CV HW1

Implementation

The first thing to do is to get the normal map of the object. This equation is given by TAs and solved by using psuedo-inverse described in the spec.

```
def get_normal_map(images, light_sources):
    kdn = np.dot(np.dot(np.linalg.inv(np.dot(light_sources.T, light_sources)), light_sources.T), images)
    norms = np.linalg.norm(kdn, axis=0).reshape(image_row*image_col, 1)
    kdn = kdn.T
    normal_map = kdn/norms
    return normal_map
```

After generating the normal map, some of value will be nan due to division by 0. I tried to fill in all the nan values with depth 0, but the matrix computation failed because of the memory limit and computing such a big matrix, so I created a mask to mask all the nan values.

```
def get_mask_from_normal_map(normal_map):
    mask = np.zeros((image_row, image_col))
    normal_map = np.reshape(normal_map, (image_row, image_col, 3))

    for i in range(image_row):
        for j in range(image_col):
            if np.isnan(normal_map[i][j][0]):
                mask[i][j] = 0
            else:
                mask[i][j] = 1
    return mask
```

We can create the matrix descibed in spec.

$$\begin{bmatrix} \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \cdots & -1 & 1 & \cdots & \cdots & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \cdots & -1 & \cdots & \cdots & 1 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix} \begin{bmatrix} \vdots \\ z_{50,50} \\ z_{51,50} \\ \vdots \\ z_{50,51} \\ \vdots \end{bmatrix} = \begin{bmatrix} \vdots \\ -\frac{n_x^{50,50}}{n_z^{50,50}} \\ \vdots \\ \vdots \\ -\frac{n_y^{50,50}}{n_z^{50,50}} \\ \vdots \end{bmatrix}$$

This part is filling in $-z_{a,b} + z_{a,b+1} = -n_x^{a,b} / n_z^{a,b}$, and I add a 0 when the next normal vector is masked (nothing in the next normal vector, so the depth should be 0).

```
for i in range(image_row):
    for j in range(image_col-1):
        if mask[i][j] == 0:
            continue
        temp = np.zeros((image_row * image_col))
        temp[i * image_col + j] = -1
        temp[i * image_col + j + 1] = 1
        used[i * image_col + j] = 1
        used[i * image_col + j + 1] = 1
        M.append(temp)
        # clipping for extreme values
        V.append(min(max(- normal_map[i][j][0] / normal_map[i][j][2], -threshold), threshold))
        if mask[i][j+1] == 0:
            temp = np.zeros((image_row * image_col))
            temp[i * image_col + j+1] = 1
            used[i * image_col + j+1] = 1
            M.append(temp)
            V.append(0)
```

Then I did the same thing on the x axis except for the value of V. I filled in the value of V with a negative sign at first, and I found that it creates a bad image because it should be a positive value. I found this by disabling the y axis, and this creates a depth map that decreases toward the middle of the image.

The mask mentioned above reduces the row of the matrix, but it is still big. I reduced the column size by filtering out unused columns. The used array is updated during insertion of the matrix.

```
M = np.array(M, dtype=np.float32)
V = np.array(V, dtype=np.float32).reshape(-1, 1)
M = M.T
M = [M[:][j] for j in range(image_row*image_col) if used[j]]
M = np.array(M, dtype=np.float32)
M = M.T
z = np.dot(np.dot(np.linalg.inv(np.dot(M.T, M)), M.T), V)
```

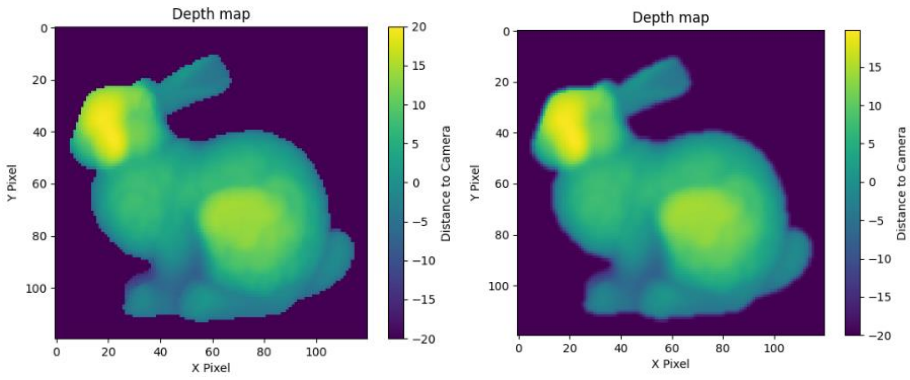
Finally, we can put all the z into the original image.

```
# fill in the values while rescale to (0, 40)
count = 0
temp = []
maximum = np.max(z)
minimum = np.min(z)
for i in range(image_row*image_col):
    if used[i] == 1:
        temp.append((z[count][0]- (minimum))*40/(maximum-minimum))
        count += 1
    else:
        temp.append(0.0)
return np.array(temp, dtype=np.float32)
```

Enhancement

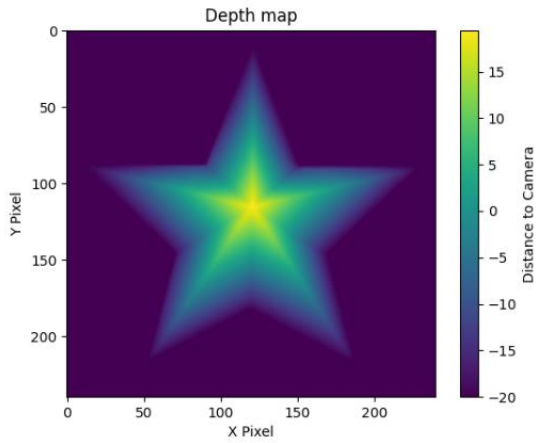
I added a Gaussian filter on the depth map, hoping to make the depth map a little smoother. It did make the edge smoother than the original one.

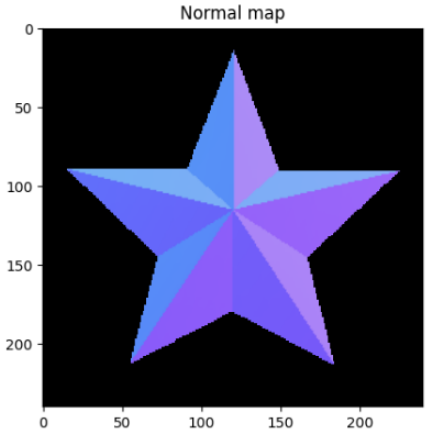
Result: before / after



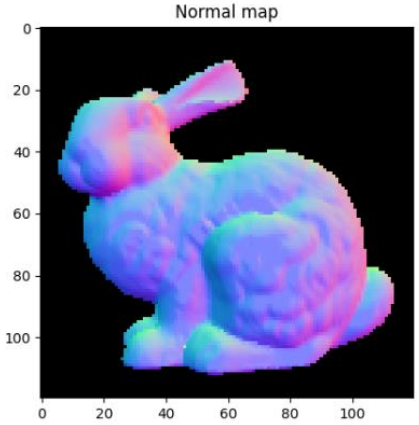
Normal map & depth map of “star” & “bunny”

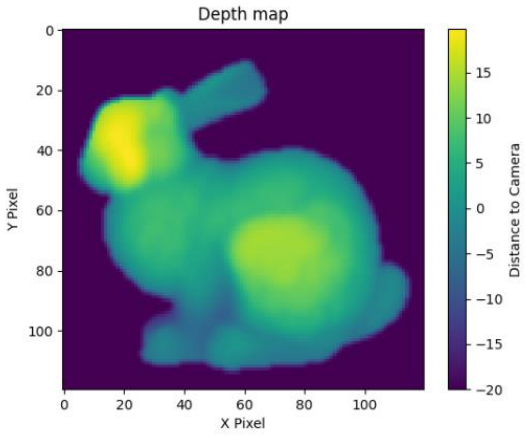
Star - normal map / depth map





Bunny – normal map / depth map

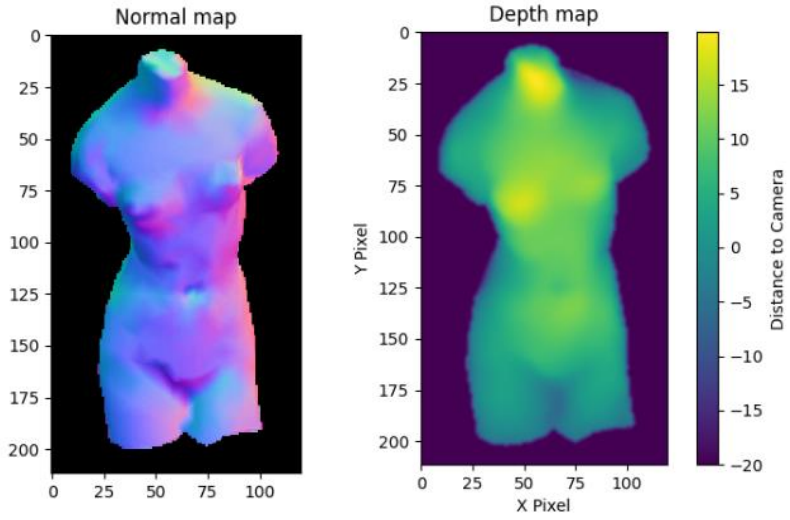




Surfaces of “venus”

There is an extreme value at somewhere near the neck of venus, making the head of venus have extremely high depth values. I solved this by clipping the values of matrix V. The value of the threshold can be tuned, and a value between 3 to 20 able to produce good results.

```
# clipping for extreme values
V.append(min(max(- normal_map[i][j][0] / normal_map[i][j][2], -threshold), threshold))
```



Surfaces of “noisy venus”

I added several tricks to reduce the noises.

I first add two filter for reducing the gaussain noise, and it turns out that the two filter have similar performance.

```
def preprocess_gaussian_filter(image):
    new_image = np.zeros(image.shape)
    # pad 3d np array
    padded = np.pad(image, 1, mode='edge')
    padded = padded.astype(np.float64)
    for i in range(1, padded.shape[0]-1):
        for j in range(1, padded.shape[1]-1):
            new_image[i-1, j-1] = np.sum(padded[i-1:i+2, j-1:j+2] * np.array([[1, 2, 1], [2, 4, 2], [1, 2, 1]])) / 16
    return new_image

def preprocess_low_pass(image):
    new_image = np.zeros(image.shape)
    padded = np.pad(image, 1, mode='edge')
    padded = padded.astype(np.float32)
    for i in range(1, padded.shape[0]-1):
        for j in range(1, padded.shape[1]-1):
            new_image[i-1, j-1] = np.sum(padded[i-1:i+2, j-1:j+2]) / 9
    return new_image
```

After sending all the images into the filters, I sum them up and check if it has small value, and I will set it to 0 if it is small. Here the value 15 is founded by trial and error.

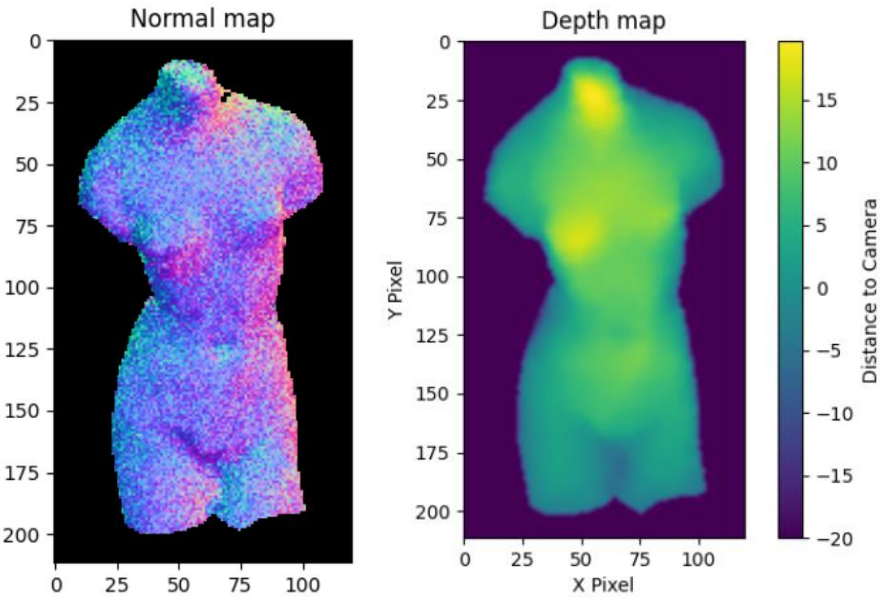
```
def mask_image_pixel(images):
    mask = np.zeros(images[0].shape)
    for image in images:
        mask += image
    mask = np.where(mask < len(images)*15, 0, 1)
    for image in images:
        image *= mask
    return images
```

After masking, I will create the normal map with the processed images. I observed that there are some points have not-nan values at the place where that should not be one. Here I do BFS from the center of the image and check if it is connected to the middle. This will mask out the small points outside of the body.

```
def mask_normal(image):
    mask = np.zeros(image.shape[:2])
    image = image.astype(np.float32)
    q = Queue()
    q.put((int(image.shape[0]/2), int(image.shape[1]/2)))
    while not q.empty():
        x, y = q.get()
        if mask[x, y] == 1:
            continue
        mask[x, y] = 1
        if x > 0 and not np.isnan(image[x-1, y][0]):
            q.put((x-1, y))
        if x < image.shape[0]-1 and not np.isnan(image[x+1, y][0]):
            q.put((x+1, y))
        if y > 0 and not np.isnan(image[x, y-1][0]):
            q.put((x, y-1))
        if y < image.shape[1]-1 and not np.isnan(image[x, y+1][0]):
            q.put((x, y+1))
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            if mask[i, j] == 0:
                image[i, j] = math.nan
    return image
```

After preserving only pixels connected to the middle point, I use a low pass filter to further reduce the noise because the normal map looks noisy, shown as the following picture. The depth map looks fine because I have another filter on depth map.

```
def normal_low_pass(image):
    new_img = np.zeros(image.shape)
    # only pad x and y axis, not the channel axis
    padded = np.pad(image, ((1, 1), (1, 1), (0, 0)), mode='edge')
    padded = padded.astype(np.float32)
    for i in range(1, padded.shape[0]-1):
        for j in range(1, padded.shape[1]-1):
            if np.isnan(padded[i, j][0]):
                new_img[i-1, j-1] = np.nan
                continue
            new_img[i-1, j-1] = np.nansum(padded[i-1:i+2, j-1:j+2], axis=(0, 1)) / np.sum(~np.isnan(padded[i-1:i+2, j-1:j+2]))
    return new_img
```



Result