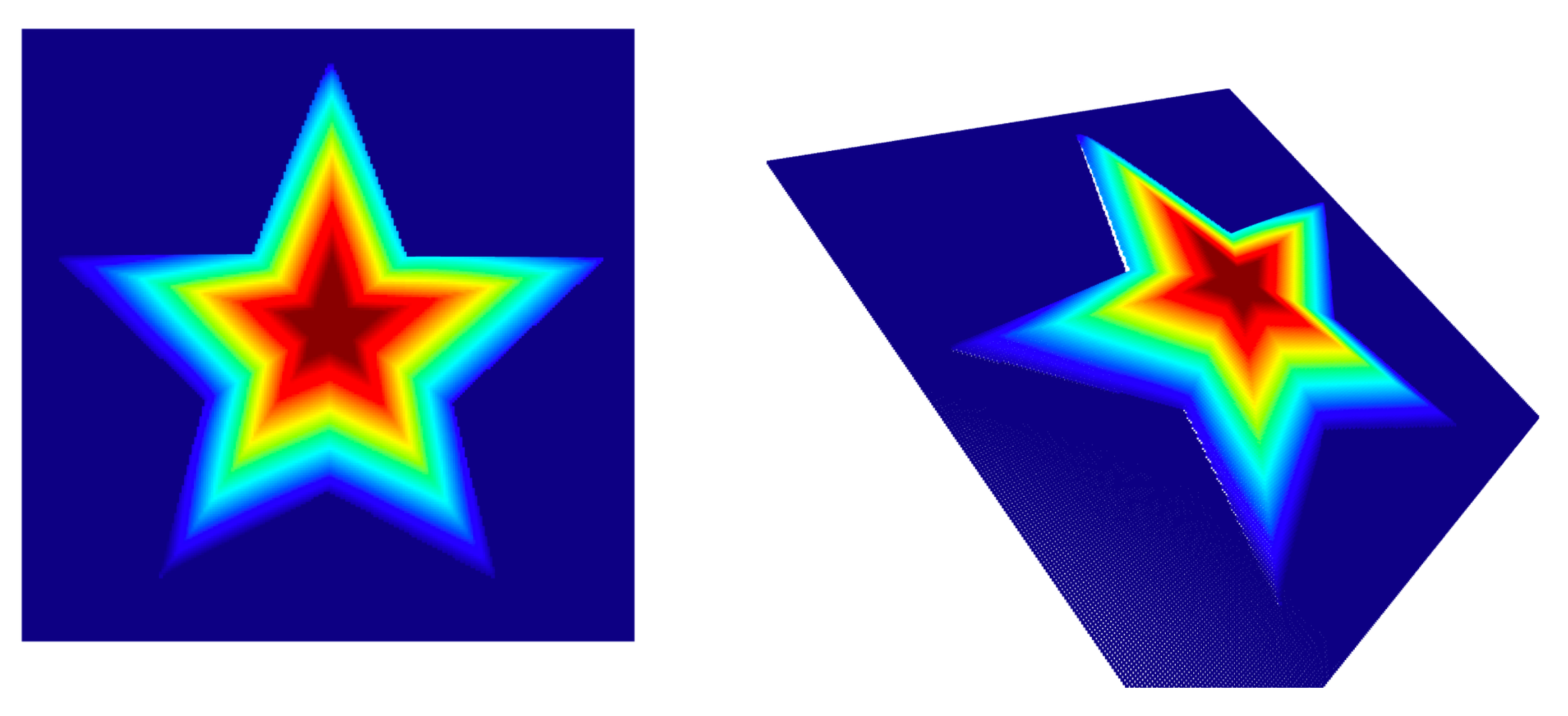
Photometric Stereo

Student_ID: 311553060

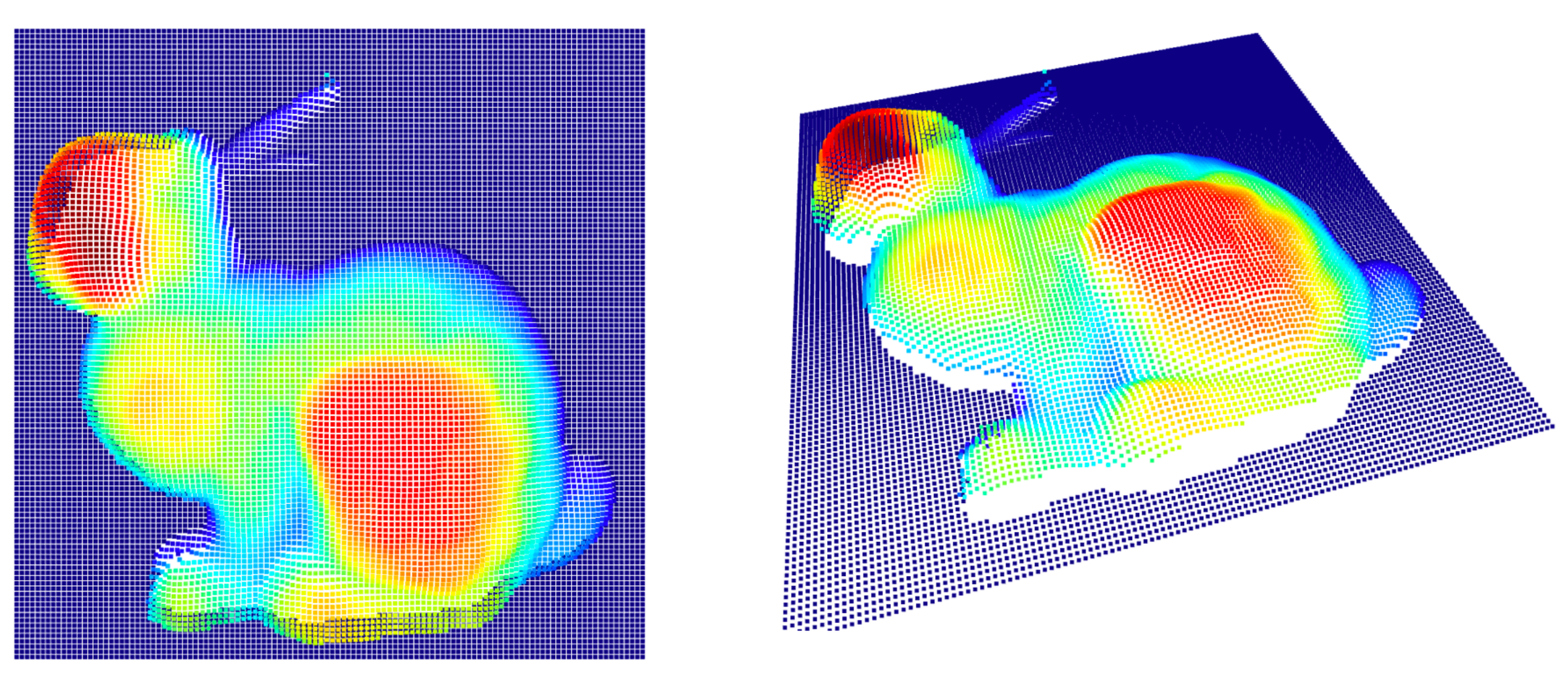
Name: 周睦鈞

1. Reconstruct surfaces of “bunny” & “star”

Star:

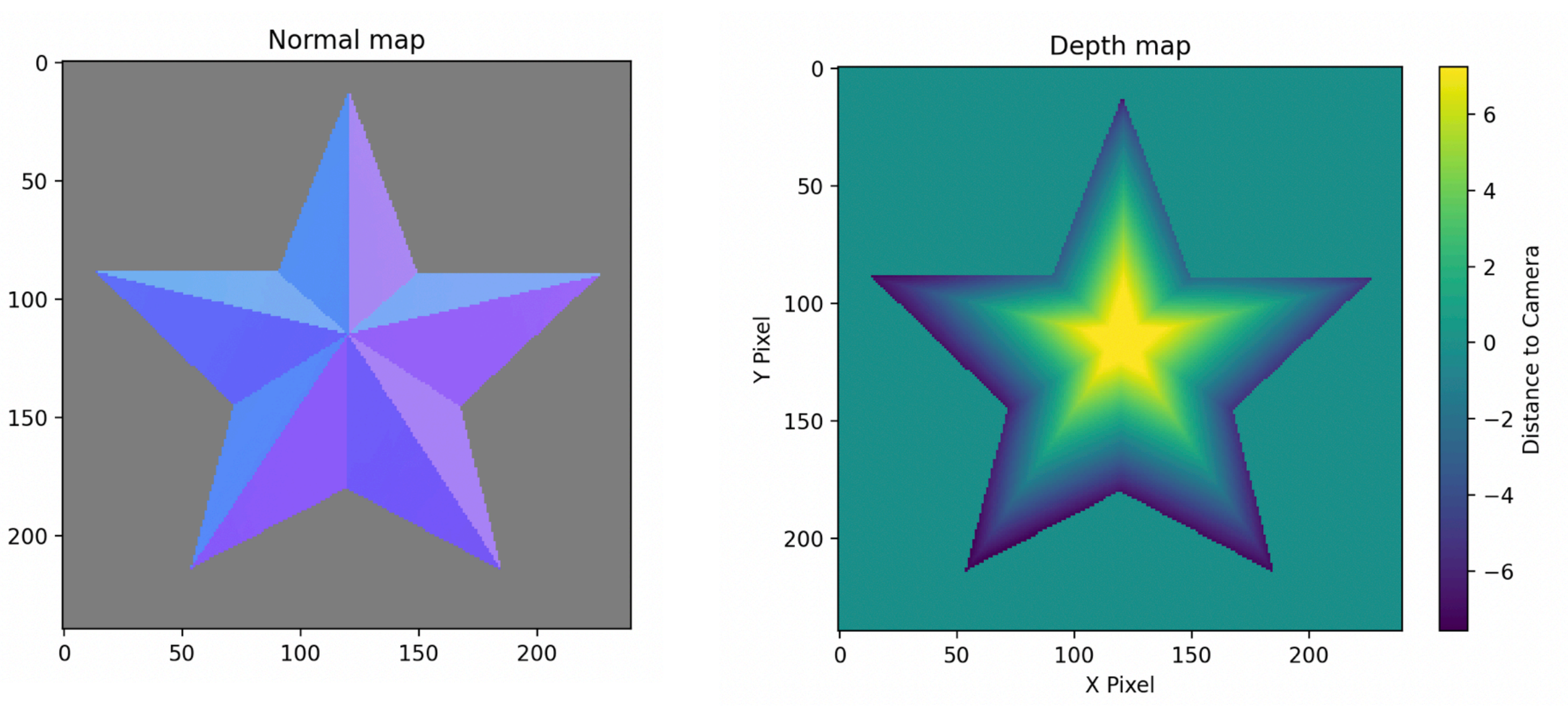


Bunny:

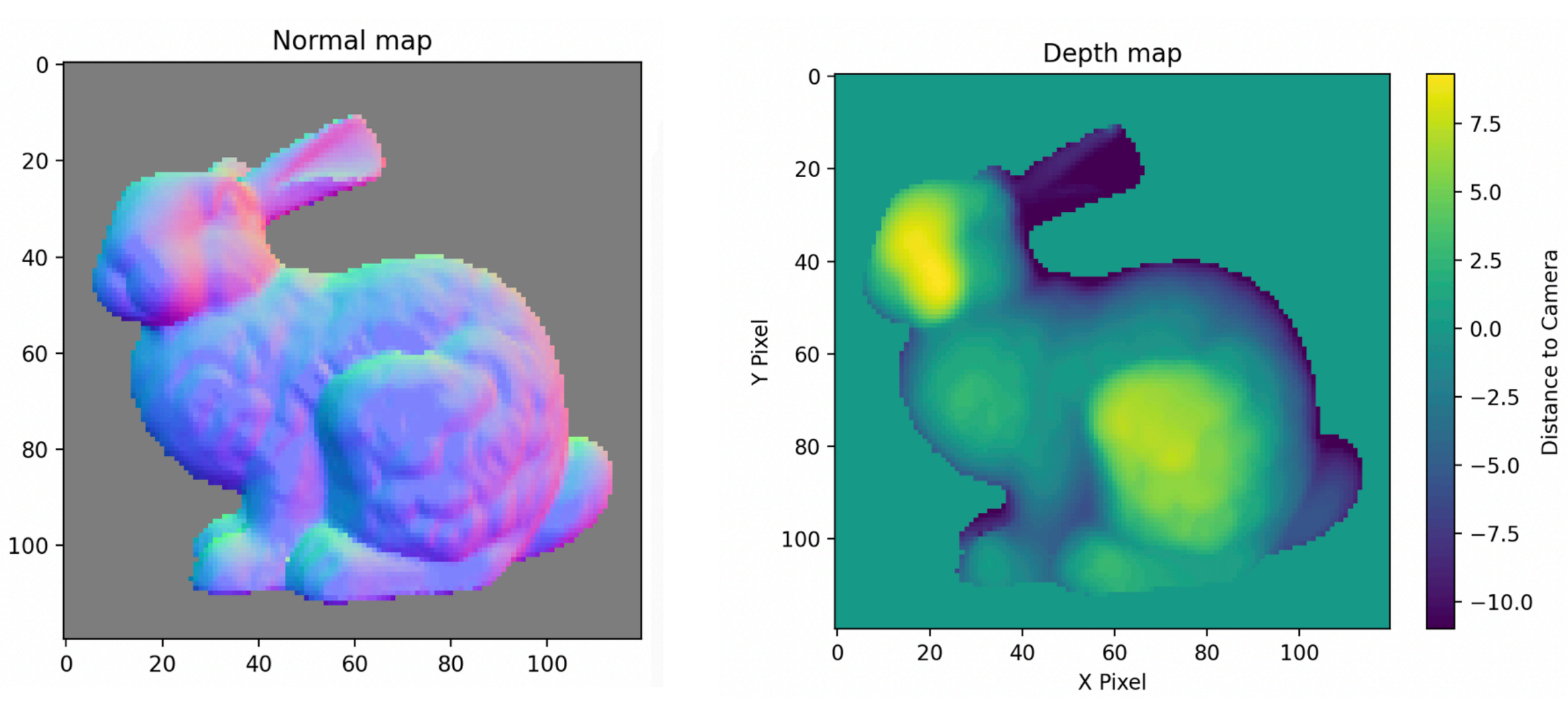


2. Show the normal map and depth map of “bunny” & “star”

Star:



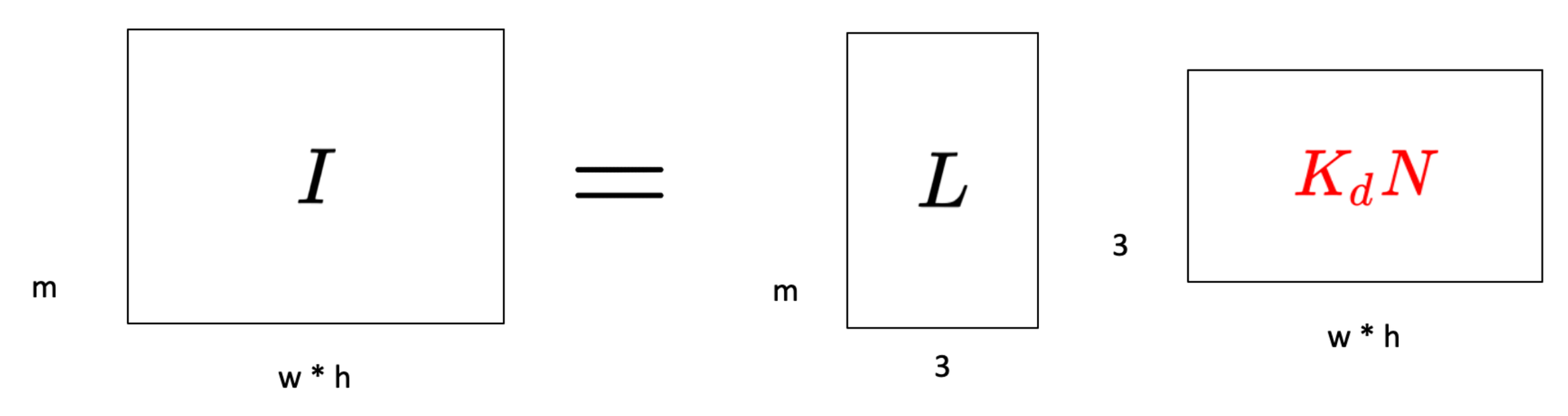
Bunny:



3. Simply explain your implementation and what kind of “method” you use to enhance the result and compare the result

a. Calculate the normal vector of the object:

$$i_{x,y}^{(m)} = l_m K_d n \quad \rightarrow \quad I = LK_d N$$




```
def read_data(filepath):
    L = []
    I = []
    num = 0

    # Get the light source from LightSource.txt
    with open(filepath + '/LightSource.txt', 'r') as f:
        for line in f.readlines():
            line = line.strip()
            point = list(map(int, line[line.find('(') + 1: line.find(')')].split(',')))
            L.append(np.array(point).astype(np.float32))
            num += 1

    # Get the "unit vector" of light source, dim of L: (m, 3), m is number of light source
    L = normalize(L, axis = 1)

    # Get image data
    for i in range(1, num + 1):
        pic = read_bmp(filepath + '/pic' + str(i) + '.bmp')
        # Flatten image size from H * W to HW
        I.append(pic.ravel())

    # Change type to ndarray
    I = np.asarray(I)

    return I, L

def get_normal_vector(I, L):
    # I = L * Kd * N, solve the equation to get Kd * N
    KdN = np.linalg.solve(L.T @ L, L.T @ I)
    KdN = normalize(KdN, axis = 0)

    return KdN.T
```

If we want to obtain the normal vector of the object, we can use the equation above $(I = LK_dN)$, where I, L, and N represent intensity, unit vector of the light source, and normal vector from m images with different positions of the light source, respectively.

To solve this equation, I used the method of pseudo-inverse by multiplying L^T on both sides of the equation. The resulting equation is: $L^T I = L^T LK_dN$.

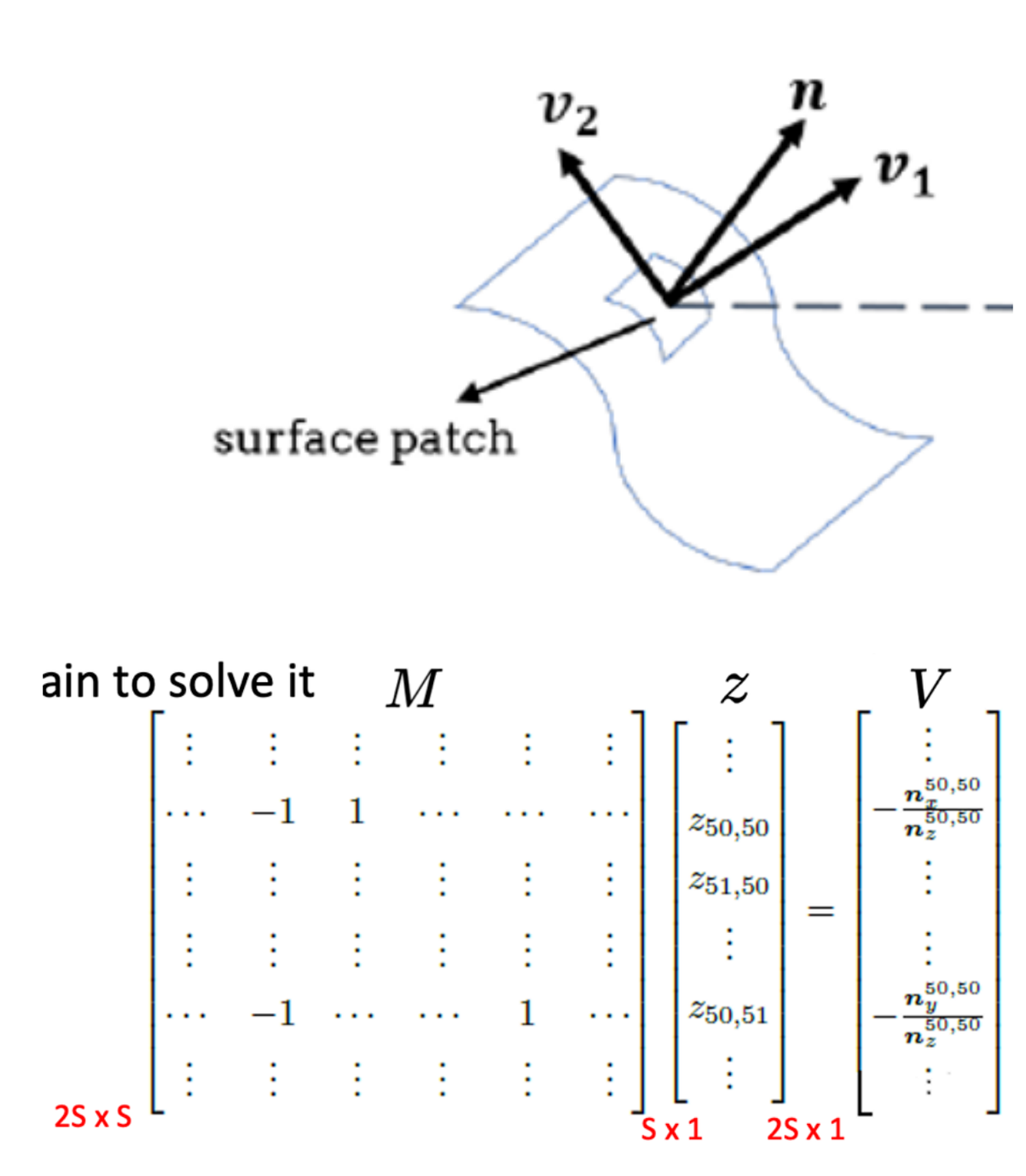
To obtain the unknown value K_dN, I used the function `np.linalg.solve(L.T @ L, L.T @ I)` provided by numpy and got the value N by normalizing the value from K_dN by using `sklearn.processing.normalize()` function.

b. Surface Reconstruction:

$$v_1 \cdot n = 0$$
$$v_2 \cdot n = 0$$

$$n_x + n_z(z_{x+1,y} - z_{x,y}) = 0$$
$$n_y + n_z(z_{x,y+1} - z_{x,y}) = 0$$

$$z_{x+1,y} - z_{x,y} = -\frac{n_x}{n_z}$$
$$z_{x,y+1} - z_{x,y} = -\frac{n_y}{n_z}$$



Since the normal vector n must be orthogonal to the vectors v1 and v2, I use this information to derive the relationship shown above the picture and construct a matrix multiplication $Mz = V$.

To be more specific, because matrix M is sparse, it is time-consuming to calculate matrix multiplication with all pixel points in the input image. Therefore, I only calculate the pixels from the object, which accelerates the matrix multiplication.

```
# number of pixels of the object
s = np.size(np.where(mask != 0)[0])

# Mz = V
M = scipy.sparse.lil_matrix((2*s, s))
v = np.zeros((2*s, 1))

# Fill the value into M and v
M, v = fill_value_into_matrix(M, v, s, N, mask)

# M.T * M * z = M.T * v
z = scipy.sparse.linalg.spsolve(M.T @ M, M.T @ v)
```

```
def fill_value_into_matrix(M, v, s, N, mask):
    nonzero_h, nonzero_w = np.where(mask!=0)

    # Calculate the index number used in filling M
    index_array = np.zeros((image_row, image_col)).astype(np.int16)
    for i in range(s):
        index_array[nonzero_h[i], nonzero_w[i]] = i

    for i in range(s):
        h = nonzero_h[i]
        w = nonzero_w[i]
        n_x = N[h, w, 0]
        n_y = N[h, w, 1]
        n_z = N[h, w, 2]

        # z(x+1, y) - z(x, y) = - nx / nz
        j = i*2
        if mask[h, w+1]:
            k = index_array[h, w+1]
            M[j, i] = -1
            M[j, k] = 1
            v[j] = -n_x/n_z
        elif mask[h, w-1]:
            k = index_array[h, w-1]
            M[j, k] = -1
            M[j, i] = 1
            v[j] = -n_x/n_z

        # z(x, y+1) - z(x, y) = -ny / nz
        j = i*2+1
        if mask[h+1, w]:
            k = index_array[h+1, w]
            M[j, i] = 1
            M[j, k] = -1
            v[j] = -n_y/n_z
        elif mask[h-1, w]:
            k = index_array[h, w-1]
            M[j, k] = 1
            M[j, i] = -1
            v[j] = -n_y/n_z

    return M, v
```

After getting M and V, I used the pseudo-inverse method again by multiplying M^T on both sides of the equation $(M^T Mz = M^T V)$ and solved the equation to obtain the value of z by using the function `z = scipy.sparse.linalg.spsolve(M.T @ M, M.T @ v)` provided by scipy.

Then it is easy for me to reconstruct the surface by replacing the estimated value z back to the input image.

c. Optimize the reconstructed surface

During the process of reconstructing the surface, we may encounter some outliers which have values that are far from the majority of the points. In this case, we reset the values of points that are above or below 2 standard deviations to maximum or minimum normalized value in the remaining point set. This may help to make the entire surface look smoother.

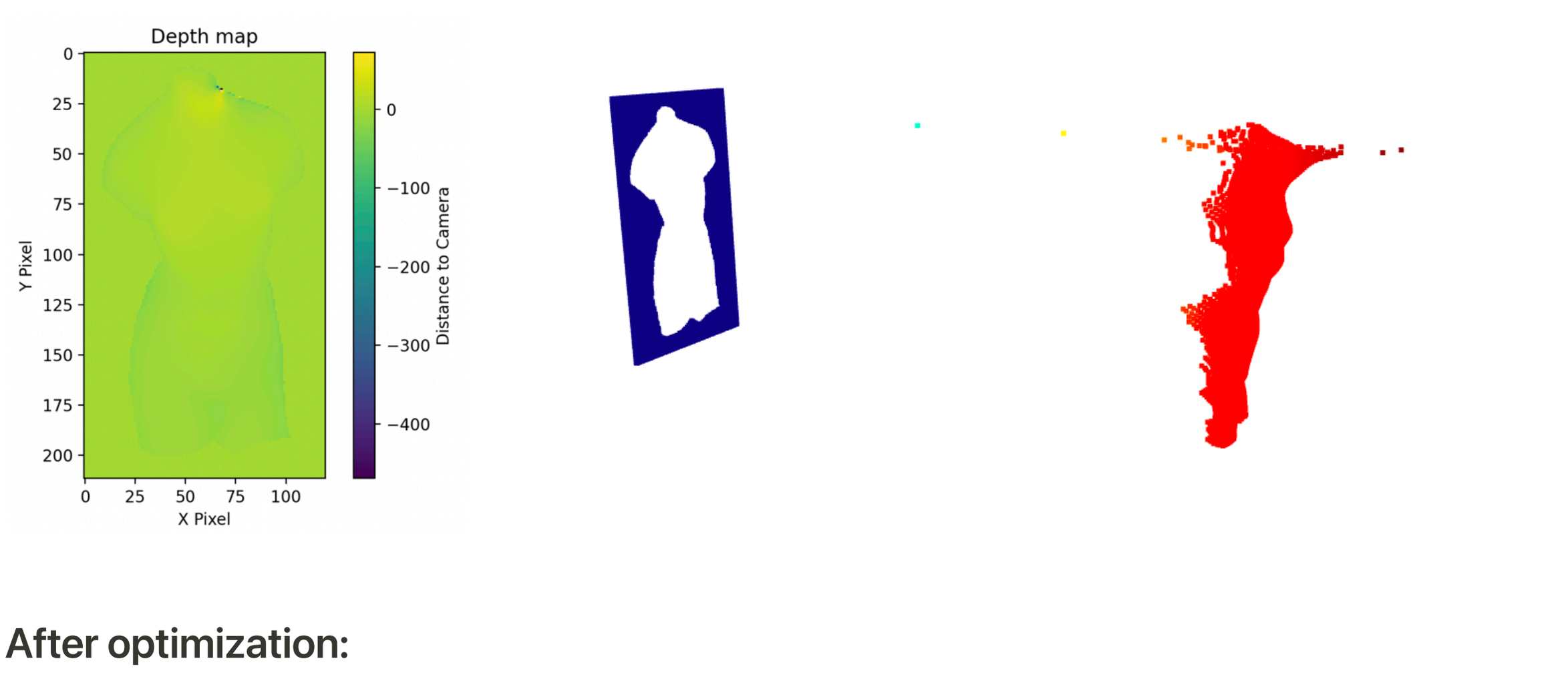
```
def get_surface_reconstruction(z, mask, s, optimize):
    nonzero_h, nonzero_w = np.where(mask!=0)

    # Filter strange point in z
    normalized_z = (z - np.mean(z)) / np.std(z)
    outliner_idx = np.abs(normalized_z) > 2
    z_max = np.max(z[~outliner_idx])
    z_min = np.min(z[~outliner_idx])

    Z = mask.astype(np.float32)

    if optimize:
        for i in range(s):
            if z[i] > z_max:
                Z[nonzero_h[i], nonzero_w[i]] = z_max
            elif z[i] < z_min:
                Z[nonzero_h[i], nonzero_w[i]] = z_min
            else:
                Z[nonzero_h[i], nonzero_w[i]] = z[i]
    else:
        for i in range(s):
            Z[nonzero_h[i], nonzero_w[i]] = z[i]

    return Z
```

4. Reconstruct surfaces of “venus”

Before optimization:

As we can seen from the picture below, there exist some extreme points affect the final surface reconstruction result.

Therefore, we adopted the method mentioned above to deal with these outliners.



After optimization: