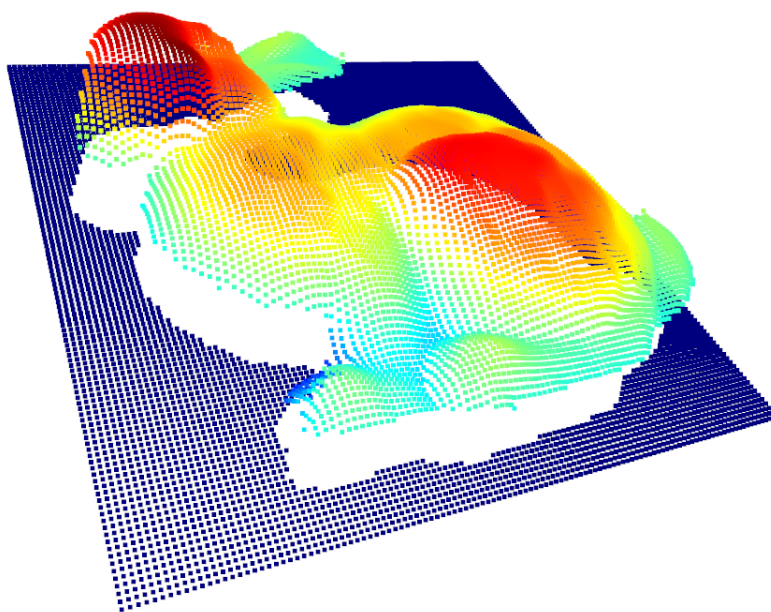
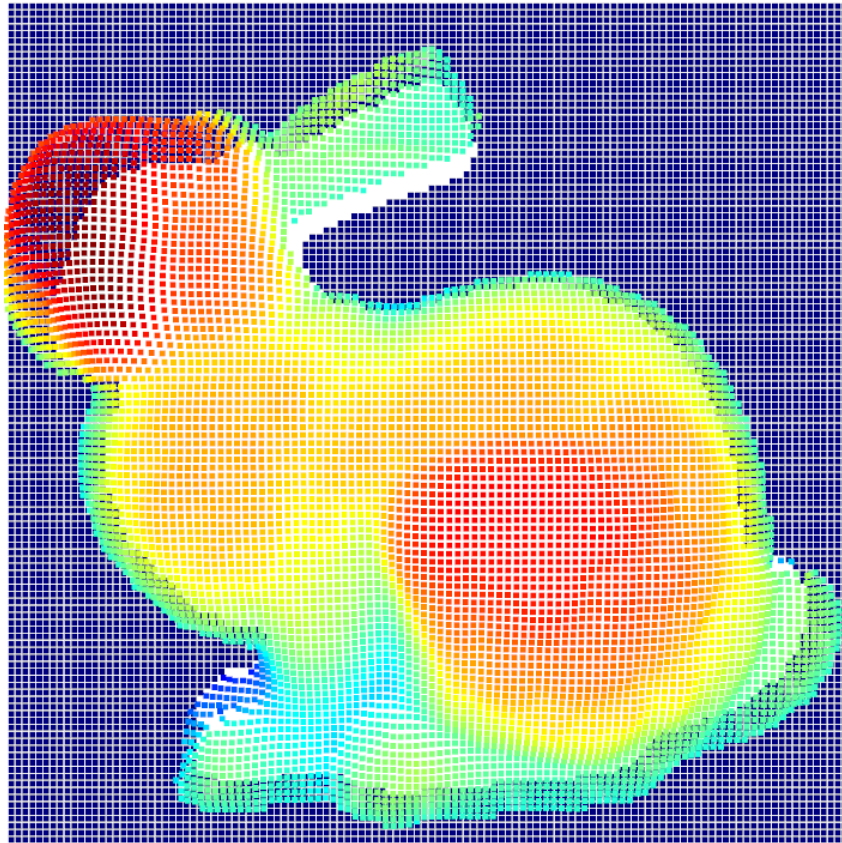
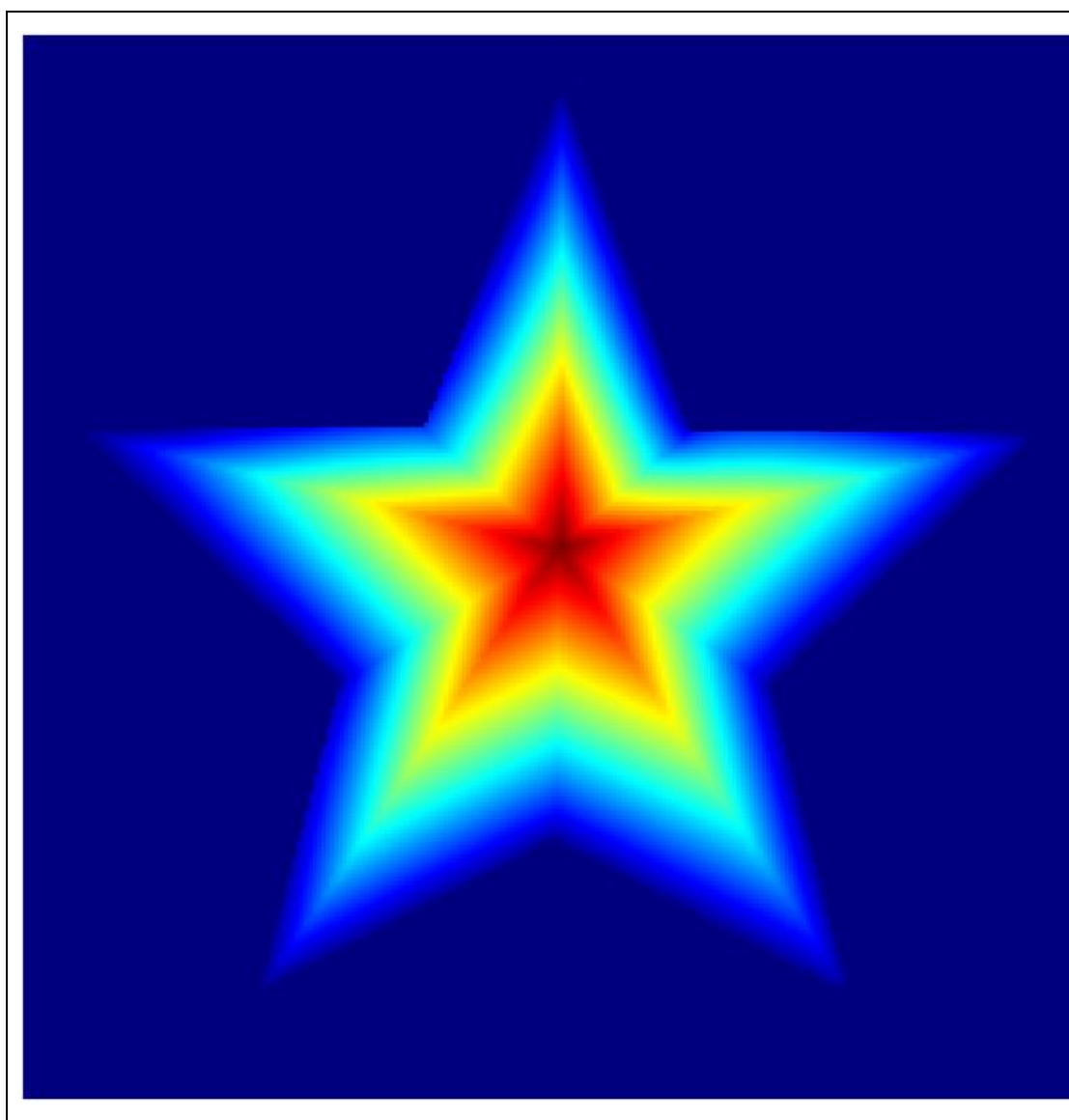
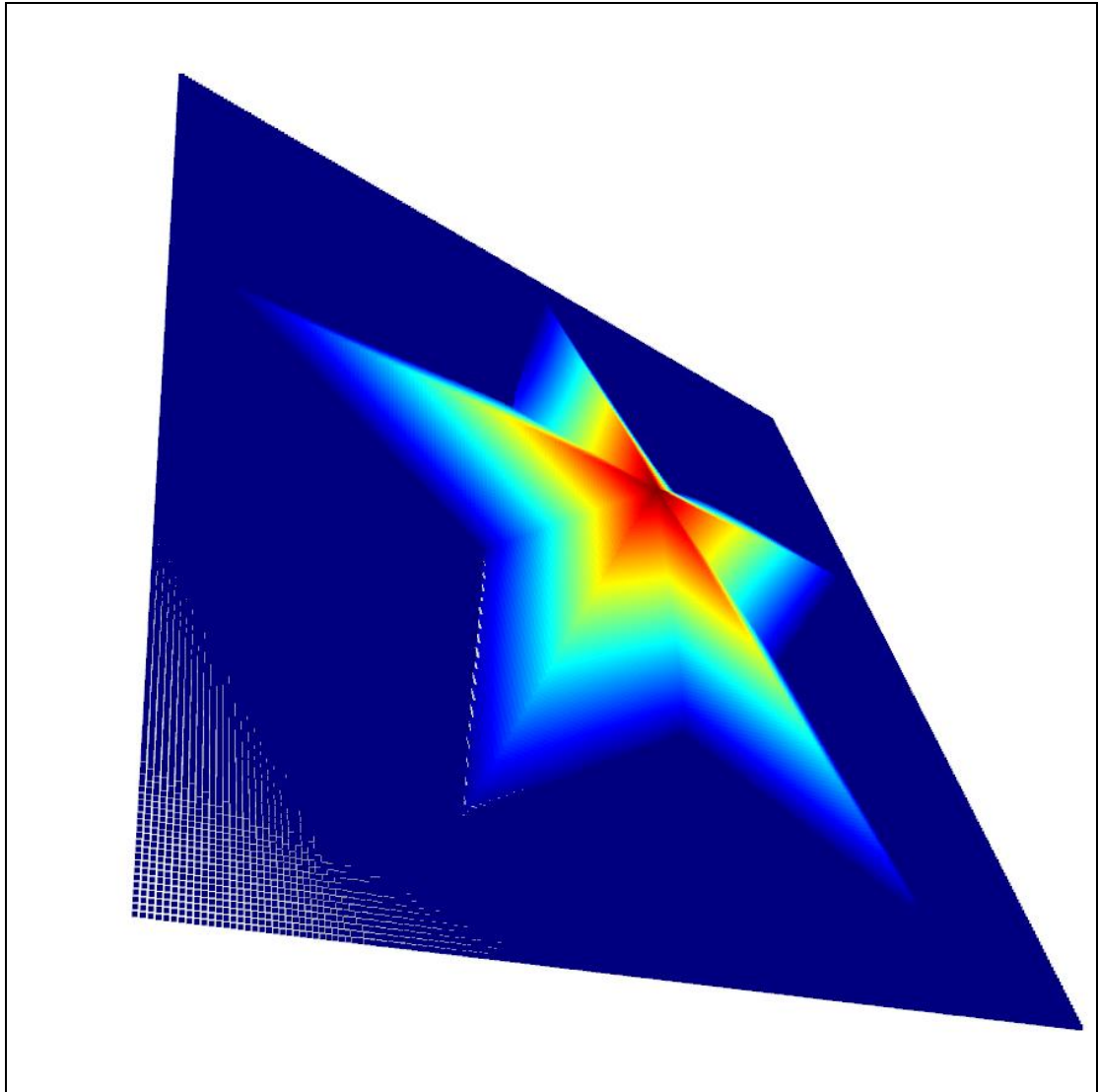


Computer Vision Hw1 report

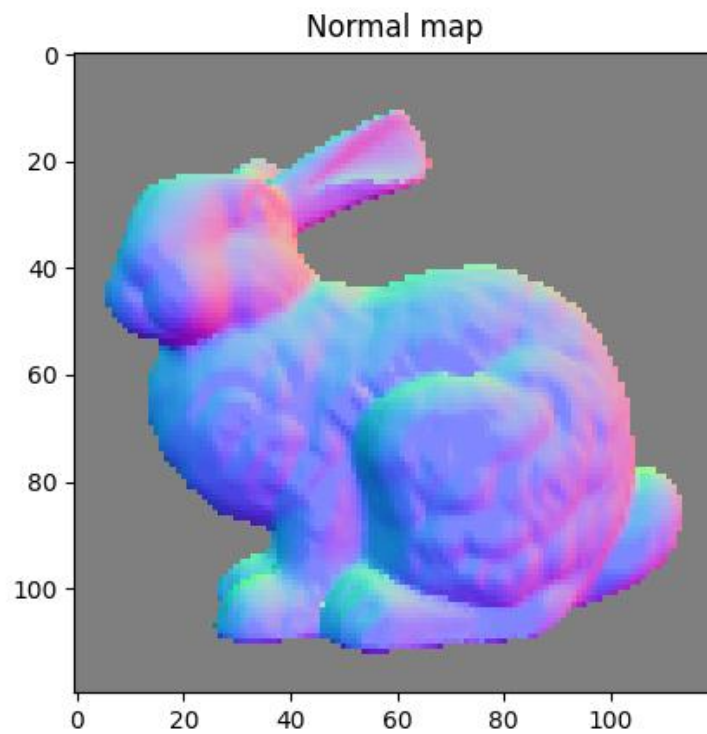
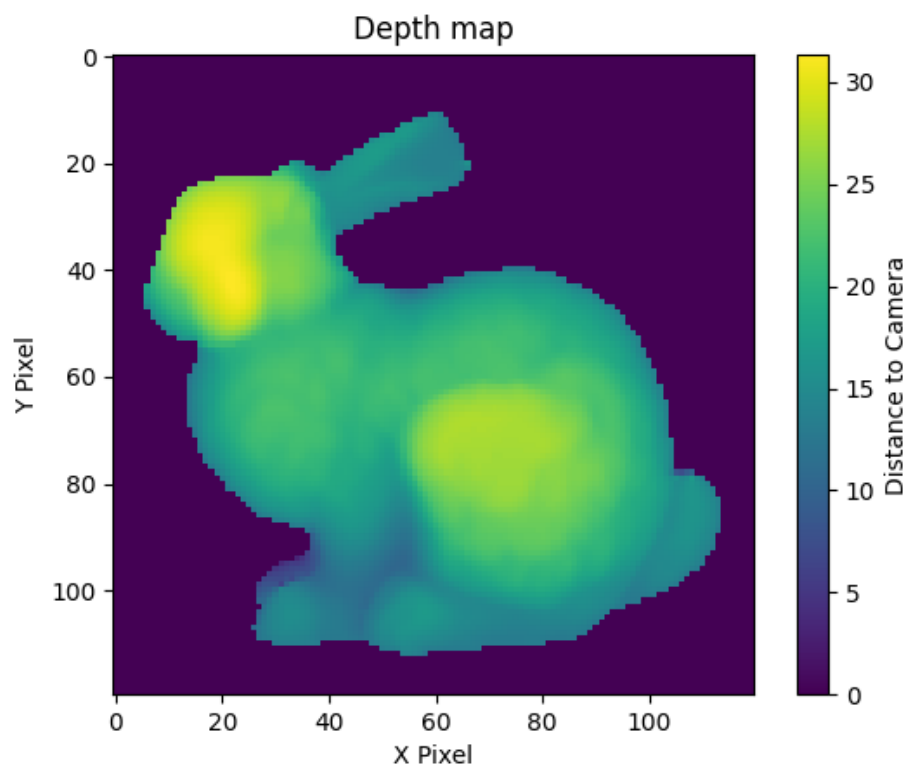
1. Reconstruct the surface of bunny and star.

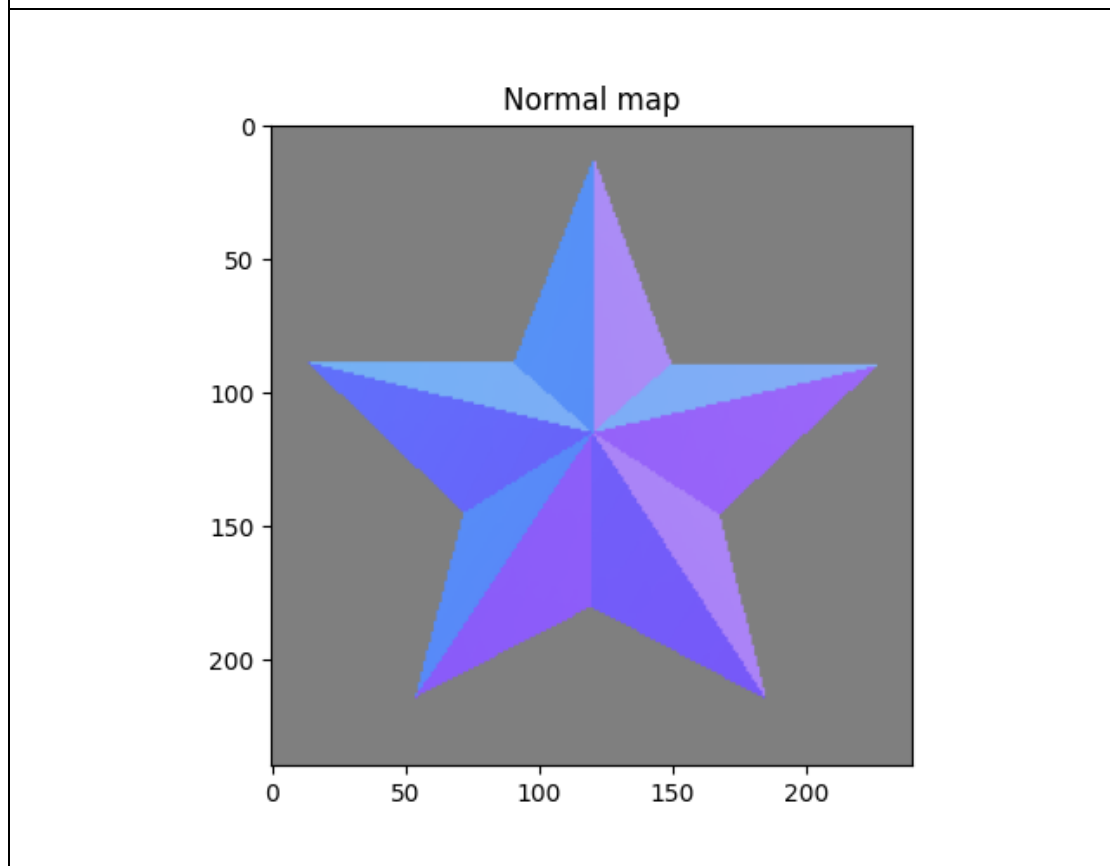
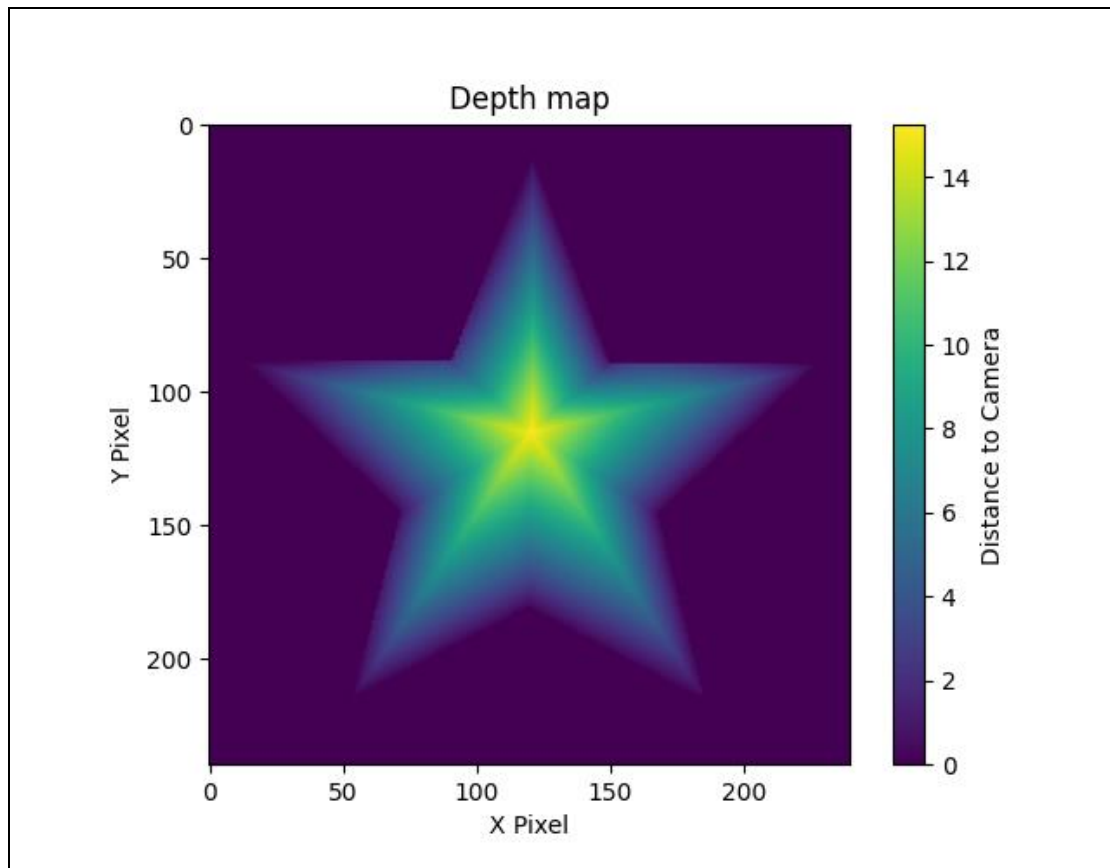






2. Show the normal map and depth map of bunny and star.



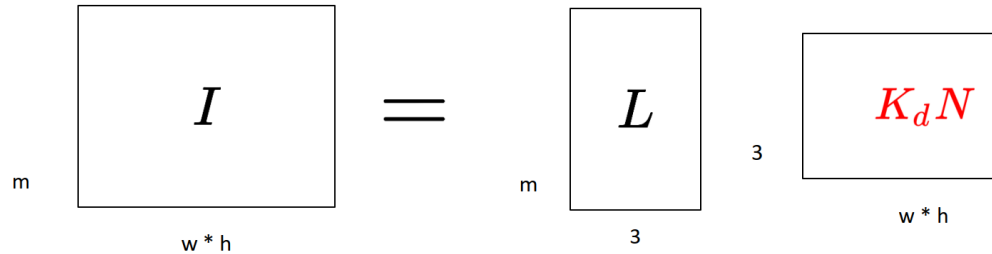


3. Simply explain the implementation and what

kind of method you use to enhance the result
and compare the result.

For reading images and light sources, it just simple so I would skip them.

First, we should calculate the normal vectors.

$$i_{x,y}^{(m)} = l_m K_d \mathbf{n} \longrightarrow I = L K_d \mathbf{N}$$


The diagram illustrates the matrix equation $I = L K_d N$. Matrix I has dimensions m (rows) by $w * h$ (columns). Matrix L has dimensions m (rows) by 3 (columns). Matrix $K_d N$ has dimensions 3 (rows) by $w * h$ (columns).

$$L^T I = L^T L K_d \mathbf{N}$$

From the formulation above, to solve $K_d N$, I simple use numpy to solve the least square solution x of

$$L^T L x = L^T I$$

And get the $x = K_d N$, then normalize it to get normal vectors.

```

142 def calculate_normal_vector(images, light_sources):
143     # calculate the normal vector
144     G = np.linalg.lstsq(light_sources.T @ light_sources, light_sources.T @ images, rcond=None)[0]
145
146     # normalize the normal vector
147     G = G.T
148     G = normalize(G)
149     G = G.reshape((image_row, image_col, 3))
150     return G

```

Second, to calculate the depth map.

$$M^T M z = M^T V$$

$$z = (M^T M)^{-1} M^T V$$

I can construct a sparse matrix by the formula above. To decide whether a pixel should be included, I would mask and only calculate the pixels with light value greater than zero in the original image.

I first create a map to record the indices of unmasked (those need to calculate) pixels, and create a sparse matrix to operate.

```

160 def calculate_depth_map(g, mask):
161     pixel_count = np.count_nonzero(mask)
162     pixel_index = np.where(mask)
163
164     ind = np.zeros( shape: (image_row, image_col), dtype=int)
165     ind.fill(-1)
166
167     for i, _ in enumerate(zip(*pixel_index)):
168         ind[_[0], _[1]] = i
169
170     V = np.zeros( shape: (2 * pixel_count, 1), dtype=np.float32)
171     # M = np.zeros((2 * pixel_count, pixel_count))
172     M = scipy.sparse.lil_matrix( arg1: (2 * pixel_count, pixel_count), dtype=np.float32)

```

After that, I test through all the recorded pixels and test which side of it is in the object. Since the formulation assume the right side and upside is in the object, so if it is not satisfied, the coefficient should change sign.

Note that the image is stored in (y, x), so I choose to use h and w to prevent confusion (This bothered me a lot when I started doing this part).

```

174     for i in range(pixel_count):
175         h = pixel_index[0][i] # y
176         w = pixel_index[1][i] # x
177         n = G[h, w]
178
179         if w + 1 < image_col and mask[h, w + 1] ≥ 0:
180             V[2 * i, 0] = -n[0] / n[2]
181             M[2 * i, i] = -1
182             M[2 * i, ind[h, w + 1]] = 1
183         elif w - 1 ≥ 0 and mask[h, w - 1] ≥ 0:
184             V[2 * i, 0] = -n[0] / n[2]
185             M[2 * i, i] = 1
186             M[2 * i, ind[h, w - 1]] = -1
187
188         if h + 1 < image_row and mask[h + 1, w] ≥ 0:
189             V[2 * i + 1, 0] = -n[1] / n[2]
190             M[2 * i + 1, i] = 1
191             M[2 * i + 1, ind[h + 1, w]] = -1
192         elif h - 1 ≥ 0 and mask[h - 1, w] ≥ 0:
193             V[2 * i + 1, 0] = -n[1] / n[2]
194             M[2 * i + 1, i] = -1
195             M[2 * i + 1, ind[h - 1, w]] = 1

```

After finishing the big matrix, we can get the depth values z by solving the least square solution z of

$$M^T M z = M^T V$$

Then fill all the values into its index, the depth map is completed.

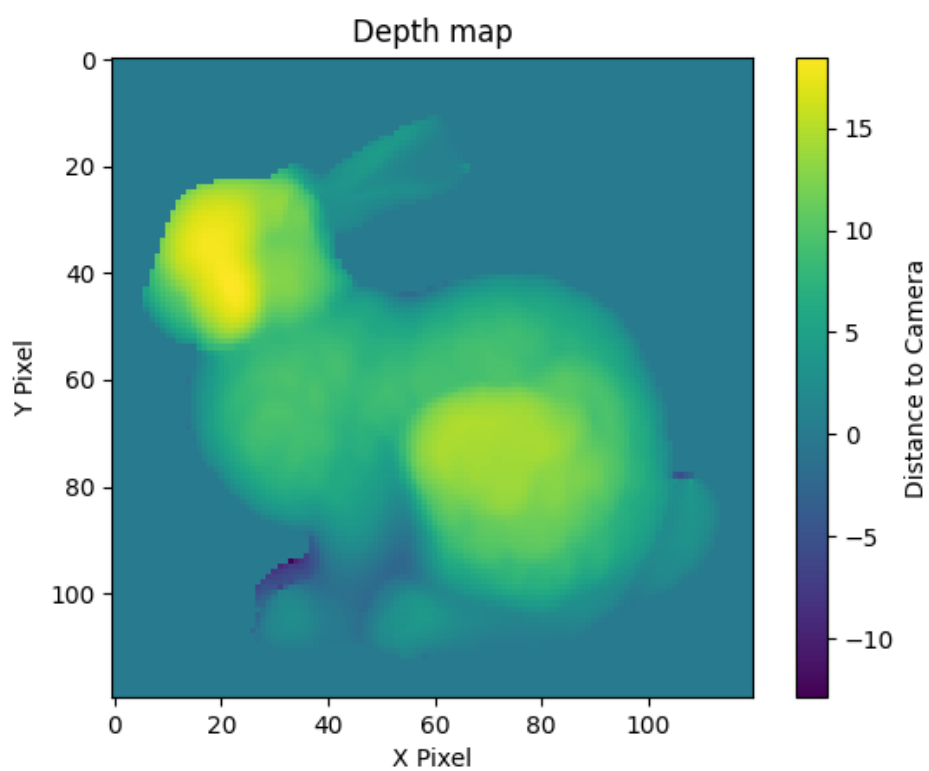
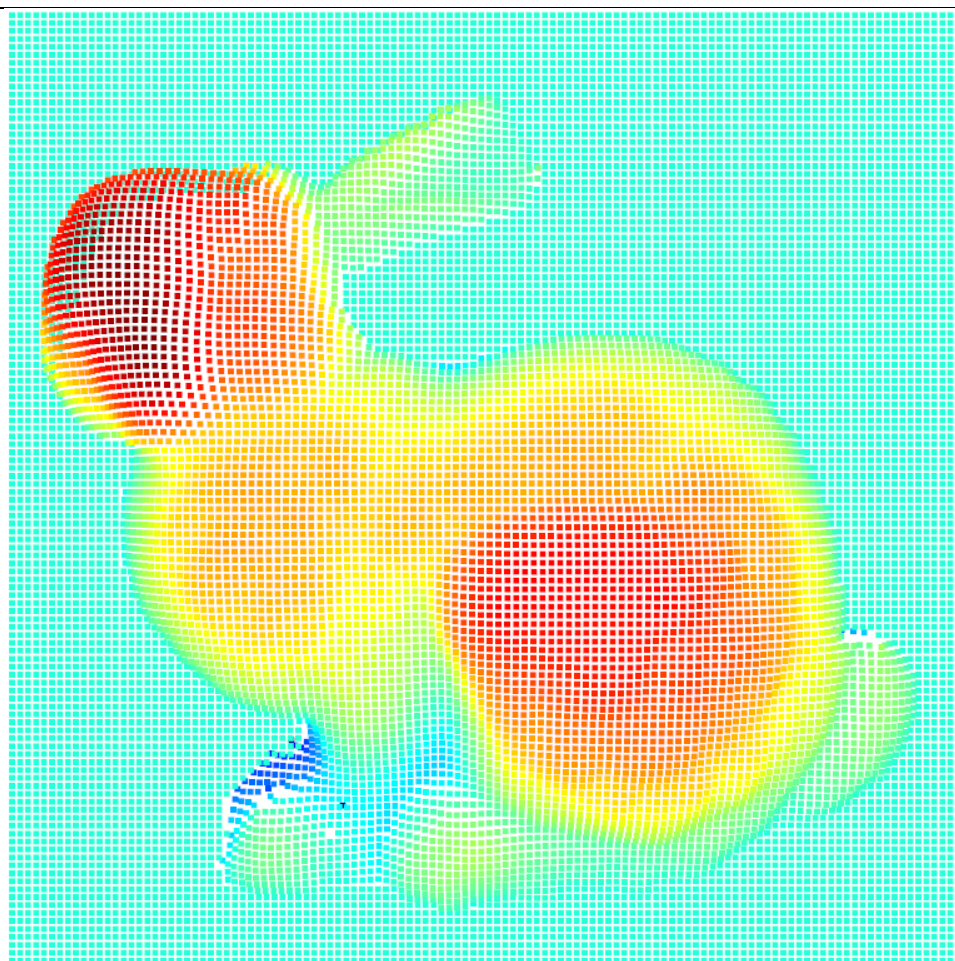

```

197     print('Solving the linear system...')
198     MM = M.T @ M
199     MV = M.T @ V
200     z = scipy.sparse.linalg.spsolve(MM, MV)
201     z -= np.min(z)
202
203     Z = np.zeros_like(mask, dtype=np.float32)
204
205     for i in range(pixel_count):
206         Z[pixel_index[0][i], pixel_index[1][i]] = z[i]
207
208     return Z

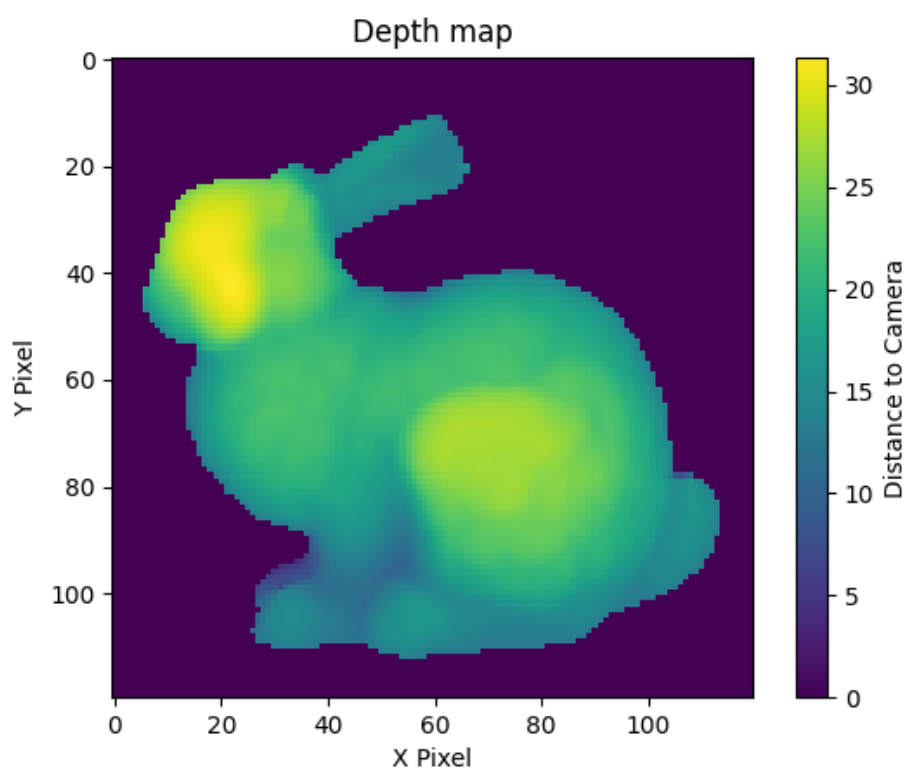
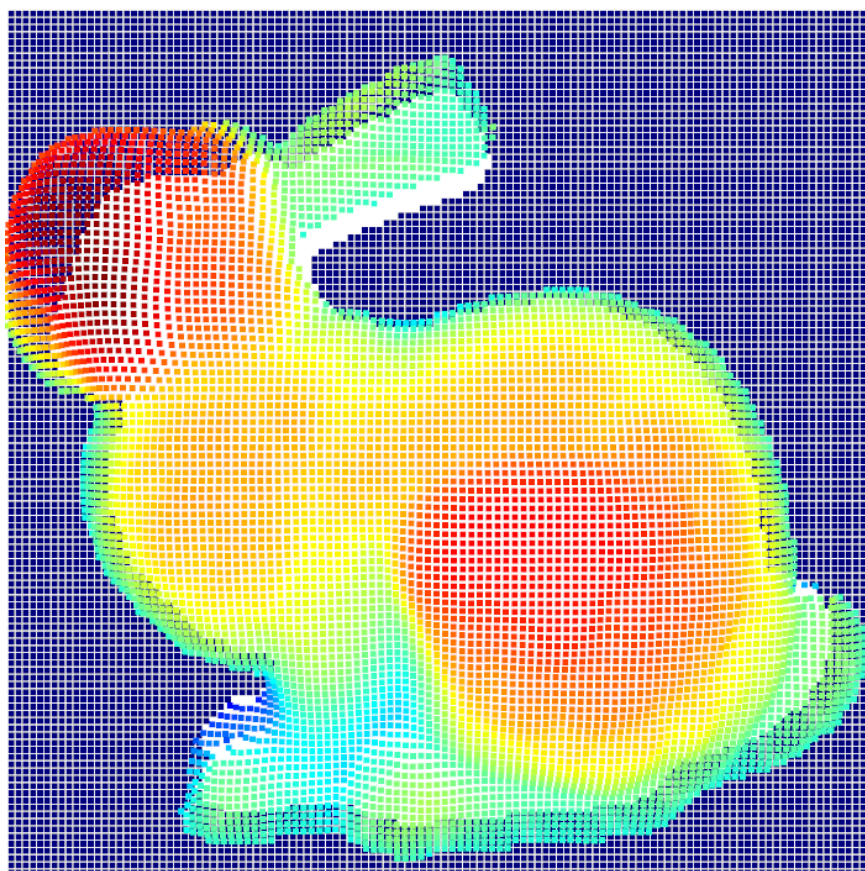
```

To enhance the result, I've conducted two methods. First one, if we directly use the solved z as the depth values, then there might be somewhere negative. So, I subtracted the minimum value from the whole depth vector.

The depth map before using this method, you can see some pixel is below the surface:

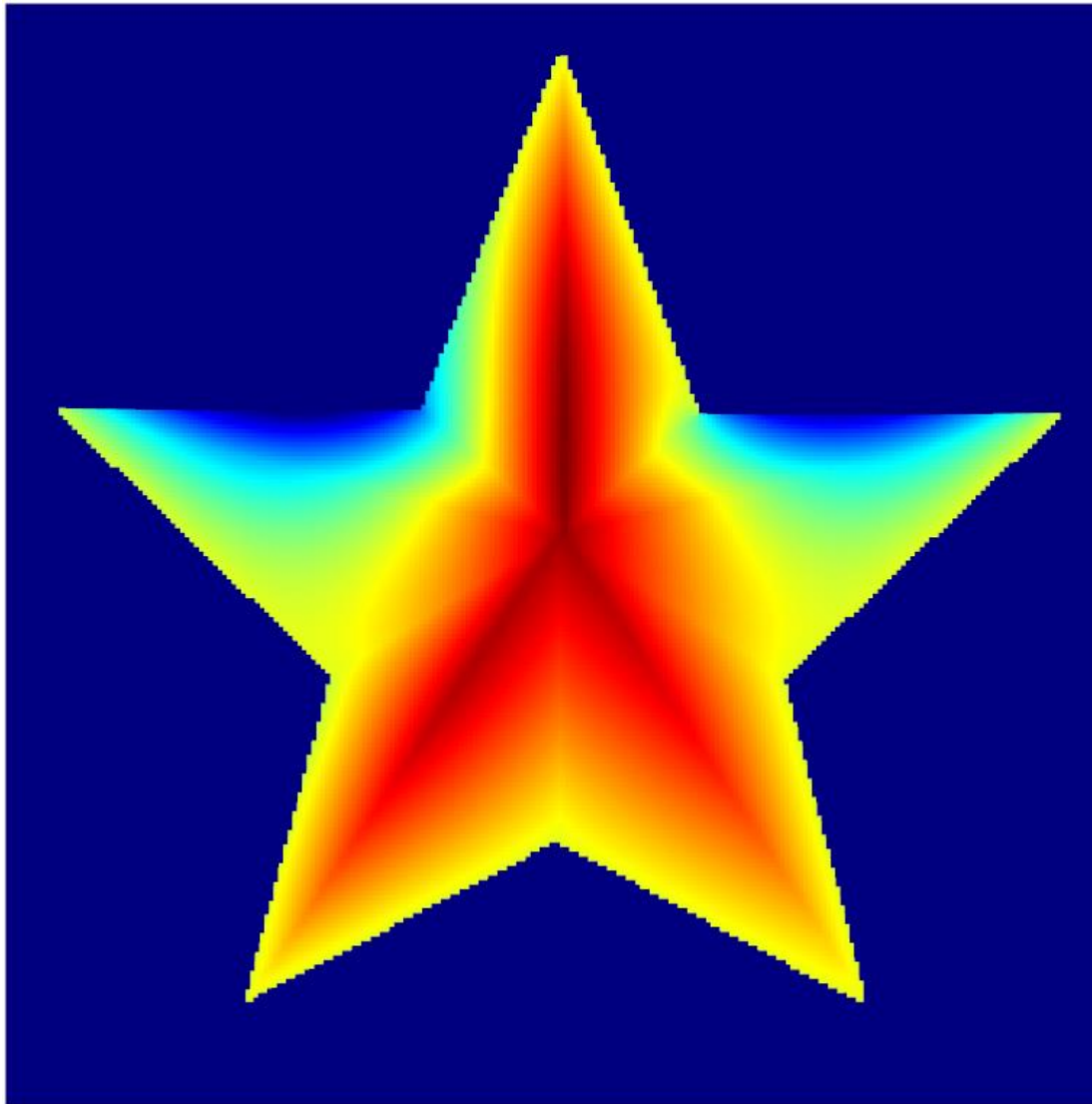


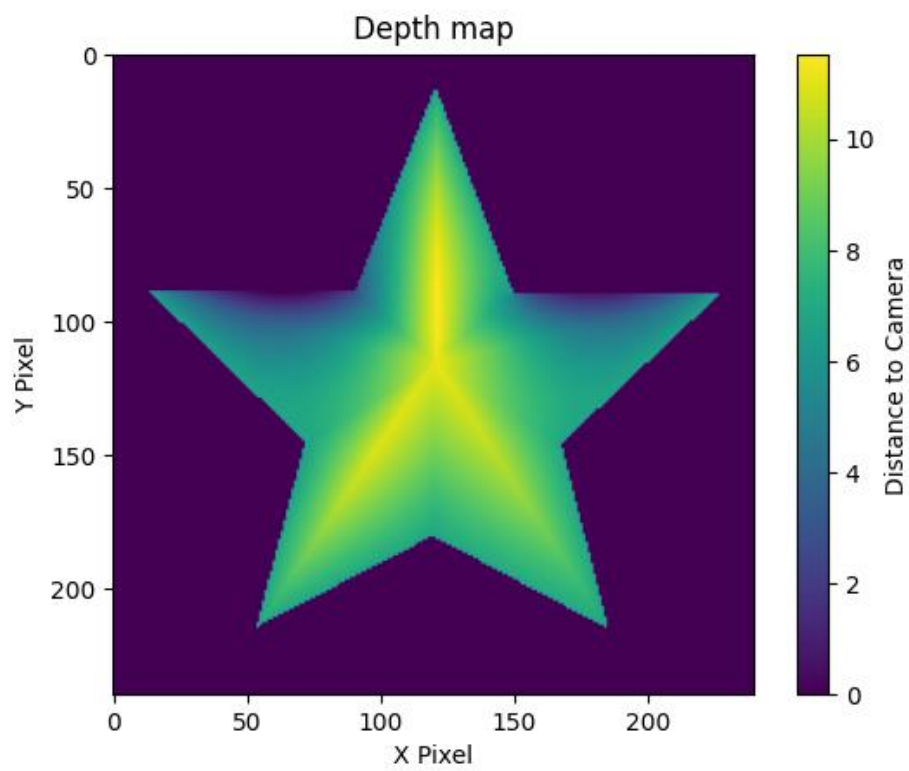
After applying:



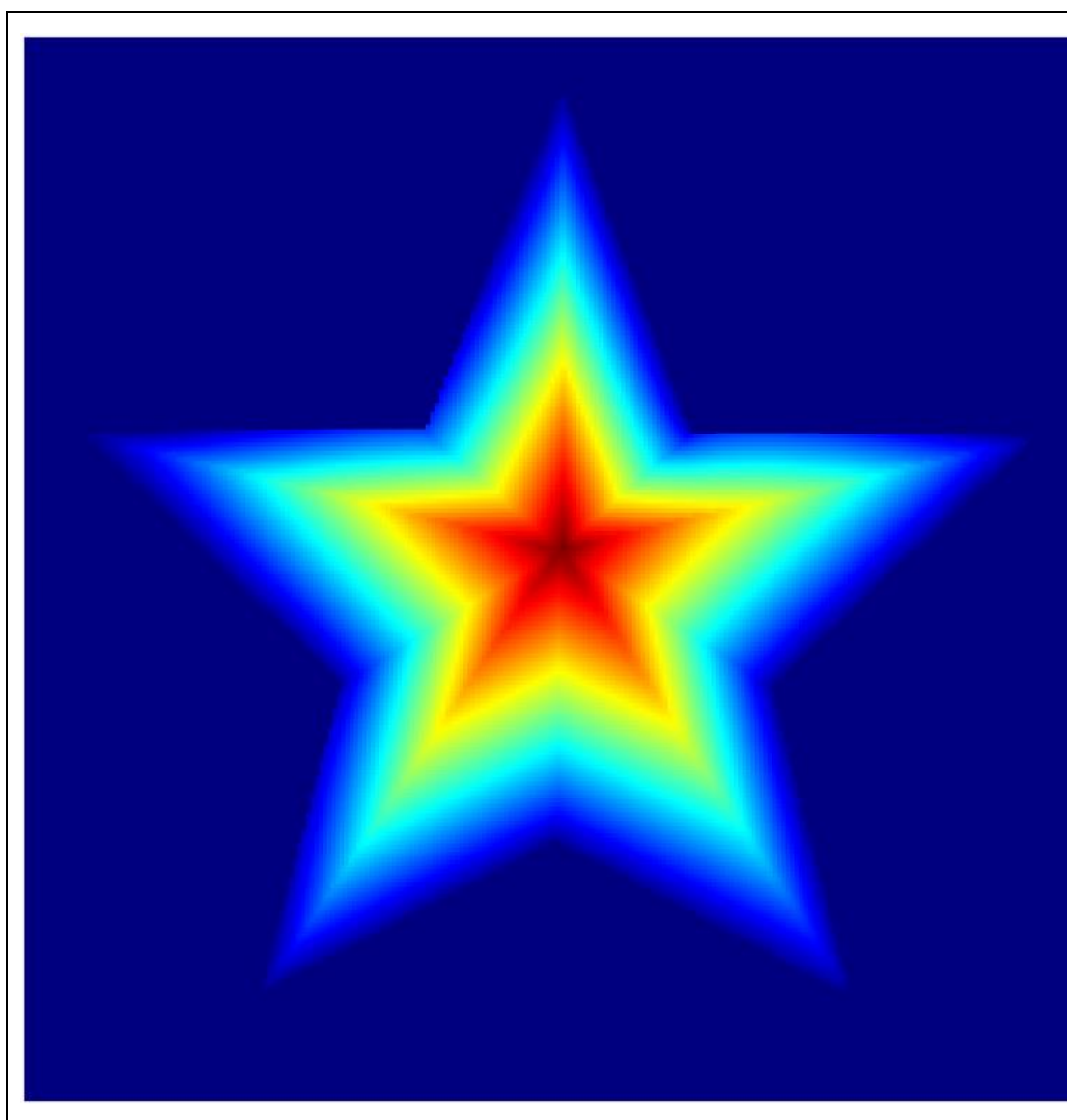
The second method is to normalize the light vector first. I think this can lessen the difference between the distance of light sources and focus more on the angle.

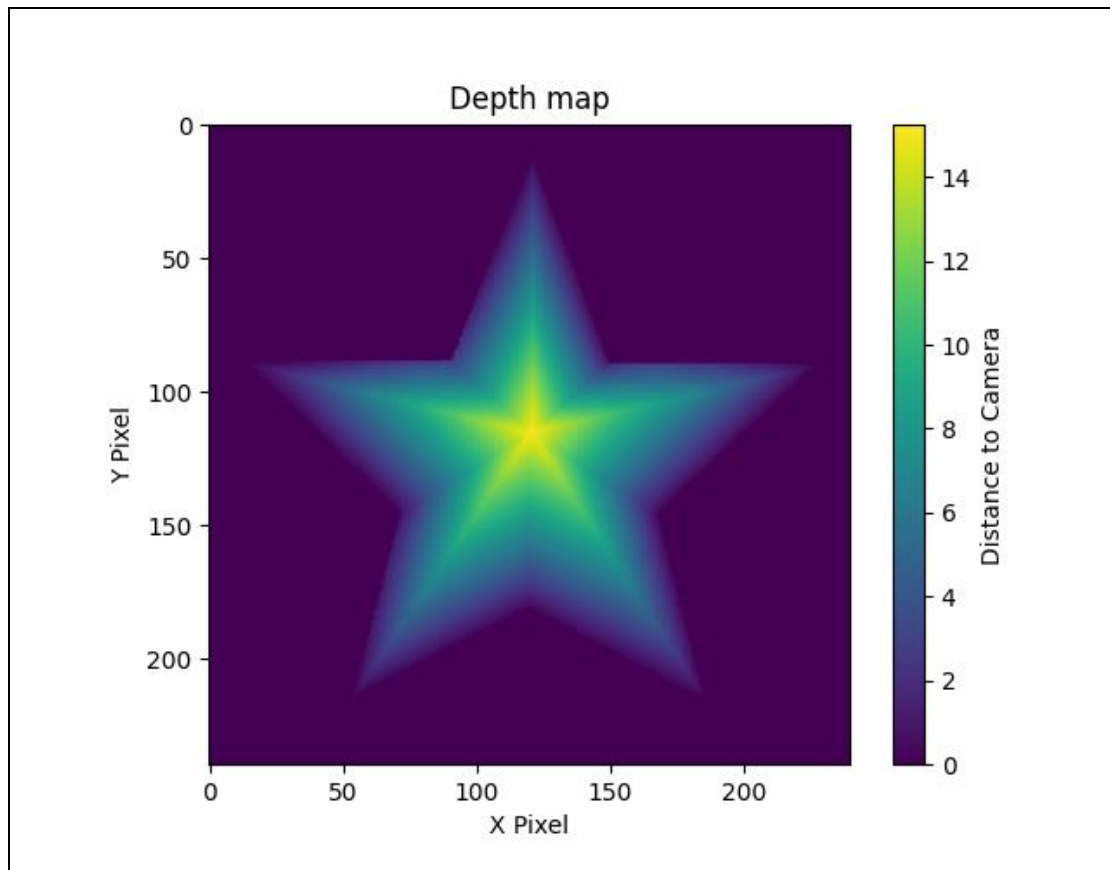
Before applying:





After applying:

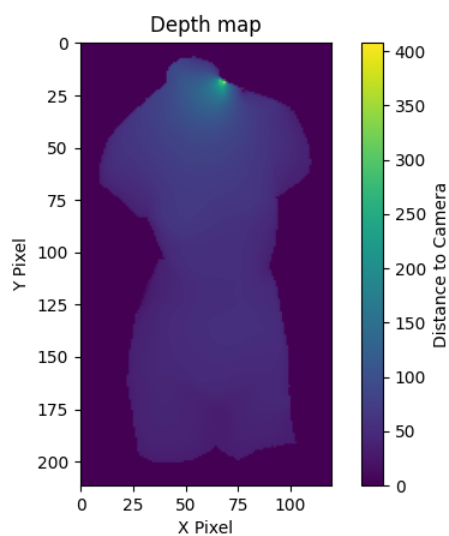




4. Reconstruct the surface of venus

Since there are some extreme large values in the calculated depth map, so I should take some method to deal with the problem.

Original depth map:

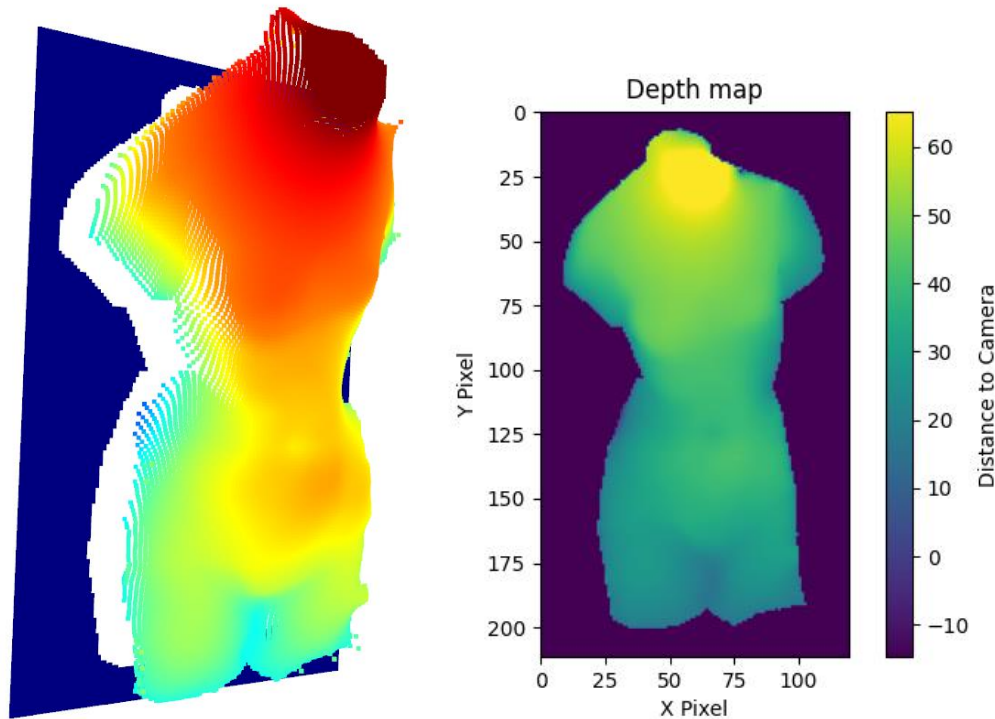


To deal with the extreme values, I discovered that the

extreme values have depth value larger than 70, while other part have depth values around 55. So, I decided to replace the pixels with $z > 60$ to

$$z_{new} = \min\left(\sqrt{z_{original} * 60, 80}\right)$$

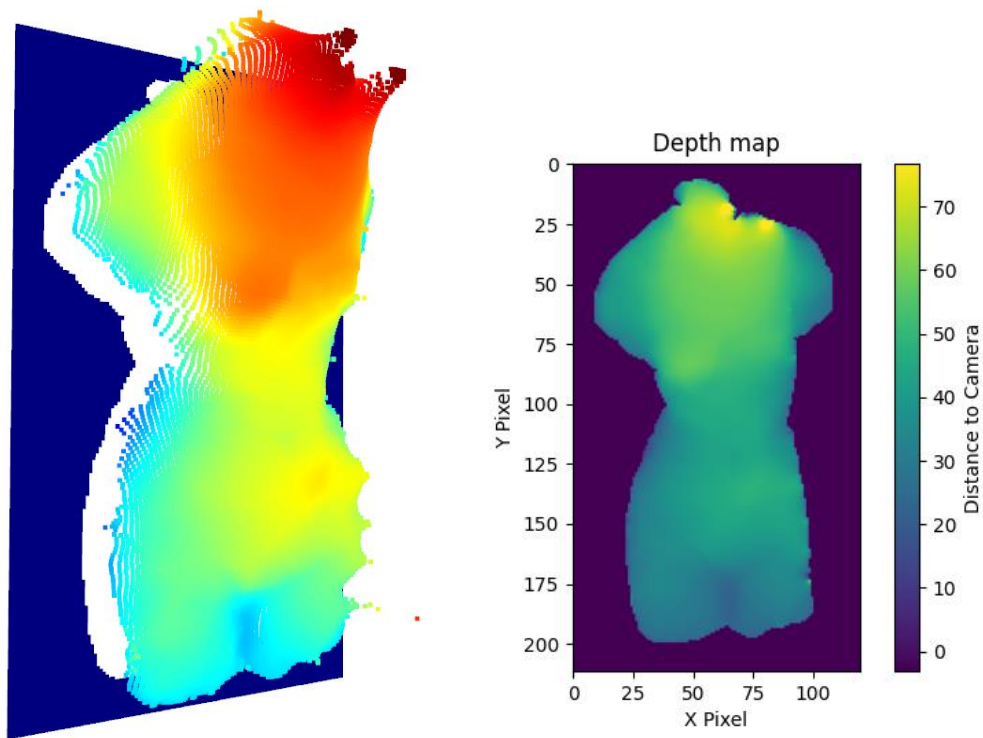
Since I've tried other methods like Gaussian blur, median blur, and take weighted average with threshold, I found this method has the best result.



5. Reconstruct the surface of noisy venus

After trying lots of methods, including write a function to average the pixels in the object and remove the white dots, some blurring functions. I found that they didn't work.

In current setup, I preprocess the noisy images with erosion and dilation. After a lot of trials and errors, I took three footprints and process the image by a sequence of erosion and dilation, and get the result below.



```

94     def read_noisy_image(filepath):
95         images = []
96         pattern1 = morphology.square(4)
97         pattern2 = morphology.disk(3)
98         pattern3 = morphology.diamond(3)
99         for fn in [f'{filepath}/pic_{_}.bmp' for _ in range(1, 7)]:
100             image = read_bmp(fn)
101
102             image = morphology.erosion(image, pattern1)
103             image = morphology.erosion(image, pattern2)
104
105             image = morphology.dilation(image, pattern1)
106             image = morphology.dilation(image, pattern2)
107
108             image = morphology.erosion(image, pattern3)
109             image = morphology.dilation(image, pattern3)
110
111             images.append(image.flatten())
112
113     return np.array(images)

```