```
```

In [13]:

```python
import numpy as np
import matplotlib.pyplot as plt
from camutils import Camera,triangulate
import pickle
import visutils
import matplotlib.patches as patches
from mpl_toolkits.mplot3d import Axes3D
import scipy.optimize

from scipy.spatial import Delaunay
#import scipy
from mpl_toolkits.mplot3d import Axes3D
from meshutils import writeply

```

In [14]:

```python
exec(open("calibrate.py").read())#needed to read files and create pickle fil
```

```
Estimated camera intrinsic parameter matrix K
[[1.40532129e+03 0.00000000e+00 9.62163839e+02]
 [0.00000000e+00 1.40390409e+03 5.90925282e+02]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]
Estimated radial distortion coefficients
[[-5.23616631e-03  8.06253000e-02  2.04814121e-05 -3.90754737e-03
  -1.08137152e-01]]
Individual intrinsic parameters
fx =  1405.3212897101798
fy =  1403.9040872333378
cx =  962.1638389973019
cy =  590.9252821271073
```

```
```

In [15]:

```
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)
    """

    #degrees are terrible radians are the only worthwhile measure of an angl

    rotX = rx*np.pi/180
    rotY = ry*np.pi/180
    rotZ = rz*np.pi/180

    XMatrix = np.array([[1,0,0],[0, np.cos(rotX), -1*np.sin(rotX)],[0,np.sin
    YMatrix = np.array([[np.cos(rotY), 0, np.sin(rotY)],[0,1,0],[-1*np.sin(r
    ZMatrix = np.array([[np.cos(rotZ),-1*np.sin(rotZ),0],[np.sin(rotZ),np.co

    return np.matmul(np.matmul(XMatrix,YMatrix),ZMatrix)

```

In [16]:

```python
def calibratePose(pts3,pts2,cam,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[0:2] are the rotation angles, params[2:5] are the translation

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

    """
    cam.update_extrinsics(params_init)
    projPTS = cam.project(pts3)
    paramsFinal = scipy.optimize.leastsq(lambda params:residuals(pts3,pts2,c

    cam.update_extrinsics(paramsFinal[0])
    return cam
```

In [17]:

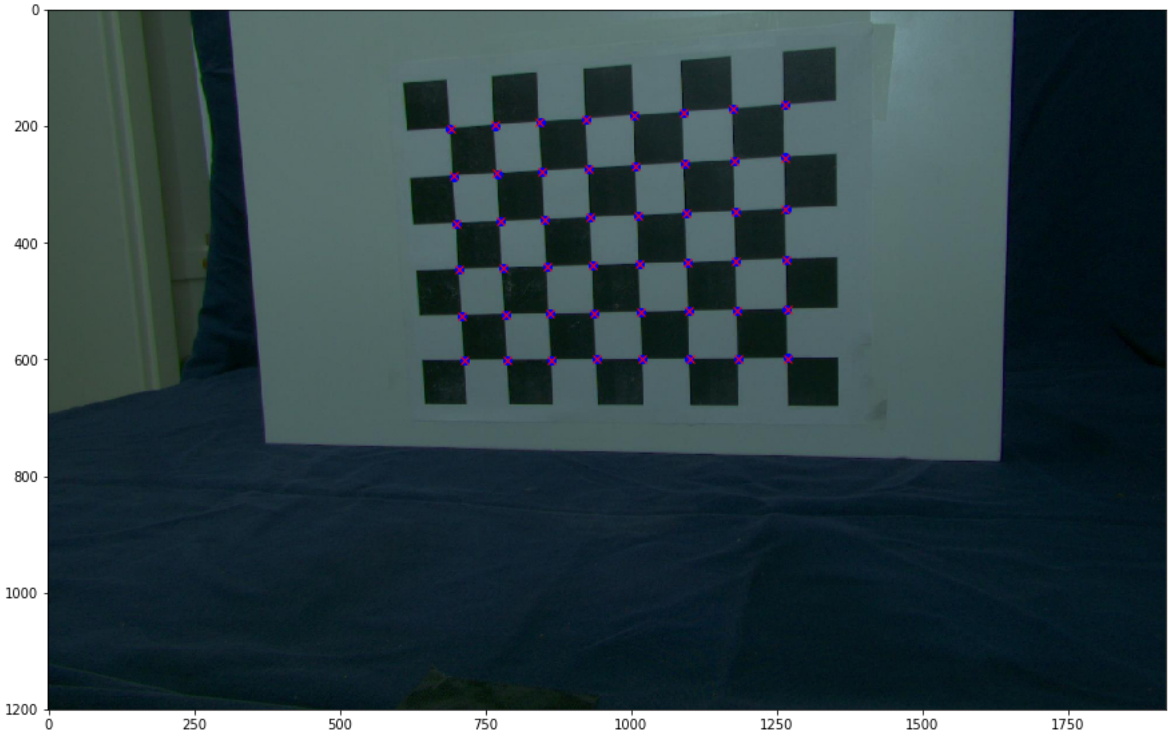
```
def residuals(pts3,pts2,cam,params):
    """
    Compute the difference between the projection of 3D points by the camera
    with the given parameters and the observed 2D locations

    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

    """
    cam.update_extrinsics(params)
    projPTS = cam.project(pts3)
    return np.abs(np.subtract(projPTS,pts2)).flatten()
```

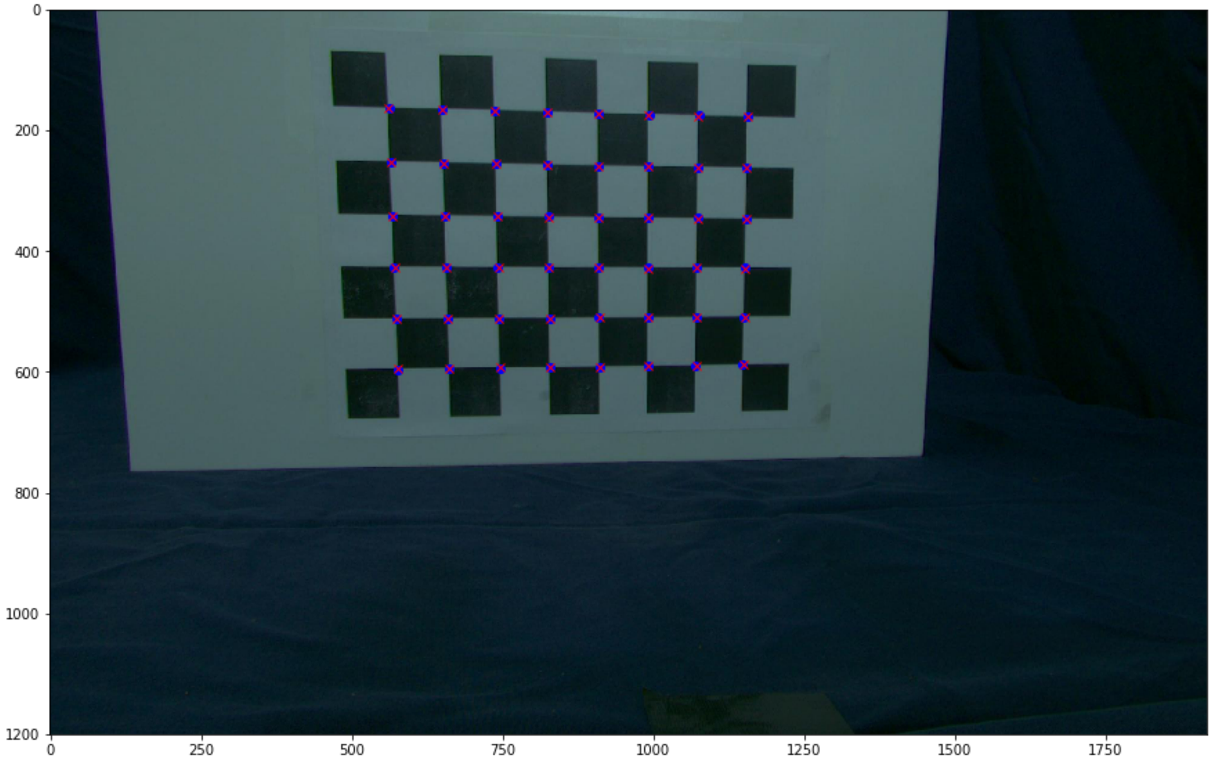
Part 1)

Below is the script finding the calibration of the two cameras,

In [18]:

```
f = open('C:\\calibration.pickle','rb')
data = pickle.load(f)
c = np.array([[data['cx']],[data['cy']]])
focalL = (data['fx']+data['fy'])/2.0
rotation_init = makerotation(0,0,90)
trans_init = [[0],[0],[-40]]

# create Camera objects representing the left and right cameras
# use the known intrinsic parameters you loaded in.
camL = Camera(focalL,c,rotation_init,trans_init)
camR = Camera(focalL,c,rotation_init,trans_init)

# load in the left and right images and find the coordinates of
# the chessboard corners using OpenCV
imgL = plt.imread('C:\\Users\\mattr\\Desktop\\CS117\\calib_jpg_u\\frame_C0_0
ret, cornersL = cv2.findChessboardCorners(imgL, (8,6), None)
pts2L = cornersL.squeeze().T

imgR = plt.imread('C:\\Users\\mattr\\Desktop\\CS117\\calib_jpg_u\\frame_C1_0
ret, cornersR = cv2.findChessboardCorners(imgR, (8,6), None)
pts2R = cornersR.squeeze().T

# generate the known 3D point coordinates of points on the checkerboard in c
pts3 = np.zeros((3,6*8))
yy,xx = np.meshgrid(np.arange(8),np.arange(6))
pts3[0,:] = 2.8*xx.reshape(1,-1)
pts3[1,:] = 2.8*yy.reshape(1,-1)

paramsL = calibratePose(pts3,pts2L,camL,np.array([0,0,90,0,0,-40]))
paramsR = calibratePose(pts3,pts2R,camR,np.array([0,0,90,0,0,-40]))


# As a final test, triangulate the corners of the checkerboard to get back t
#def triangulate(pts2L, camL, pts2R, camR):
pts3r = triangulate(pts2L,camL,pts2R,camR)

# Display the reprojected points overlayed on the images to make
# sure they line up
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()
```


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


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

In [19]:

```
print(camL)
print(camR)
```

```
Camera :
 f=1404.6126884717587
 c=[[962.163839   590.92528213]]
 R=[[ 0.03843677  0.98947349  0.13951639]
 [ 0.97735757 -0.00815333 -0.21143723]
 [-0.20807401  0.14448436 -0.9673828 ]]
 t = [[ 6.86592297 19.52347734 47.34466546]]
Camera :
 f=1404.6126884717587
 c=[[962.163839   590.92528213]]
 R=[[-0.00259822  0.99096856  0.13406922]
 [ 0.99277871 -0.01352312  0.11919546]
 [ 0.11993199  0.13341077 -0.98377736]]
 t = [[ 7.5003338   7.20907829 47.76536838]]
```

Part 2)

In [20]:

```python
def decode(imprefix,start,threshold):
    """
    Given a sequence of 20 images of a scene showing projected 10 bit gray c
    decode the binary sequence into a decimal value in (0,1023) for each pix
    Mark those pixels whose code is likely to be incorrect based on the user
    provided threshold.  Images are assumed to be named "imageprefixN.png" w
    N is a 2 digit index (e.g., "img00.png,img01.png,img02.png...")

    Parameters
    ----------
    imprefix : str
       Image name prefix

    start : int
       Starting index

    threshold : float
       Threshold to determine if a bit is decodeable

    Returns
    -------
    code : 2D numpy.array (dtype=float)
        Array the same size as input images with entries in (0..1023)

    mask : 2D numpy.array (dtype=logical)
        Array indicating which pixels were correctly decoded based on the th

    """

    # we will assume a 10 bit code
    nbits = 10

    # don't forget to convert images to grayscale / float after loading then

    fileEnd = ".png"
    maskShape = (plt.imread(imprefix+"00"+fileEnd).shape[0],plt.imread(impre
    #force mask shape to be 2D
    mask = np.ones(maskShape)#create
    grayCode = np.zeros(maskShape)#,nbits))#create grayCode blocks
    gCount = 0 #count of graycode images

    for i in range(start, start + 20,2):
        #setup the first file to be read and read it

        if(i < 10):
            fileNum = "0" + str(i)
        else:
            fileNum = str(i)
        I = plt.imread(imprefix+fileNum+fileEnd)
        if (I.dtype == np.uint8):
            I = I.astype(float) / 256
        if(len(I.shape)>2):
            #print("Image set to gray values!")
            image = (I[:,:,0] + I[:,:,1] + I[:,:,2])/3.0
        else:
            image = I

        #print("File name 1: ", imprefix+fileNum+fileEnd)

        #read second file
        if(i+1 < 10):
            fileNum2 = "0" + str(i+1)
        else:
            fileNum2 = str(i+1)
        I2 = plt.imread(imprefix+fileNum2+fileEnd)
        if (I2.dtype == np.uint8):
            I2 = I2.astype(float) / 256
        if(len(I.shape)>2):
            #print("Image2 set to gray values!")
            image2 = (I2[:,:,0] + I2[:,:,1] + I2[:,:,2])/3.0
        else:
            image2 = I2

        #print("IMage1 shape: ", image.shape)
        #print("image2 shape: ", image2.shape)
        temp = np.where(image > image2, 1, 0)
        #print("File name 2: ", imprefix+fileNum2+fileEnd)


        #update gray
        if(i == start):
            grayCode = temp
        else:
            grayCode = np.dstack((grayCode, temp))

        mask = mask * np.where(abs(image - image2) > threshold, 1,0)
        gCount += 1

    #convert from gray to binary
    #make each value the XOR of the least significant bit with its neighbor
    #converts to binary
    binCode = np.zeros((maskShape[0],maskShape[1],nbits))

    binCode[:,:,0] = grayCode[:,:,0]

    for i in range(0,nbits-1):
        binCode[:,:,i+1] = np.logical_xor(binCode[:,:,i], grayCode[:,:,i+1])


    #convert binary to decimal
    code = np.zeros((maskShape[0],maskShape[1]))
    for i in range(nbits):
        code[:,:] = code[:,:] + (binCode[:,:,(nbits-1)-i]* (2**i))


    return code,mask
```

In [61]:

```
def reconstruct(imprefixL,imprefixR,threshold,camL,camR, backgroundIML,backg
    """
    Performing matching and triangulation of points on the surface using str
    illumination. This function decodes the binary graycode patterns, matche
    pixels with corresponding codes, and triangulates the result.

    The returned arrays include 2D and 3D coordinates of only those pixels w
    were triangulated where pts3[:,i] is the 3D coordinte produced by triang
    pts2L[:,i] and pts2R[:,i]

    Parameters
    ----------
    imprefixL, imprefixR : str
        Image prefixes for the coded images from the left and right camera

    threshold : float
        Threshold to determine if a bit is decodeable

    camL,camR : Camera
        Calibration info for the left and right cameras

    Returns
    -------
    pts2L,pts2R : 2D numpy.array (dtype=float)
        The 2D pixel coordinates of the matched pixels in the left and right
        image stored in arrays of shape 2xN

    pts3 : 2D numpy.array (dtype=float)
        Triangulated 3D coordinates stored in an array of shape 3xN

    """

    # Decode the H and V coordinates for the two views

    LeftH, LHMask = decode(imprefixL,0,threshold)
    LeftV, LVMask = decode(imprefixL,20,threshold)
    RightH, RHMask = decode(imprefixR,0,threshold)
    RightV, RVMask = decode(imprefixR,20,threshold)

    #create masks for the background color and the object color
    backGroundMaskL = plt.imread(backgroundIML)
    backGroundMaskR = plt.imread(backgroundIMR)

    objectMaskL = plt.imread(colorIML)
    objectMaskR = plt.imread(colorIMR)

    color_maskL = (np.linalg.norm(objectMaskL - backGroundMaskL,axis = 2)> c
    color_maskR = (np.linalg.norm(objectMaskR - backGroundMaskR,axis = 2)> c


    # Construct the combined 20 bit code C = H + 1024*V and mask for each vi
    LeftCode = LeftH + (1024 * LeftV)
    RightCode = RightH + (1024*RightV)
    LMask = np.logical_and(LHMask, LVMask)
    RMask = np.logical_and(RHMask, RVMask)

    LMask = np.logical_and(color_maskL, LMask)
    RMask = np.logical_and(color_maskR, RMask)


    LeftCodeFinal = np.where( LMask, LeftCode, np.nan)
    RightCodeFinal = np.where(RMask, RightCode, np.nan)#only keep pixels tha


    # Find the indices of pixels in the left and right code image that
    # have matching codes. If there are multiple matches, just
    # choose one arbitrarily.
    LR,matchL, matchR = np.intersect1d(LeftCodeFinal, RightCodeFinal, return





    # Let CL and CR be the flattened arrays of codes for the left and right
    # Suppose you have computed arrays of indices matchL and matchR so that
    # CL[matchL[i]] == CR[matchR[i]] for all i.  The code below gives one ap
    # to generating the corresponding pixel coordinates for the matched pixe
    h,w =  LeftH.shape
    xx,yy = np.meshgrid(range(w),range(h))
    xx = np.reshape(xx,(-1,1))
    yy = np.reshape(yy,(-1,1))
    pts2R = np.concatenate((xx[matchR].T,yy[matchR].T),axis=0)
    pts2L = np.concatenate((xx[matchL].T,yy[matchL].T),axis=0)



    # Now triangulate the points
    #def triangulate(pts2L,camL,pts2R,camR)
    pts3 = triangulate(pts2L, camL, pts2R, camR)

    #now create color set associated with each point
    colorLeft = objectMaskL[pts2L[1],pts2L[0]].T
    colorRight = objectMaskR[pts2R[1],pts2R[0]].T

    finalColor = .5*(colorLeft+colorRight)


    return pts2L,pts2R,pts3,finalColor
```

In [91]:

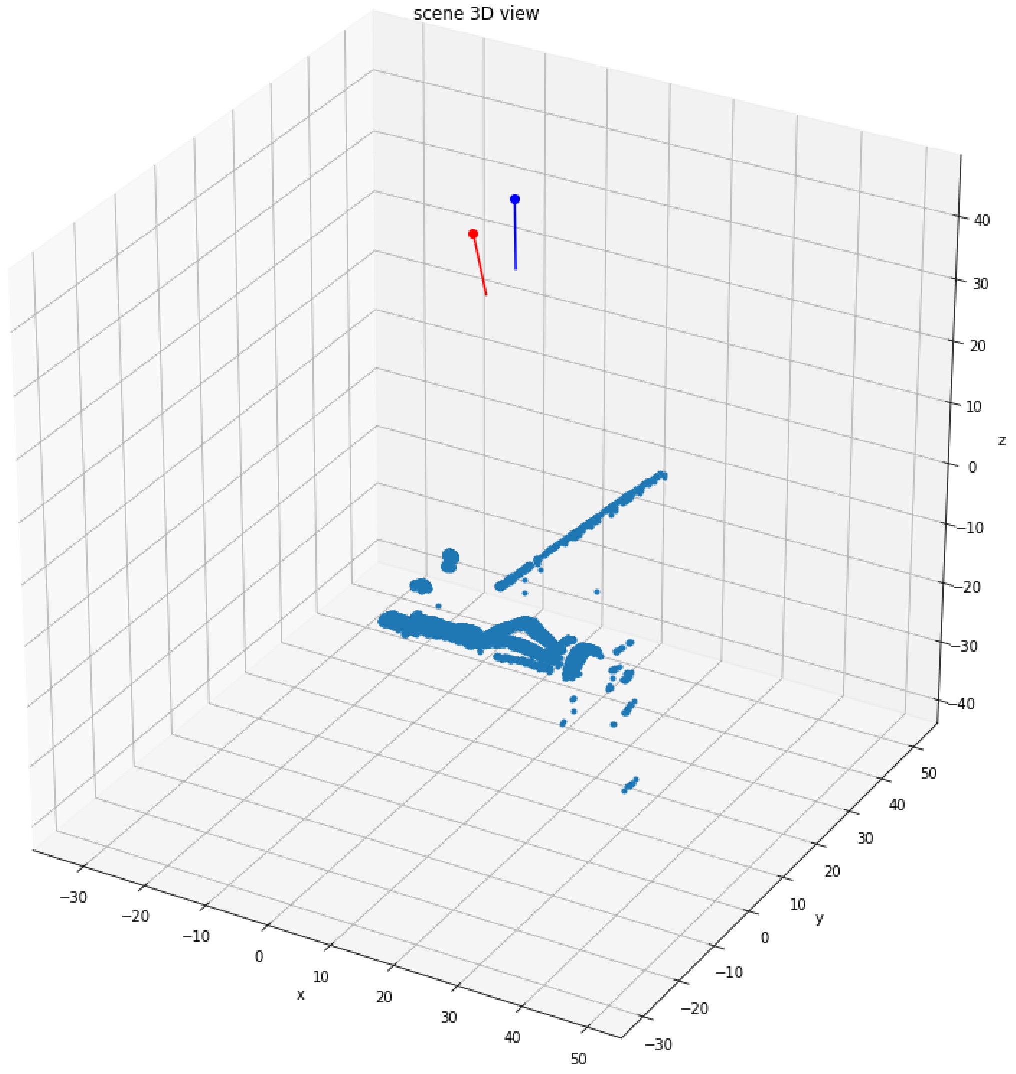
```
#reconstruct(imprefixL,imprefixR,threshold,camL,camR, backgroundIML,backgrou
imprefixC0 = "C:\\grab_0_u\\frame_C0_"
imprefixC1 = "C:\\frame_C1_"
threshold = .05

backL = "C:\\grab_0_u\\color_C0_00.png"
backR = "C:\\grab_0_u\\color_C1_00.png"

objL = "C:\\grab_0_u\\color_C0_01.png"
objR = "C:\\grab_0_u\\color_C1_01.png"

colorThresh = .02

pts2L,pts2R,pts3,color = reconstruct(imprefixC0,imprefixC1,threshold,camL,ca

#print(pts2L)
```

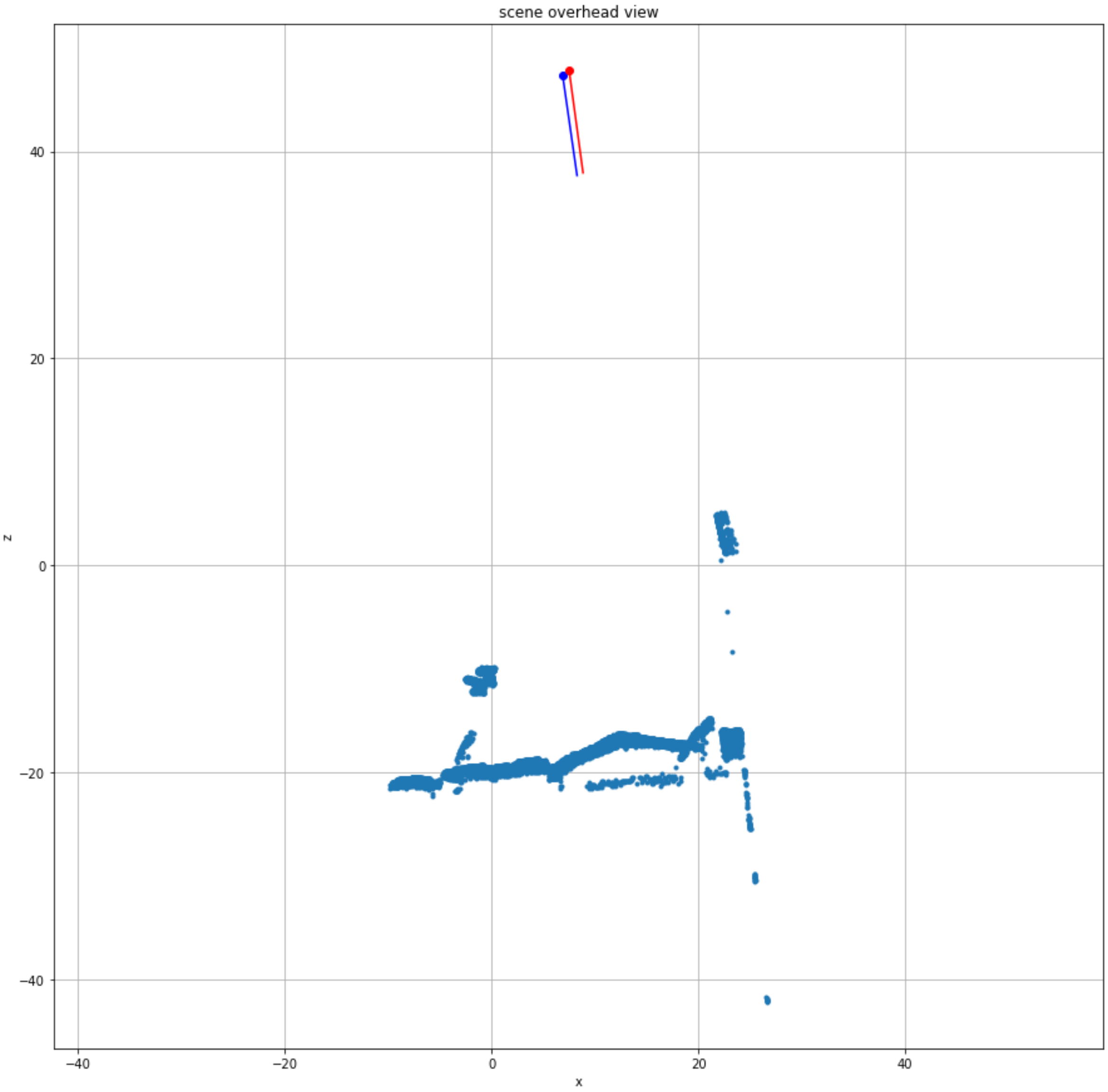
In [41]:

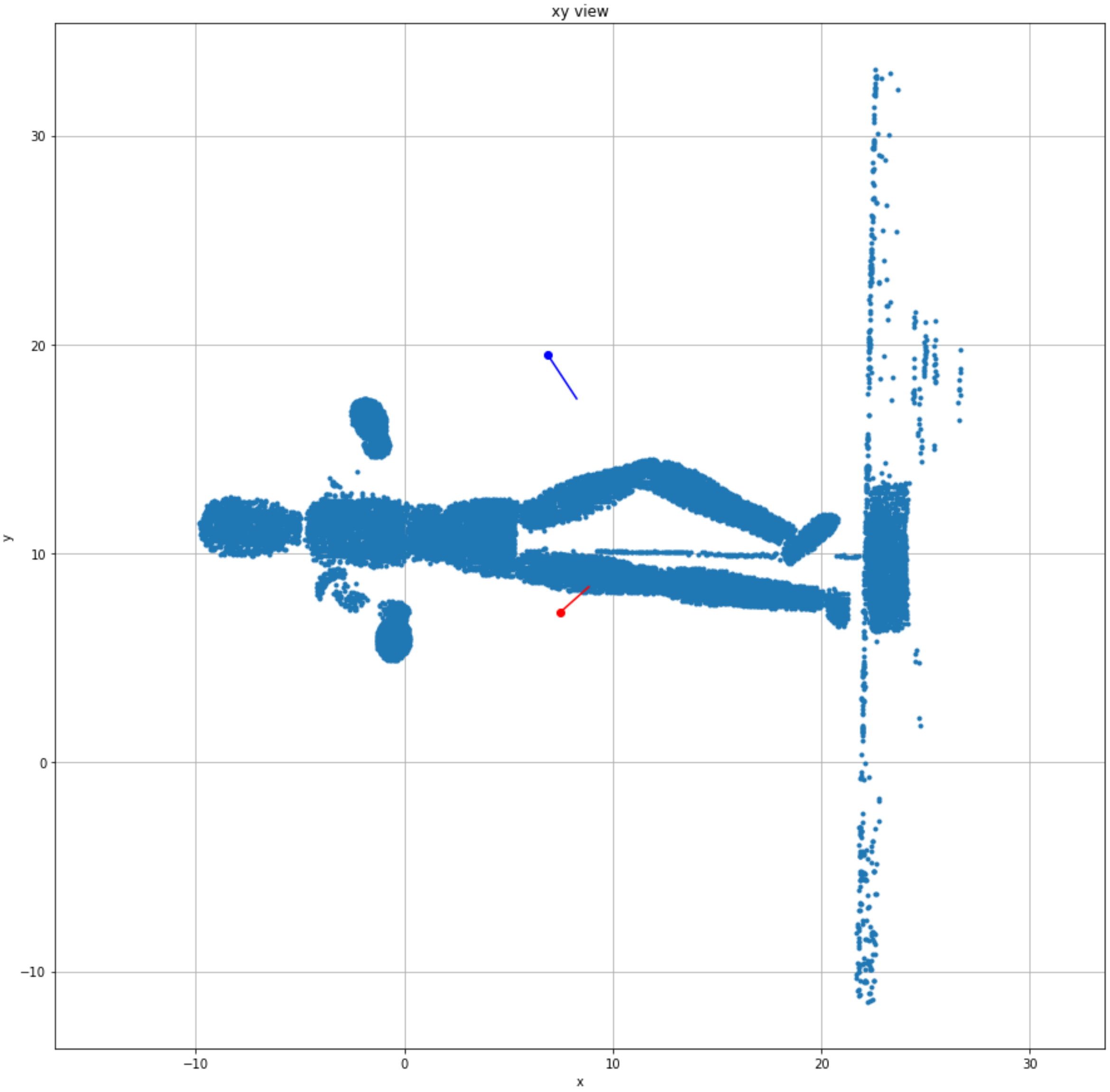
```python
lookL = np.hstack((camL.t,camL.t+camL.R @ np.array([[0,0,10]]).T))
lookR = np.hstack((camR.t,camR.t+camR.R @ np.array([[0,0,10]]).T))
fig = plt.figure()


#visualize 3D layout of points, camera positions
# and the direction the camera is pointing
ax = fig.add_subplot(1,1,1,projection='3d')
ax.plot(pts3[0,:],pts3[1,:],pts3[2,:],'.')
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)
"""ax.set_xlim3d(-200, 500)
ax.set_ylim3d(-500,300)
ax.set_zlim3d(-200,200)"""
plt.title('scene 3D view')

# overhead view showing points, camera
# positions, and direction camera is pointed
fig = plt.figure()
ax = fig.add_subplot(1,1,1)
ax.plot(pts3[0,:],pts3[2,:],'.')
ax.plot(camL.t[0],camL.t[2],'bo')
ax.plot(lookL[0,:],lookL[2,:],'b')
ax.plot(camR.t[0],camR.t[2],'ro')
ax.plot(lookR[0,:],lookR[2,:],'r')
plt.axis('equal')
plt.grid()
plt.xlabel('x')
plt.ylabel('z')

#plt.zlim(-200,200)
plt.title('scene overhead view')


fig = plt.figure()
ax = fig.add_subplot(1,1,1)
ax.plot(pts3[0,:],pts3[1,:],'.')
ax.plot(camL.t[0],camL.t[1],'bo')
ax.plot(lookL[0,:],lookL[1,:],'b')
ax.plot(camR.t[0],camR.t[1],'ro')
ax.plot(lookR[0,:],lookR[1,:],'r')
plt.axis('equal')
plt.grid()
plt.xlabel('x')
plt.ylabel('y')

#plt.zlim(-200,200)
plt.title('xy view')

fig = plt.figure()
ax = fig.add_subplot(1,1,1)
```

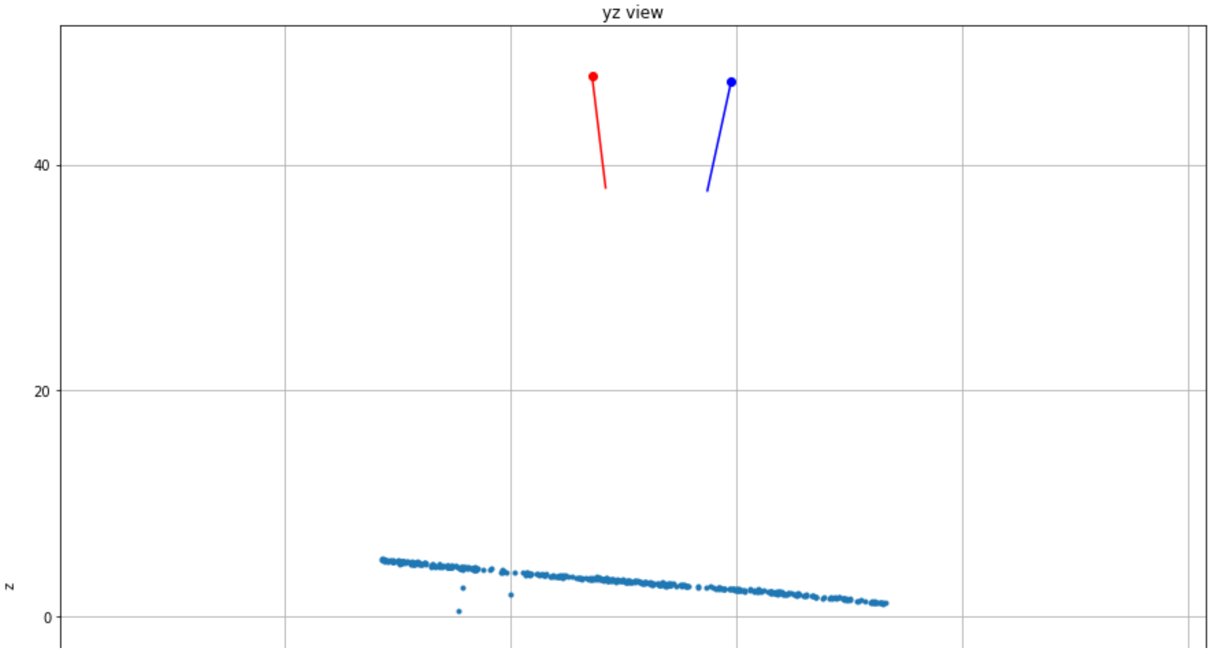
```
ax.plot(pts3[1,:],pts3[2,:],'.')
ax.plot(camL.t[1],camL.t[2],'bo')
ax.plot(lookL[1,:],lookL[2,:],'b')
ax.plot(camR.t[1],camR.t[2],'ro')
ax.plot(lookR[1,:],lookR[2,:],'r')
plt.axis('equal')
plt.grid()
plt.xlabel('y')
plt.ylabel('z')

#plt.zlim(-200,200)
plt.title('yz view')

```

```
Out[41]: Text(0.5, 1.0, 'yz view')
```




scene overhead view
40
20
0
−20
−40
z
−40
−20
0
20
40
x


xy view
y
x
30
20
10
0
−10
−10
0
10
20
30



Part 3)

Part 4 smoothing function was attempted as part of meshGen function

In [89]:

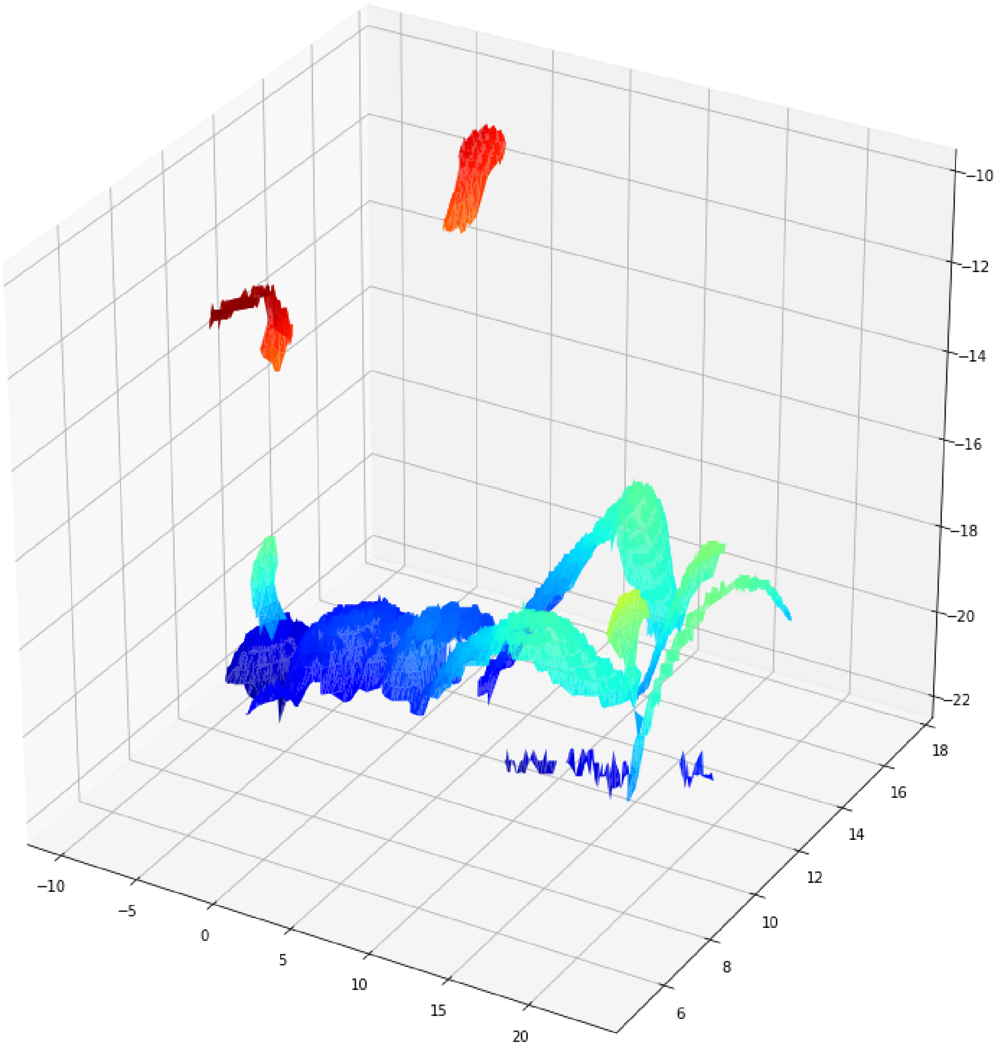
```
def meshGen(imprefixC0,imprefixC1,threshold,camL,camR, backL,backR, objL,obj
    pts2L,pts2R,pts3,color = reconstruct(imprefixC0,imprefixC1,threshold,cam
    boxlimits = np.array([-25,23,-25,25,-25,-5])
    #xl,xh,yl,yh,zl,zh
    #l = lower, h = higher bounds

    # Specify a longest allowed edge that can appear in the mesh. Remove tri
    # from the final mesh that have edges longer than this value
    trithresh = 1




    #
    # bounding box pruning
    #

    prunedPoints3D = np.zeros(pts3.shape)
    prunedPoints3D = np.where((pts3[0,:] > boxlimits[0]) *  (pts3[0,:] < box
                            (pts3[1,:] < boxlimits[3]) * (pts3[2,:] > boxlim

    prunedPoints2L = np.zeros(pts2L.shape)
    prunedPoints2L = np.where((pts3[0,:] > boxlimits[0]) *  (pts3[0,:] < box
                            (pts3[1,:] < boxlimits[3]) * (pts3[2,:] > boxlim

    prunedPoints2R = np.zeros(pts2R.shape)
    prunedPoints2R = np.where((pts3[0,:] > boxlimits[0]) *  (pts3[0,:] < box
                            (pts3[1,:] < boxlimits[3]) * (pts3[2,:] > boxlim




    #
    # triangulate the 2D points to get the surface mesh
    #


    prunedPoints2L = prunedPoints2L[:,~np.all(np.isnan(prunedPoints2L), axis
    prunedPoints3D = prunedPoints3D[:,~np.all(np.isnan(prunedPoints3D), axis
    prunedPoints2R = prunedPoints2L[:,~np.all(np.isnan(prunedPoints2L), axis

    tri2LPrunedPts = Delaunay(prunedPoints2L.T)




    #triangle pruning given by Professor Charless Fowlkes

    tri = tri2LPrunedPts.simplices.copy()

    d01 = np.sqrt(np.sum(np.power(pts3[:,tri[:,0]]-pts3[:,tri[:,1]],2),axis=
    d02 = np.sqrt(np.sum(np.power(pts3[:,tri[:,0]]-pts3[:,tri[:,2]],2),axis=
    d12 = np.sqrt(np.sum(np.power(pts3[:,tri[:,1]]-pts3[:,tri[:,2]],2),axis=

    goodtri = (d01<trithresh)&(d02<trithresh)&(d12<trithresh)
    tri = tri[goodtri,:]
    #end of code by Professor Charless Fowlkes










    return(prunedPoints3D,tri,color)
```

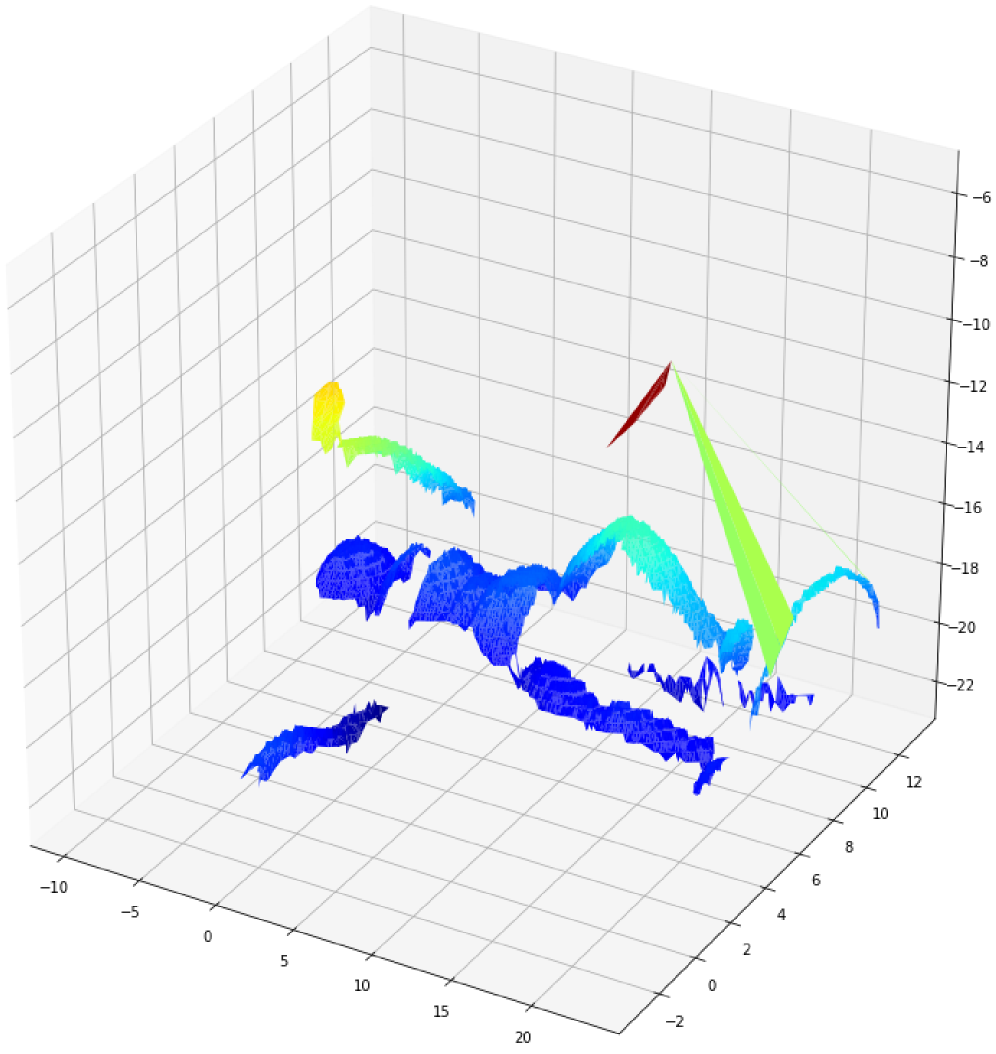
In [64]:

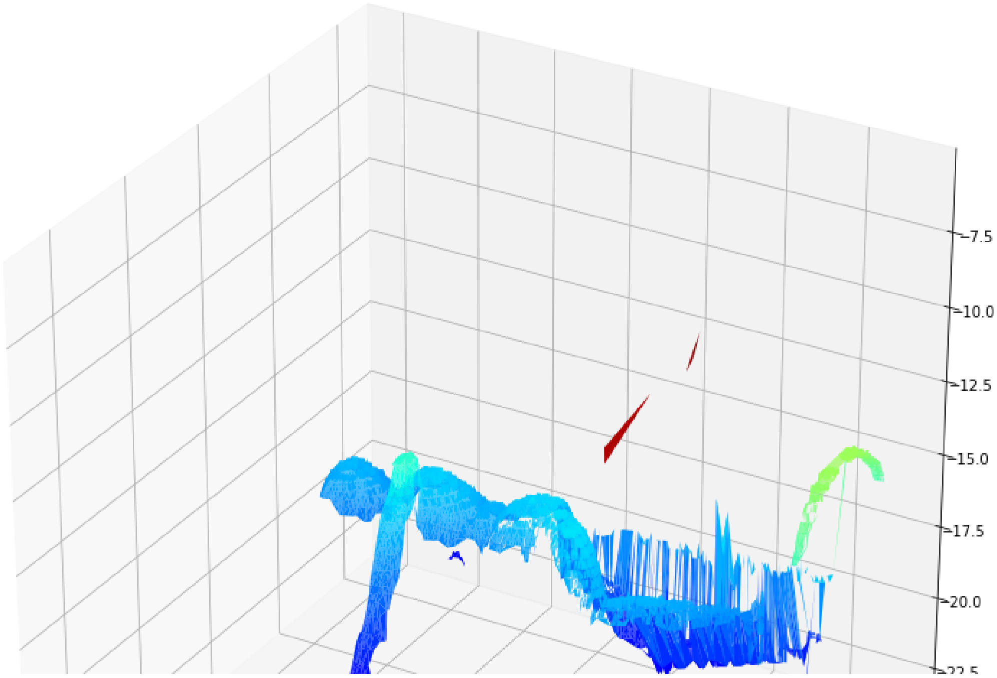
```
pts = []
tris = []
colors = []

for i in range(5):
    imprefixC0 = "C:\\grab_" +str(i) + "_u\\frame_C0_"
    imprefixC1 = "C:\\grab_" +str(i) +"_u\\frame_C1_"

    backL = "C:\\grab_" +str(i) +"_u\\color_C0_00.png"
    backR = "C:\\grab_" +str(i) +"_u\\color_C1_00.png"

    objL = "C:\\grab_" +str(i) +"_u\\color_C0_01.png"
    objR = "C:\\grab_" +str(i) +"_u\\color_C1_01.png"

    colorThresh = .02
#prunedPoints3D, tri
    ptsTemp, trisTemp, colorTemp = meshGen(imprefixC0,imprefixC1,threshold,c
    pts.append(ptsTemp)
    tris.append(trisTemp)
    colors.append(colorTemp)

```

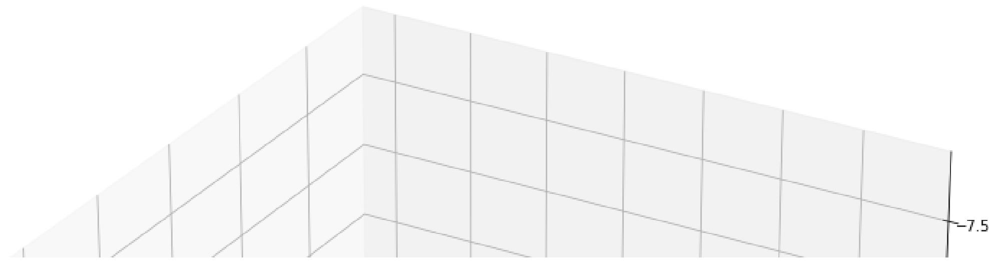
In [65]:

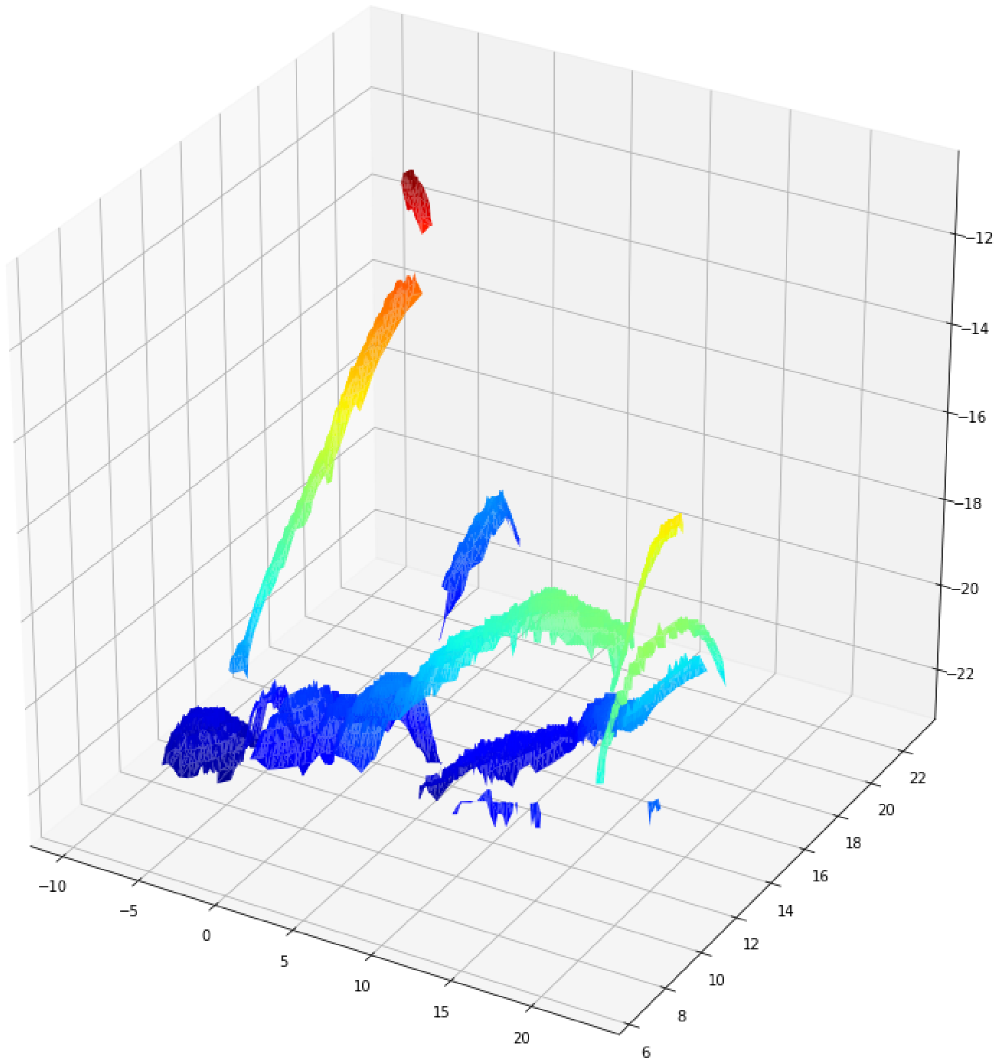
```
for i in range(5):
    fig = plt.figure()
    ax = fig.add_subplot(1,1,1,projection='3d')

    ax.plot_trisurf(pts[i][0], pts[i][1], pts[i][2], triangles=tris[i], cmap

```




−10
−5
0
5
10
15
20
−2
0
2
4
6
8
10
12
−6
−8
−10
−12
−14
−16
−18
−20
−22


−7.5
−10.0
−12.5
−15.0
−17.5
−20.0
−22.5


−7.5


−12
−14
−16
−18
−20
−22
−10
−5
0
5
10
15
20
6
8
10
12
14
16
18
20
22

In [66]:

```python
for i in range(5):
    #def writeply(X,color,tri,filename):
    writeply(pts[i],colors[i],tris[i], "meshColoredAgain"+str(i)+".ply")
```