.imread('calib1/Right.jpg')
ret, cornersR = cv2.findChessboardCorners(imgR, (8,6), None)
pts2R = cornersR.squeeze().T

# generate the known 3D point coordinates of points on the checkerboard in cm
# you need to modify this code to take into account that the
# squares are 2.8cm x 2.8cm
pts3 = np.zeros((3,6*8))
xx,yy = np.meshgrid(np.arange(8),np.arange(6))
pts3[0,:] = 2.8*yy.reshape(1,-1)
pts3[1,:] = 2.8*xx.reshape(1,-1)

# Now use your calibratePose function to get the extrinsic parameters
# for the two images. You may need to experiment with the initialization
# in order to get a good result
paramsL_init = np.array([0,-1,0,0,0,-1])
paramsR_init = np.array([0,-1,0,0,0,-1])

camL = calibratePose(pts3,pts2L,camL,paramsL_init)
camR = calibratePose(pts3,pts2R,camR,paramsR_init)

# Finally, triangulate the checkerboard points based on the estimated camera pose
pts3r = triangulate(pts2L,camL,pts2R,camR)

# Display the reprojected points overlayed on the images
plt.rcParams['figure.figsize']=[15,15]
```

```
pts2Lp = camL.project(pts3)
plt.imshow(imgL)
plt.plot(pts2Lp[0,:],pts2Lp[1,:],'bo')
plt.plot(pts2L[0,:],pts2L[1,:],'rx')
plt.show()

pts2Rp = camR.project(pts3)
plt.imshow(imgR)
plt.plot(pts2Rp[0,:],pts2Rp[1,:],'bo')
plt.plot(pts2R[0,:],pts2R[1,:],'rx')
plt.show()

print(camL)
print(camR)
print(np.sqrt(np.sum((camL.t-camR.t)*(camL.t-camR.t))))

print(np.mean(np.sqrt(np.sum((pts3r-pts3)*(pts3r-pts3),axis=0))))
```

0
250
500
750
1000
1250
1500
1750
2000
0
200
400
600
800
1000
1200
1400

0
250
500
750
1000
1250
1500
1750
2000
0
200
400
600
800
1000
1200
1400

```
Camera :
 f=1561.0139703220098
 c=[[1021.1465  755.8365]]
 R=[[ 0.7928  0.5479 -0.2671]
 [ 0.6058 -0.66    0.4443]
 [ 0.0671 -0.514  -0.8552]]
 t = [[20.2533 -0.3845 37.7861]]
Camera :
 f=1561.0139703220098
 c=[[1021.1465  755.8365]]
 R=[[ 0.9506  0.271  -0.1511]
 [ 0.3029 -0.7042  0.6421]
 [ 0.0676 -0.6562 -0.7516]]
 t = [[ 17.4715 -11.9475  24.0513]]
18.16829831699334
0.2375570217550823
```

In [12]:

```
import visutils
import warnings
warnings.filterwarnings("ignore", category=UserWarning)
# generate coordinates of a line segment running from the center
# of the camera to 3 units in front of the camera
lookL = np.hstack((camL.t,camL.t+camL.R @ np.array([[0,0,2]]).T))
lookR = np.hstack((camR.t,camR.t+camR.R @ np.array([[0,0,2]]).T))

# visualize the left and right image overlaid
fig = plt.figure()
ax = fig.add_subplot(2,2,1,projection='3d')
ax.plot(pts3[0,:],pts3[1,:],pts3[2,:],'.')
ax.plot(pts3r[0,:],pts3r[1,:],pts3r[2,:],'rx')
ax.plot(camR.t[0],camR.t[1],camR.t[2],'ro')
ax.plot(camL.t[0],camL.t[1],camL.t[2],'bo')
ax.plot(lookL[0,:],lookL[1,:],lookL[2,:],'b')
ax.plot(lookR[0,:],lookR[1,:],lookR[2,:],'r')
visutils.set_axes_equal_3d(ax)
visutils.label_axes(ax)
plt.title('scene 3D view')

ax = fig.add_subplot(2,2,2)
ax.plot(pts3[0,:],pts3[2,:],'.')
ax.plot(pts3r[0,:],pts3r[2,:],'rx')
ax.plot(camR.t[0],camR.t[2],'ro')
ax.plot(camL.t[0],camL.t[2],'bo')
ax.plot(lookL[0,:],lookL[2,:],'b')
ax.plot(lookR[0,:],lookR[2,:],'r')
plt.title('XZ-view')
plt.grid()
plt.xlabel('x')
plt.ylabel('z')

ax = fig.add_subplot(2,2,3)
ax.plot(pts3[1,:],pts3[2,:],'.')
ax.plot(pts3r[1,:],pts3r[2,:],'rx')
ax.plot(camR.t[1],camR.t[2],'ro')
ax.plot(camL.t[1],camL.t[2],'bo')
ax.plot(lookL[1,:],lookL[2,:],'b')
ax.plot(lookR[1,:],lookR[2,:],'r')
plt.title('YZ-view')
plt.grid()
plt.xlabel('y')
plt.ylabel('z')

ax = fig.add_subplot(2,2,4)
ax.plot(pts3[0,:],pts3[1,:],'.')
```

```
ax.plot(pts3r[0,:],pts3r[1,:],'rx')
ax.plot(camR.t[0],camR.t[1],'ro')
ax.plot(camL.t[0],camL.t[1],'bo')
ax.plot(lookL[0,:],lookL[1,:],'b')
ax.plot(lookR[0,:],lookR[1,:],'r')
plt.title('XY-view')
plt.grid()
plt.xlabel('x')
plt.ylabel('y')
```

Out[12]: Text(0, 0.5, 'y')

The code below provides a visualization of the estimate camera positions relative to the checkerboard.

4.2 Recovered Pose

Using the provided calibration images, what are the recovered parameters for the left and right cameras? How far apart are the camera centers in centimeters (i.e. what is the baseline) ?

In [13]:
```
print("Left",camL)
print("Right",camR)
print("The baseline length is",np.sqrt(np.sum((camL.t-camR.t)*(camL.t-camR.t))), "cm.")
```

```
Left Camera :
 f=1561.0139703220098
 c=[[1021.1465  755.8365]]
 R=[[ 0.7928  0.5479 -0.2671]
 [ 0.6058 -0.66    0.4443]
 [ 0.0671 -0.514  -0.8552]]
 t = [[20.2533 -0.3845 37.7861]]
Right Camera :
 f=1561.0139703220098
 c=[[1021.1465  755.8365]]
 R=[[ 0.9506  0.271  -0.1511]
 [ 0.3029 -0.7042  0.6421]
 [ 0.0676 -0.6562 -0.7516]]
 t = [[ 17.4715 -11.9475  24.0513]]
The baseline length is 18.16829831699334 cm.
```

4.3 Reconsruction Accuracy

What is the average error (in cm) for your recovered 3D locations of the grid corner points relative to their true coordinates? Where might this error be coming from?

In [14]:
```
print("The average error is", np.mean(np.sqrt(np.sum((pts3r-pts3)*(pts3r-pts3),axis=0))),"cm.
print("The error might come from the initial parameters or the scipy.optimize.leastsq functic
```

```
The average error is 0.2375570217550823 cm.
The error might come from the initial parameters or the scipy.optimize.leastsq function, whi
ch minimizes the error, but not accurate.
```

4.4 Focal Length

The photos were taken with an iPhone Xs. Teardowns of this device reveal that the sensor is 5.6mm wide. Based on this and your recovered value for f, what was the focal length in milimeters? Explain how you computed this. Is the result you get a reasonable match to the published focal length of of 4.25mm?

In [15]:
```
ratio = imgL.shape[0]/ imgL.shape[1]
heightsensor = 5.6/ratio
print("We can get Focal Length from sensor size/image size, then multiple by f")
print("Focal Length= ", getf*(5.6/imgL.shape[0]), "mm")
```

```
We can get Focal Length from sensor size/image size, then multiple by f
Focal Length=  4.336149917561138 mm
```

In [ ]: