Assignment 4 : Structured Illumination and Meshing

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

name: Tzu Hsuan Huang

SID: 67913387

In [1]:
```
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

%matplotlib inline
```

In [2]:
```
plt.rcParams['figure.figsize']=[15,15] #adjust this as needed to get clearly visible figures
```

---

1. Decoding

Write a function called **decode** that reads in a set of images captured with the camera showing the projected gray code and returns an array which is the same size as the camera image where each element contains the decoded value (0..1023) as well as a binary image (mask) indicating which pixels could be reliably decoded. We will end up calling this function 4 times, once for the horizontal and once for the vertical coordinates in both the left and right cameras.

1.1 Implement [25pts]

Use a for loop to generate the list of image file names to load in. Assume that there are 20 images for the 10 bit gray code. The images come in pairs where the second is the inverse of the first. For each pair of images, recover the bit by checking to see that the first image is greater or less than the second. You should also maintain a seperate binary array (mask) the same size as the images in which you mark "undecodable" pixels for which the absolute difference between the first and second image in the pair is smaller than some user defined threshold. This will allow us to ignore pixels for which the decoding is likely to fail (e.g., pixels that weren't illuminated by the projector). I used a threshold of 0.02 when reconstructing the scan below. You will want to mark a pixel as bad if any of the 10 bits was undecodeable.

After thresholding the pairs you should have 10 binary images. You can convert this stack of binary images into a single "decimal image" by decoding each bit pattern to its corresponding decimal representation. As discussed in class, the bits are coded using a graycode rather than a standard binary code so you will need to do a little work to decode them correctly.

I recommend first converting the 10 bit code from the gray code to standard binary (binary coded decimal) using the algorithm we described in class that successively XORs the bits. Once you have converted to BCD, you can then produce the final decimal value using the standard binary-to-decimal conversion (i.e., assuming B[0] is the most significant bit then $\sum_{n=0}^{9} B[9-n] * 2^n$).

In NumPy you can implement both steps efficiently with a for-loop over the 10 bits and vectorized operations over the spatial locations. Make sure you are processing the bits in the correct order (i.e. from least-significant to most-significant).

In [3]:
```
def decode(imprefix,start,threshold):
    """
    Given a sequence of 20 images of a scene showing projected 10 bit gray code,
    decode the binary sequence into a decimal value in (0,1023) for each pixel.
    Mark those pixels whose code is likely to be incorrect based on the user
    provided threshold.  Images are assumed to be named "imageprefixN.png" where
    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 threshold

    """

    # we will assume a 10 bit code
    nbits = 10
    mask = []
    bits= []

    # Load images and create masks
    # don't forget to convert images to grayscale / float after loading them in
    for i in range(start, start + nbits * 2, 2):
        img1 = plt.imread(f"{imprefix}{i:02d}.png")
        if img1.dtype == np.uint8:
            img1 = img1.astype(float) / 256
        if len(img1.shape)==3:
            img1 = img1.mean(axis=-1)
        #print(img1.shape)
        img2 = plt.imread(f"{imprefix}{i+1:02d}.png")
        if img2.dtype == np.uint8:
            img2 = img2.astype(float) / 256
        if len(img2.shape)==3:
            img2 = img2.mean(axis=-1)
        #mark "undecodable" pixels for which the absolute difference between the first and se
        #in the pair is smaller than some user defined threshold
        diff = 1*(abs(img1 - img2))#true =1 false =0
        undecodable = 1*(diff > threshold)
        if len(mask) == 0:
            mask=np.ones(img1.shape)
        mask=mask*undecodable
        #For each pair of images, recover the bit by checking to see that the first image is
        bits.append(1*(img1>img2))


    # decoding the gray code
    code = np.zeros_like(img1, dtype=float)
    binary_code=[]
    binary_code.append(bits[0])

    # Convert gray code to binary
    for i in range(nbits-1):
        binary_code.append(np.logical_xor(binary_code[i], bits[i + 1]))

    # Convert binary code to decimal
    for i in range(nbits):
         code+=binary_code[9-i]*(2**i)


    return code,mask
```

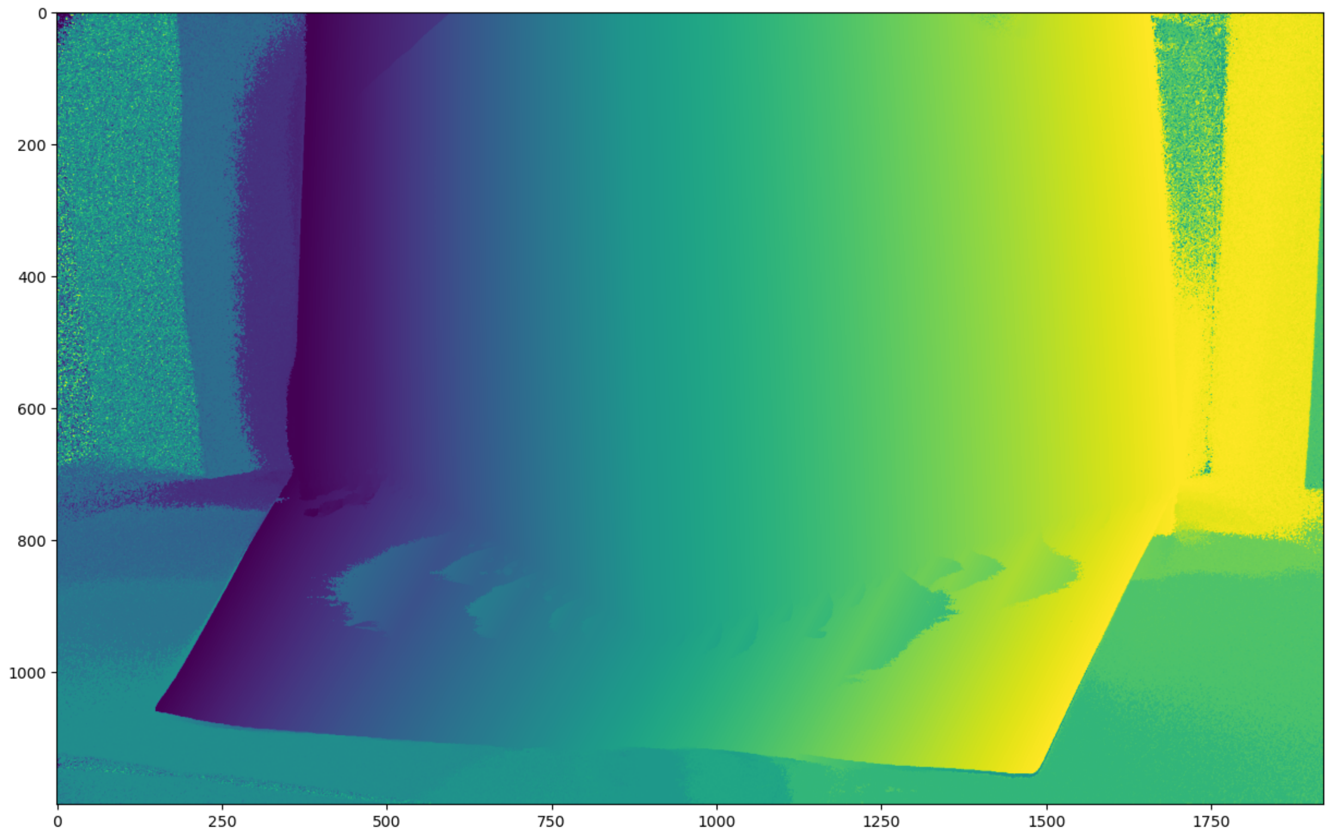
1.2 Test and Visualize [5pts]

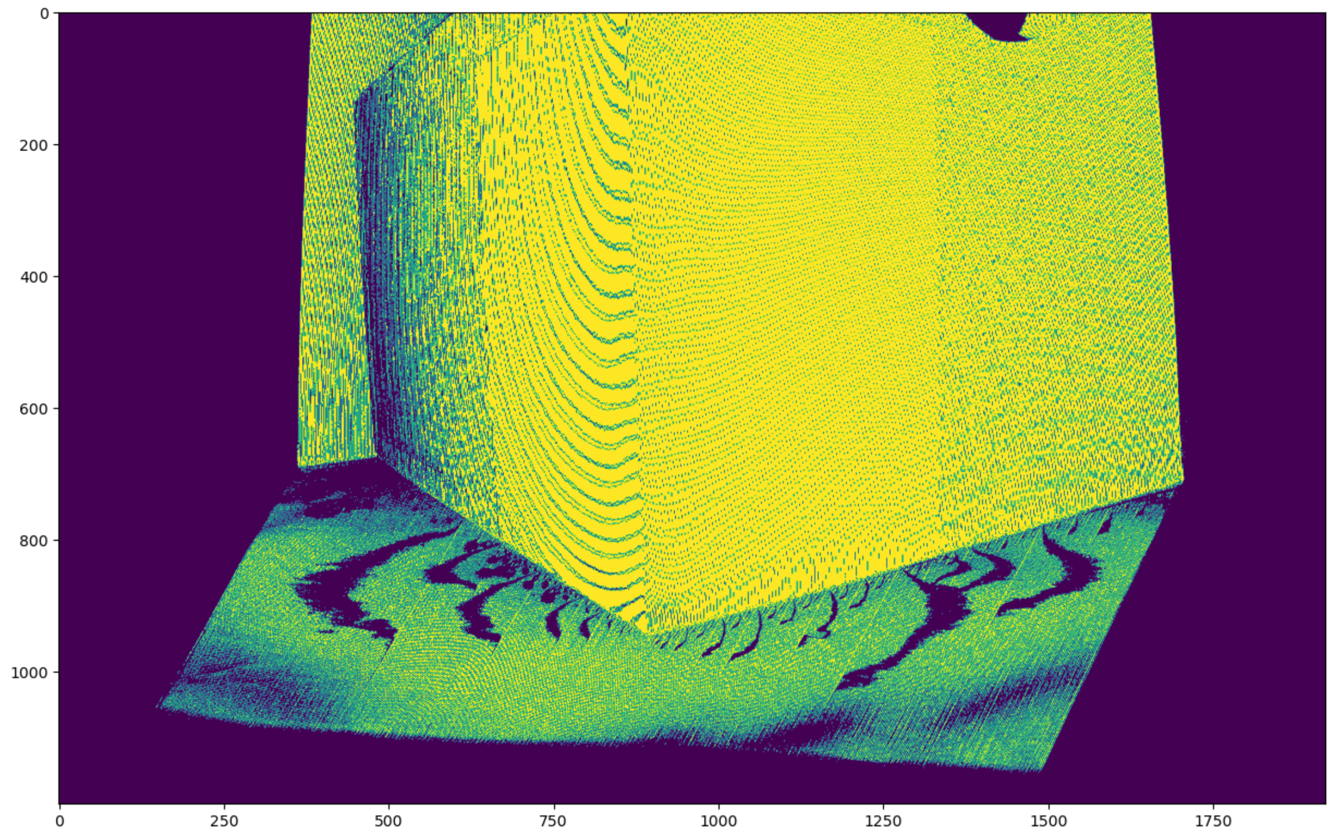
The graycode images which are projected are included in the data sub-directory **gray/**. This is useful for debugging your decode function prior to running it on the real scan data since the results should be perfect (no noise!).

1. Check that your decode function correct decodes the projector data
2. Visualize the results of running your decode function on one of the provided scans

In [4]:

```
#
# first generate arrays containing the "true" codes projected by
# the projector
#
# the projector had a resolution of 1280x800 so there are black bars
# of 128 pixels on either side of our 1024 pixel region

pad = np.zeros((800,128))
hcode,vcode = np.meshgrid(range(1024),range(800))
Htrue = np.concatenate((pad,hcode,pad),axis=1)
Vtrue = np.concatenate((pad,vcode,pad),axis=1)
masktrue = np.concatenate((pad,np.ones((800,1024)),pad),axis=1)


#
# run your decoding on the projector images
#
thresh = 0.0000001  #this data is perfect so we can use a very small threshold
H,Hmask = decode("C:/Users/allye/Desktop/UCI/cs117/Assignment 4/gray/" ,0,thresh)
V,Vmask = decode("C:/Users/allye/Desktop/UCI/cs117/Assignment 4/gray/",20,thresh)
# compare to the known "true" code
assert((H==Htrue).all())
assert((V==Vtrue).all())
assert((Hmask==masktrue).all())
assert((Vmask==masktrue).all())



#
# Visualize results for the first scan and experiment to determine a good threshold
#
thresh = 0.02
code,mask = decode("C:/Users/allye/Desktop/UCI/cs117/Assignment 4/scan0/frame_C0_",0,thresh)

# display view code and mask as images
plt.rcParams['figure.figsize'] = [15,15]
plt.imshow(code), plt.show();
plt.imshow(mask), plt.show()
```





Out[4]:
```
(<matplotlib.image.AxesImage at 0x1e49d8f9c50>, None)
```

---

2. Reconstruction

We can now use the decoded pixel ids in order to easily find correspondences between two different camera views and triangulate the resulting points. You can use your own implementation from assignment 2 or import the provided **camutils.py** module as needed to do the triangulation.

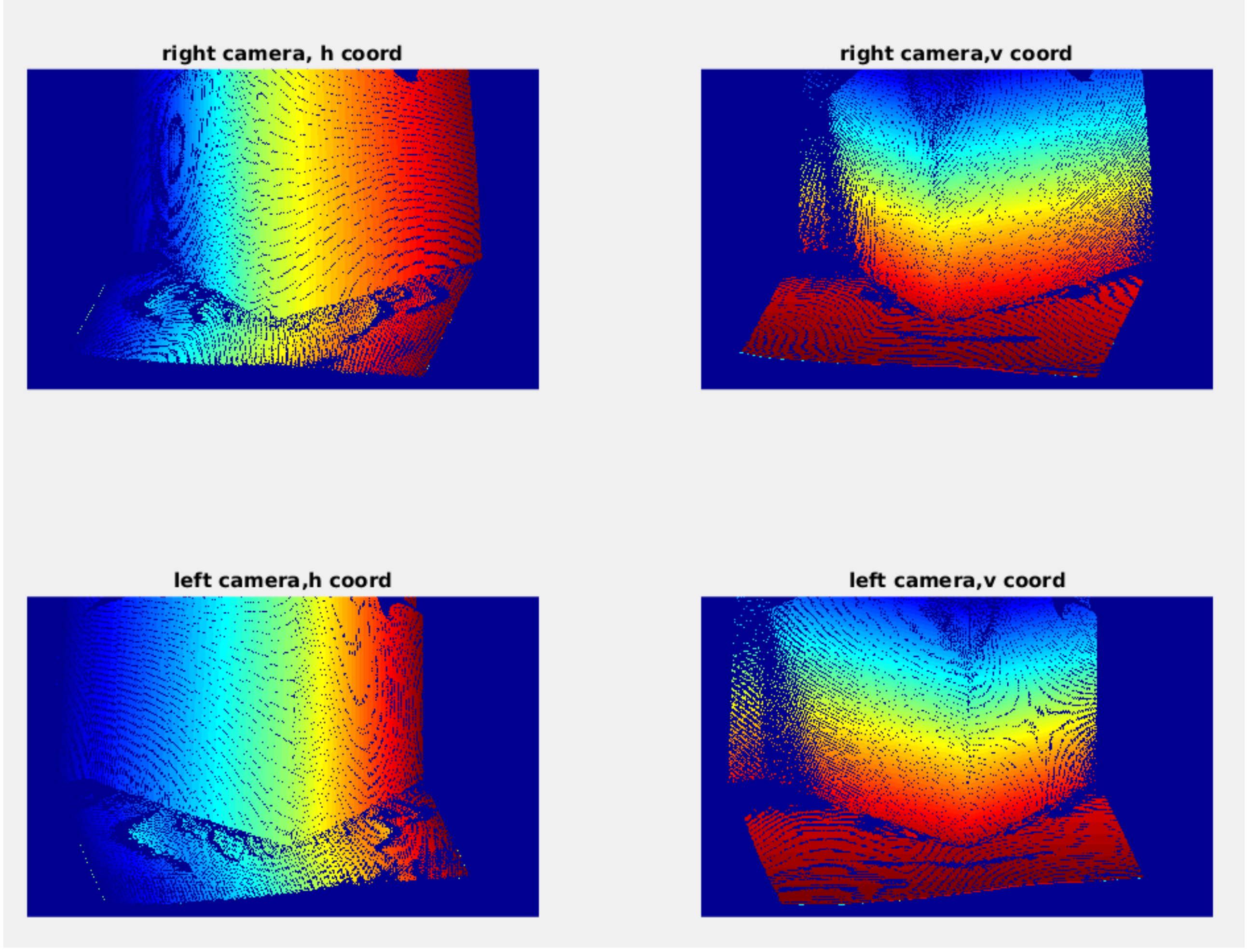
2.1 Implementation [25pts]

Write a function called **reconstruct** that takes a set of graycoded images from the pair of cameras and produces a 3D point cloud. To achieve this:

1. Call your decode function four times to decode the horizontal and vertical images for both the left and right cameras. If you visualize the resulting codes, they should look like the figure below (after masking). You should combine the horizontal and vertical codes to get a single (20-bit) integer for the left and right cameras. You should also combine the corresponding binary masks so only pixels with both good horizontal and vertical codes are marked as valid.

2. For each pixel in the left image which was succesfully decoded, find the pixel in the right image with the corresponding code. One way to do this effeciently is using the **numpy.intersect1d** function with *return_indices=True* in order to get indices of matching pixels in the two images.

3. Now that you have corrsponding pixel coordinates in the two images and the camera parameters, use **triangulate** to get the 3D coordinates for this set of pixels.



In [5]:

```python
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 = new_pts2L.astype(np.float64)
    new_pts2L /=  camL.f
    oneArr = np.ones(N)
    qL = np.vstack((new_pts2L, oneArr))
    new_pts2R = pts2R - camR.c
    new_pts2R = new_pts2R.astype(np.float64)
    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
```

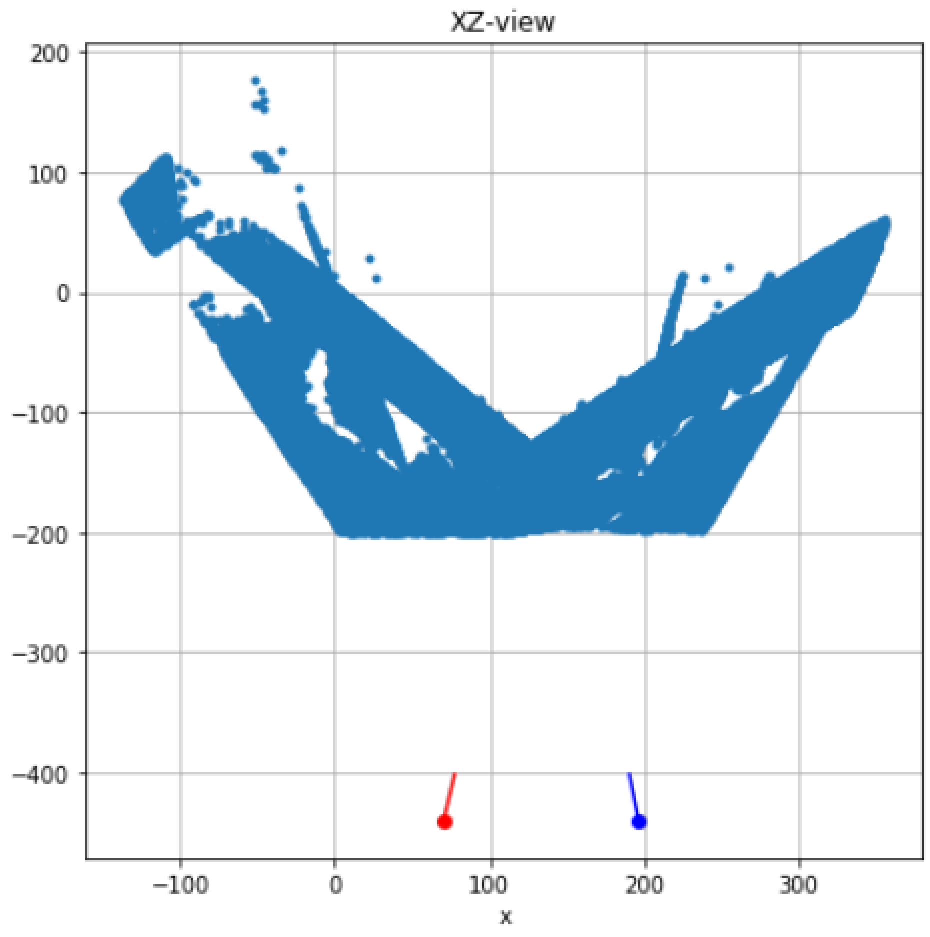
In [6]:

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

    The returned arrays include 2D and 3D coordinates of only those pixels which
    were triangulated where pts3[:,i] is the 3D coordinte produced by triangulating
    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
    Hl,Hlmask = decode(imprefixL,0,threshold)
    Vl,Vlmask = decode(imprefixL,20,threshold)
    Hr,Hrmask = decode(imprefixR,0,threshold)
    Vr,Vrmask = decode(imprefixR,20,threshold)

    # Construct the combined 20 bit code C = H + 1024*V and mask for each view
    CL = (Hl + 1024*Vl)*Hlmask*Vlmask
    CR = (Hr + 1024*Vr)*Hrmask*Vrmask

    # 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.
    matchL=np.intersect1d(CL, CR, return_indices=True)[1]
    matchR=np.intersect1d(CR, CL, return_indices=True)[1]
    # Let CL and CR be the flattened arrays of codes for the left and right view
    # 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 approach
    # to generating the corresponding pixel coordinates for the matched pixels.
    w= Hl.shape[1]
    h= Hl.shape[0]

    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
    pts3=triangulate(pts2L,camL,pts2R,camR)


    return pts2L,pts2R,pts3
```

2.2 Visualization [5pts]

I've have provided the calibration for the two cameras in the scan0/scan0_calibration.pickle so that you can easily load them in to your code. Call your **reconstruct** function on the provided image data and visualize the reconstructed 3D points to make sure everything is working. You may want to experiment with adjusting the decoding threshold to get a good number of points without too much noise.

Please note that at this stage the results are quite noisy so if you plot them, you will probably need to zoom in to see the main object. You can adjust your plot axes to set the extent of the plot axis after displaying the points. For this scan, axis limits of [-200 400],[-200 300], and [-200 200] in the x,y and z directions repectively should work well.

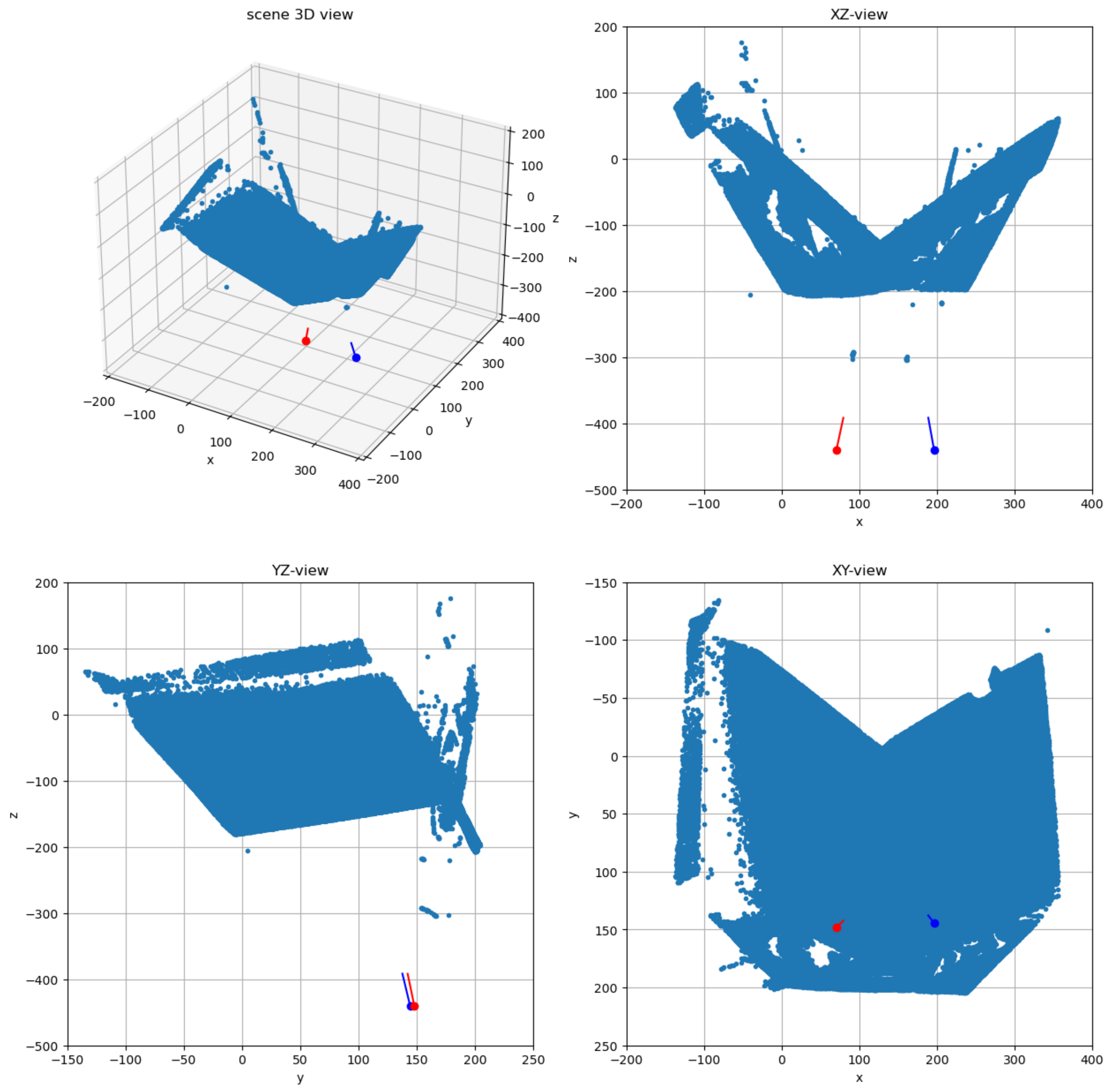


In [7]:

```python
#
# Reconstruct and visualize the results
#
imprefixC0 = "C:/Users/allye/Desktop/UCI/cs117/Assignment 4/scan0/frame_C0_"
imprefixC1 = "C:/Users/allye/Desktop/UCI/cs117/Assignment 4/scan0/frame_C1_"
threshold = 0.02

fid = open("C:/Users/allye/Desktop/UCI/cs117/Assignment 4/scan0/scan0_calibration.pickle",'rb
(camC0,camC1) = pickle.load(fid)
fid.close
camL = camC0
camR = camC1
print(camL)
print(camR)

pts2L,pts2R,pts3 = reconstruct(imprefixC0,imprefixC1,threshold,camC0,camC1)

# Add your visualization code here.  As we have done previously it is good to visualize diffe
# 2D projections XY, XZ, YZ and well as a 3D version

# generate coordinates of a line segment running from the center
# of the camera to 2 units in front of the camera

lookL = np.hstack((camL.t,camL.t+camL.R @ np.array([[0,0,50]]).T))
lookR = np.hstack((camR.t,camR.t+camR.R @ np.array([[0,0,50]]).T))

#visualize 3D layout of points, camera positions
# and the direction the camera is pointing
fig = plt.figure(figsize=(15,15))
ax = fig.add_subplot(2,2,1,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')
ax.set_xlim3d([-200,400])
ax.set_ylim3d([-200,400])
ax.set_zlim3d([-400,200])
visutils.label_axes(ax)
plt.title('scene 3D view')

ax = fig.add_subplot(2,2,2)
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([-200, 400, -500, 200])
plt.grid()
plt.xlabel('x')
plt.ylabel('z')
plt.title('XZ-view')

ax = fig.add_subplot(2,2,3)
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([-150, 250, -500, 200])
plt.grid()
plt.xlabel('y')
plt.ylabel('z')
plt.title('YZ-view')

ax = fig.add_subplot(2,2,4)
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([ -200, 400 ,250,-150 ])
plt.grid()
plt.xlabel('x')
plt.ylabel('y')
plt.title('XY-view')
```

```
Camera :
 f=1412
 c=[[995 582]]
 R=[[ 0.9885 -0.0185 -0.1504]
 [-0.0018  0.991  -0.1336]
 [ 0.1515  0.1324  0.9796]]
 t = [[ 196.4  144.5 -440.6]]
Camera :
 f=1410
 c=[[986 608]]
 R=[[ 0.9834 -0.024   0.1797]
 [ 0.0454  0.9922 -0.1157]
 [-0.1756  0.122   0.9769]]
 t = [[  70.5  148.1 -440.6]]
```

```
Out[7]: Text(0.5, 1.0, 'XY-view')
```



***

3. Meshing

To display the reconstructed scan as a surface, we need to generate triangular faces of the mesh which connect up the points. Write code that takes the output of **reconstruct** and produces a triangulated mesh. Your mesh should be represented by an array of points (vertices) and an array of triplets of point indices (triangular faces).

3.1 Implementation [35pts]

reconstruct provides us with the set of verticies. To find the faces of your mesh, we will use the 2D coordinates of the points as they were visible in the left image (**pts2L**). Use the function **scipy.spatial.Delaunay** on these 2D coordinates to get the list of triangles. These faces along with the 3D coordinates of the points provide a description of the surface in 3D. You can display the resulting surface mesh in using **plot_trisurf** passing it the coordinates in **pts3** and the triangles in **tri** returned by Delaunay that contains the list of triangles.

You will also need to implement some code in your mesh function for pruning out noisy points. Please implement the following two ideas:

(a) Bounding Box Pruning: The scanned object lies in a bounding box roughly 400mm on a side. Prune the set of triangulated points to only keep those that are inside this 3D volume. You will need to experiment to find the best bounding box dimensions along the x,y and z coordinates in order to enclose the surface but exclude as much of the noise points as possible. Once you have determined a good range, you should drop points in *pts3* that are outside that range. You should also drop the corresponding 2D points stored in *pts2L* and *pts2R*. Once you have dropped the points outside the bounding box, you should (re)run **Delaunay** to get a triangulation which only includes the good points.

(b) Triangle Pruning: Remove triangles from the surface mesh that include edges that are longer than a user-defined threshold. Typically when we have points on the surface, they will be relatively close by so this gives a way to get rid of noisy points off the surface. After removing mesh triangles with edges that are too long, you will find points that are no-longer connected to any neighbor in the mesh (e.g. they are no longer in any mesh triangle). Find these points and remove them as well so that the final mesh you produce doesn't include any unreferenced vertices.

Your script should include a user specified parameter **trithresh** which specifies the longest allowed edge that can appear in the mesh. You should remove any triangles for which the longest edge of the triangle has a length greater than **trithresh**. The units of the reconstruction are in mm so the threshold will be in those same units. You will need to experiment to determine a good setting.

In [10]:
```
# Mesh cleanup parameters
from scipy.spatial import Delaunay

#fig = plt.figure(figsize=(15,10))
#ax = fig.add_subplot(2,2,1,projection='3d')
#get the list of triangles
#Delaunay=> the input array should have a shape of (N, 2).
#trias=Delaunay(pts2L.T)
#ax.plot_trisurf(pts3[0,:], pts3[1,:], pts3[2,:], triangles=trias.simplices)
#ax.set_xlim3d([-200,400])
#ax.set_ylim3d([-200,400])
#ax.set_zlim3d([-400,200])
#visutils.label_axes(ax)

# Specify limits along the x,y and z axis of a box containing the object
# we will prune out triangulated points outside these limits
boxlimits = np.array([-60,350,-100,140,-180,70])

# Specify a longest allowed edge that can appear in the mesh. Remove triangles
# from the final mesh that have edges longer than this value
trithresh = 0.9



#
# bounding box pruning
delete_index = []
#drop points in pts3 that are outside that range
for i in range(pts3.shape[1]):
    #smaller or larger than x, y, z limitation
    if(pts3[0][i]< boxlimits[0] or pts3[0][i]>boxlimits[1] #x
    or pts3[1][i]< boxlimits[2] or pts3[1][i]>boxlimits[3] #y
    or pts3[2][i]< boxlimits[4] or pts3[2][i]>boxlimits[5]): #z
        delete_index.append(i)
# numpy.delete function, the obj parameter refers to the index or indices of the elements you
pts3 = np.delete(pts3, delete_index, axis = 1)
pts2L = np.delete(pts2L, delete_index, axis = 1)
pts2R = np.delete(pts2R, delete_index, axis = 1)
#
# triangulate the 2D points to get the surface mesh
#
goodTrias = Delaunay(pts2L.T)
#goodTrias shape is (n,3)
#
# triangle pruning

removedTrias = []
# define a function that find the distance between two points
def distance(p1, p2):#p1(1,3)
    return np.linalg.norm(p2-p1)
#Remove triangles from the surface mesh that include edges that are longer than a user-define
for i in range(0, goodTrias.simplices.shape[0]):
    if(distance(pts3.T[goodTrias.simplices[i][0]],pts3.T[goodTrias.simplices[i][1]])>trithres
    or distance(pts3.T[goodTrias.simplices[i][1]],pts3.T[goodTrias.simplices[i][2]])>trithres
    or distance(pts3.T[goodTrias.simplices[i][0]],pts3.T[goodTrias.simplices[i][2]])>trithres
        removedTrias.append(i)
goodTrias.simplices = np.delete(goodTrias.simplices, removedTrias, axis = 0)

#
# remove any points which are not refenced in any triangle
#
# Extract unique indices of points involved in the remaining triangles
points_to_keep = np.unique(goodTrias.simplices)
tri = points_to_keep
# Remove unreferenced vertices from pts3, pts2L, and pts2R
pts3 = pts3[:, points_to_keep]
pts2L = pts2L[:, points_to_keep]
pts2R = pts2R[:, points_to_keep]
```

3.2 Results and Discussion [5pts]

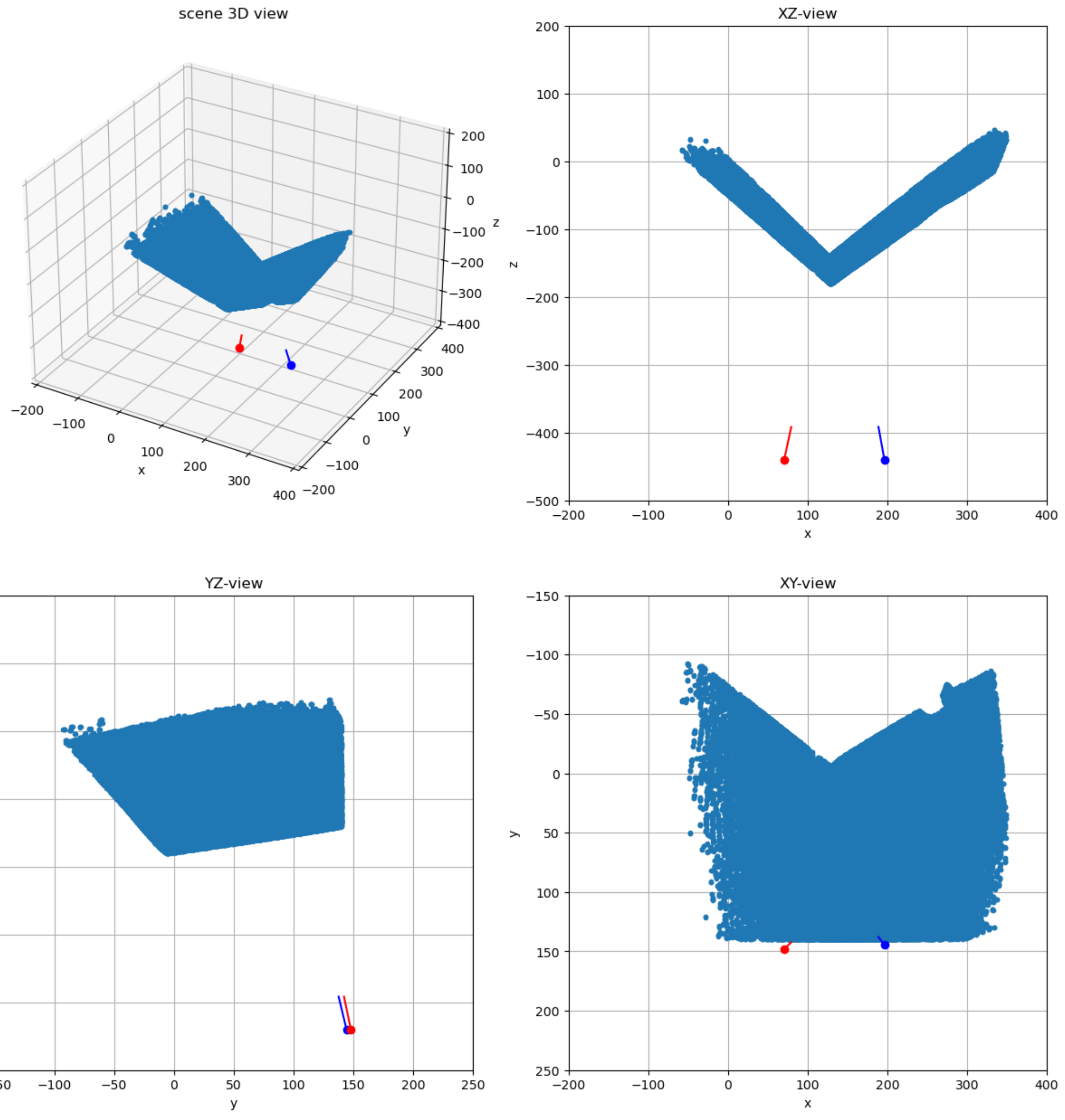
1. Generate visualizations below showing (a) a plot of the triangulated points generated by your reconstruct code, and (b) visualizations the final mesh produced by your code from *two* different viewpoints.

2. Write a paragraph describing the techniques you used in to clean up noisy points. Please explain what your code does. Are there places in this example where your cleanup proceedure fails? What other ideas might work to provide a nicer result with the least amount of user intervention? How might we improve the reconstruction matching process to eliminate noise?

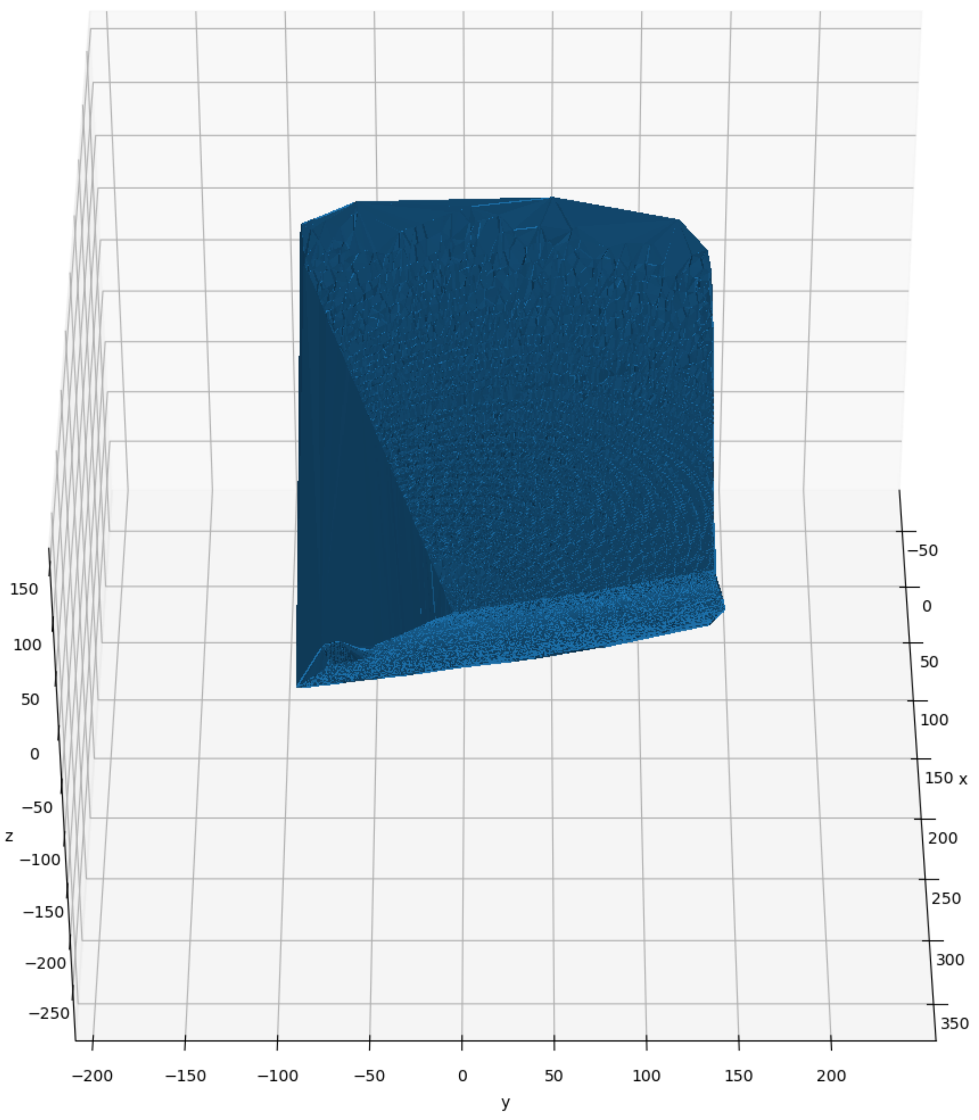
3. Discuss the quality of the scan data. What artifacts do you see? What would be the best placement of the cameras and projector to get the best data possible for both faces of the box?

In [9]:
```
# vis code goes here
lookL = np.hstack((camL.t,camL.t+camL.R @ np.array([[0,0,50]]).T))
lookR = np.hstack((camR.t,camR.t+camR.R @ np.array([[0,0,50]]).T))

#visualize 3D layout of points, camera positions
# and the direction the camera is pointing
```

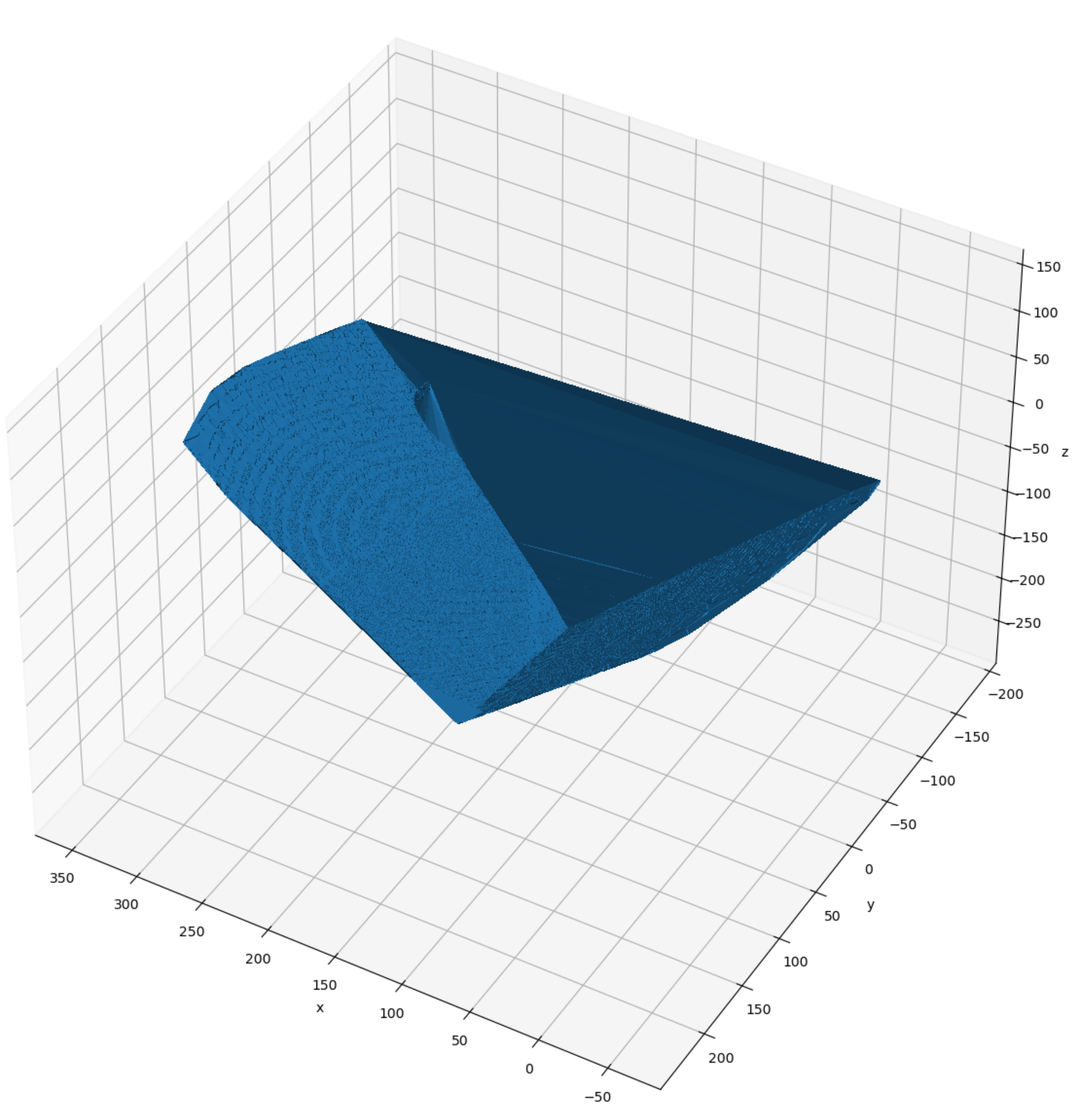
```
fig = plt.figure(figsize=(15,15))
ax = fig.add_subplot(2,2,1,projection='3d')
ax.plot(pts3[0,:],pts3[1,:],pts3[2,:],'.')
ax.plot(camR.t[0],camR.t[1],camR.t[2],'ro')
ax.plot(camL.t[0],camL.t[1],camL.t[2],'bo')
ax.plot(lookL[0,:],lookL[1,:],lookL[2,:],'b')
ax.plot(lookR[0,:],lookR[1,:],lookR[2,:],'r')
ax.set_xlim3d([-200,400])
ax.set_ylim3d([-200,400])
ax.set_zlim3d([-400,200])
visutils.label_axes(ax)
plt.title('scene 3D view')

ax = fig.add_subplot(2,2,2)
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([-200, 400, -500, 200])
plt.grid()
plt.xlabel('x')
plt.ylabel('z')
plt.title('XZ-view')

ax = fig.add_subplot(2,2,3)
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([-150, 250, -500, 200])
plt.grid()
plt.xlabel('y')
plt.ylabel('z')
plt.title('YZ-view')

ax = fig.add_subplot(2,2,4)
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([ -200, 400 ,250,-150 ])
plt.grid()
plt.xlabel('x')
plt.ylabel('y')
plt.title('XY-view')

#
# example code using plot_trisurf to visualize... this will be
# a bit slow so you have to be patiennt
#
fig1 = plt.figure()
ax = fig1.add_subplot(111, projection='3d')
ax.plot_trisurf(pts3[0,:], pts3[1,:], pts3[2,:], triangles=tri,antialiased=False)
ax.view_init(azim=0,elev=40)  #set the camera viewpointn
visutils.set_axes_equal_3d(ax)
visutils.label_axes(ax)
plt.title('Final Mesh View 1')
plt.show()
fig1 = plt.figure()
ax = fig1.add_subplot(111, projection='3d')
ax.plot_trisurf(pts3[0,:], pts3[1,:], pts3[2,:], triangles=tri,antialiased=False)
ax.view_init(azim=120,elev=40)  #set the camera viewpointn
visutils.set_axes_equal_3d(ax)
visutils.label_axes(ax)
plt.title('Final Mesh View 2')
plt.show()

#
# An alternative is to use the trimesh library (https://github.com/mikedh/trimesh)
# you will need to install it via. "pip install trimesh"
# A nice feature is that it will create a plot which is interactive (so you can
# drag with the mouse pointer to rotate the 3D mesh to view from different directions)
#
#mesh = trimesh.Trimesh(vertices=pts3.T,faces=tri[:,[0,2,1]])
#mesh.show()
```



Final Mesh View 1



Final Mesh View 2



your discussion goes here

2. First, I used Bounding Box Pruning to find noisy points outside a specific range. I kept points within the given x, y, and z values. Then, I applied triangle pruning to remove points causing triangles with edges exceeding a set limit. To calculate the edge, I created a new function using np.linalg.norm to find the distance between two points. I removed triangles failing these rules. Adjusting the x, y, and z range, as well as the edge maximum, can affect the success of the procedure. If the range is too large, there may still be lots of noise. Changing trithresh from 0.5 to 0.9 helps maintain more points, avoiding strange shapes. Also, adding more cameras improves the reconstruction process, reducing noise and requiring less user intervention for a better result. 3. I did see "pattern ambiguity" when the number of sequences of light and dark increases. It becomes challenging to distinguish between them so it should adjust the intensity of the projected light to increase the contrast between light and dark regions. I believe the placement of the cameras for the scan data is excellent, as they are neither too close nor too far from the object. This avoids difficulties in distinguishing structured light patterns and prevents excessive noise. The distance between the two cameras is also optimal, avoiding issues related to being too far away that could hinder the matching of corresponding points in the images. For the projector, if

the projector is at the same height as the object, causing the structured light patterns to give parallel lines along the box sides, it can lead to better pattern contrast on the surfaces.