MP1 - Report

Course: Computer Vision (CS543/ECE549)

Credit: 4

Instructor: Prof. Svetlana Lazebnik

TA: Arun Mallya, Zhicheng Yan

Name: Bo-Yu Chiang

UIN: 677629729

email: bchiang3@illinois.edu

Implemented solution

In Pre-processing part:

```
imarray = bsxfun(@minus, imarray, ambient_image);
imarray(imarray < 0) = 0;
imarray = imarray / 255;
```

I replace all the for-loop by bsxfun function. The quality is the same but the speed is 10 times faster.

In photometric_stereo.m :

```
I_matrix = reshape(imarray(:,:,1), 1, 32256);
for i = 2:64
    I_matrix = cat(1, I_matrix, reshape(imarray(:,:,i), 1, 32256));
end
```

Because original imarray's size is 192*168*3, which can not fit the requirement of backslash operator, so we have to transform it to 3*(192*168) in order to do a 2-D computation. My implementation is to append each (192*168) pixels line by line to construct I_matrix.

```
current_sum = bsxfun(@hypot, g(1,:,:), g(2,:,:));
current_sum = bsxfun(@hypot, current_sum, g(3,:,:));
albedo_image = squeeze(current_sum);
surface_normals(:, :, 1) = bsxfun(@rdivide, squeeze(g(1,:,:)), albedo_image);
surface_normals(:, :, 2) = bsxfun(@rdivide, squeeze(g(2,:,:)), albedo_image);
surface_normals(:, :, 3) = bsxfun(@rdivide, squeeze(g(3,:,:)), albedo_image);
```

I replace all the for-loop by bsxfun function. current_sum's size is 1*192*168, so I use squeeze function to remove singleton dimension. Therefore, albedo_image is the magnitude of g. Then I use bsxfun to compute surface_normals respectively.

In get_surface.m :

```
case 'column'
    cum_fx = cumsum(fx, 2);
    cum_fy = cumsum(fy, 1);
    height_map = cum_fx + repmat(cum_fy(:,1), 1, 168);
case 'row'
    cum_fx = cumsum(fx, 2);
    cum_fy = cumsum(fy, 1);
    height_map = cum_fy + repmat(cum_fx(1,:), 192, 1);
```

I compute the cumulative sum of the element along first dimension of fy and cumulative sum of the element along second dimension of fx in advance. In column approach, the height_map(i, j) is equal to cum_fx(i, j) + cum_fy(i, 1), so I repeat the first column of cum_fy 168 times. This trick speed my implementation 7 times.

Speed up trick

In Pre-processing part:

Oringinal:

```
for j = 1:64
    imarray(:,:,j) = imarray(:,:,j)-ambient_image;
end


for i = 1:192
    for j = 1:168
        for k = 1:64
            if(imarray(i,j,k) < 0)
                imarray(i,j,k) = 0;
            end
            imarray(i,j,k) = imarray(i,j,k)/255;
        end
    end
end
```

Elapsed time is 0.267124 seconds.

New:

```
imarray = bsxfun(@minus, imarray, ambient_image);
imarray(imarray < 0) = 0;
imarray = imarray / 255;
```

Elapsed time is 0.029358 seconds.

=>It is 9 times faster than previous one.

In photometric_stereo.m :

Oringinal:

```
for i = 1:192
    for j = 1:168
        albedo_image(i, j) = sqrt(g(1, i, j)*g(1, i, j) + g(2, i, j)*g(2, i, j) + g(3, i, j)*g(3, i, j));
        surface_normals(i, j, 1) = g(1, i, j)/ albedo_image(i, j);
        surface_normals(i, j, 2) = g(2, i, j)/ albedo_image(i, j);
        surface_normals(i, j, 3) = g(3, i, j)/ albedo_image(i, j);
    end
end
```

Elapsed time is 0.006216 seconds.

New:

```
current_sum = zeros(1, 192, 168);
current_sum = bsxfun(@hypot, g(1,:,:), g(2,:,:));
current_sum = bsxfun(@hypot, current_sum, g(3,:,:));
albedo_image = squeeze(current_sum);
surface_normals(:, :, 1) = bsxfun(@rdivide, squeeze(g(1,:,:)), albedo_image);
surface_normals(:, :, 2) = bsxfun(@rdivide, squeeze(g(2,:,:)), albedo_image);
surface_normals(:, :, 3) = bsxfun(@rdivide, squeeze(g(3,:,:)), albedo_image);
```

Elapsed time is 0.003644 seconds.

=>It is 2 times faster than previous one.

In get_surface.m :

Oringinal:

```
for i = 1:192
    for j = 1:168
        fx(i, j) = surface_normals(i, j, 1)/surface_normals(i, j, 3);
        fy(i, j) = surface_normals(i, j, 2)/surface_normals(i, j, 3);
    end
end
```

Elapsed time is 0.003045 seconds.

New:

```
fx = squeeze(surface_normals(:,:,1)./surface_normals(:,:,3));
fy = squeeze(surface_normals(:,:,2)./surface_normals(:,:,3));
```

Elapsed time is 0.002717 seconds.

=>It is 1.1 times faster than previous one.

In method "row" :

Oringinal:

```
for i = 1:192
    for j = 1:168
        for k = 1:i
            height_map(i, j) = height_map(i, j) + fy(k, 1);
        end
        for k = 1:j
            height_map(i, j) = height_map(i, j) + fx(i, k);
        end
    end
end
```

Elapsed time is 0.376807 seconds.

New:

```
cum_fx = cumsum(fx, 2);
cum_fy = cumsum(fy, 1);
for i = 1:192
    for j = 1:168
        height_map(i, j) = cum_fy(i,1) + cum_fx(i, j);
    end
end
```

Elapsed time is 0.002144 seconds.

=>It is 180 times faster than previous one.

Fastest:

```
cum_fx = cumsum(fx, 2);
cum_fy = cumsum(fy, 1);
height_map = cum_fx + repmat(cum_fy(:,1), 1, 168);
```

Elapsed time is 0.000313 seconds.

=>It is 7 times faster than previous one.

In method "column" :

Oringinal:

```
for i = 1:192
    for j = 1:168
        for k = 1:j
            height_map(i, j) = height_map(i, j) + fx(1, k);
        end
        for k = 1:i
            height_map(i, j) = height_map(i, j) + fy(k, j);
        end
    end
end
```

Elapsed time is 0.407252 seconds.

New:

```
cum_fx = cumsum(fx, 2);
cum_fy = cumsum(fy, 1);
for i = 1:192
    for j = 1:168
        height_map(i, j) = cum_fx(1,j) + cum_fy(i, j);
    end
end
```

Elapsed time is 0.003802 seconds.

=>It is 107 times faster than previous one.

Fastest:

```
cum_fx = cumsum(fx, 2);
cum_fy = cumsum(fy, 1);
height_map = cum_fy + repmat(cum_fx(1,:), 192, 1);
```

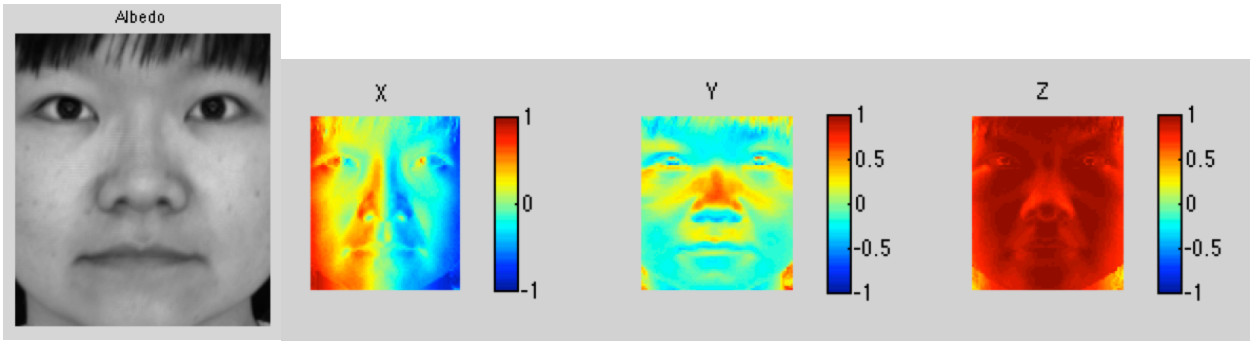
Elapsed time is 0.000399 seconds.

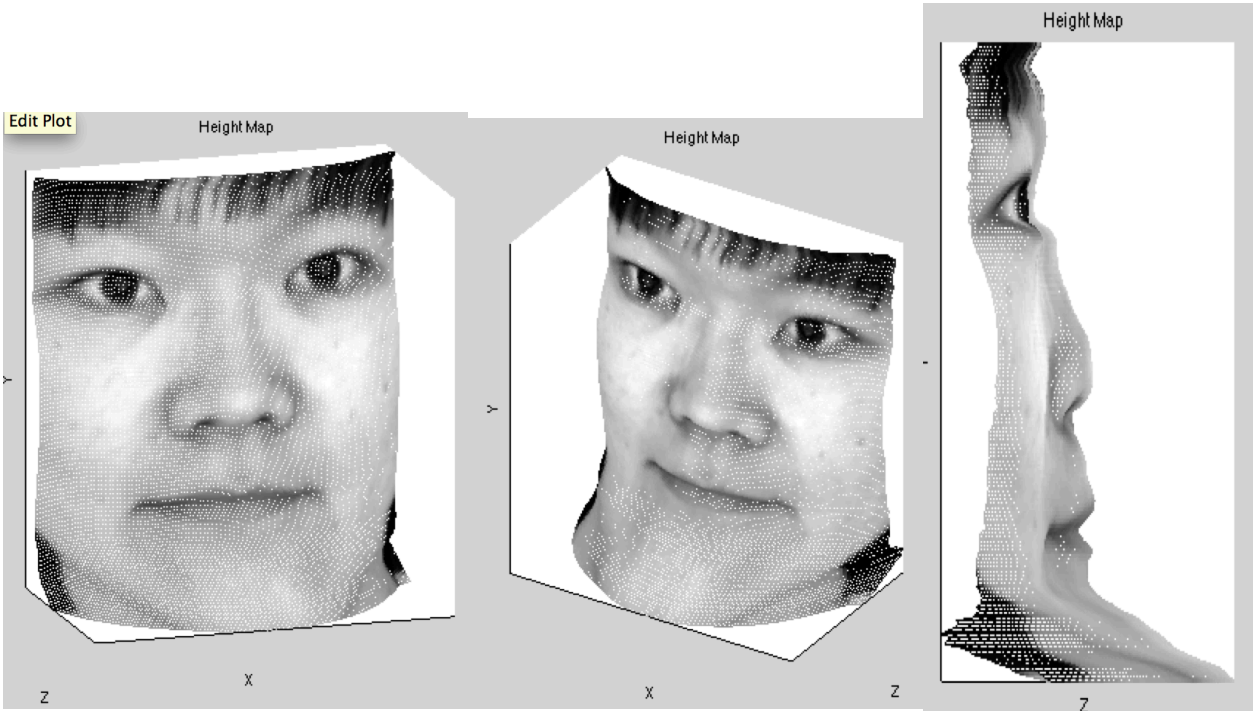
=>It is 9.5 times faster than previous one.

Discuss the differences between different integration methods

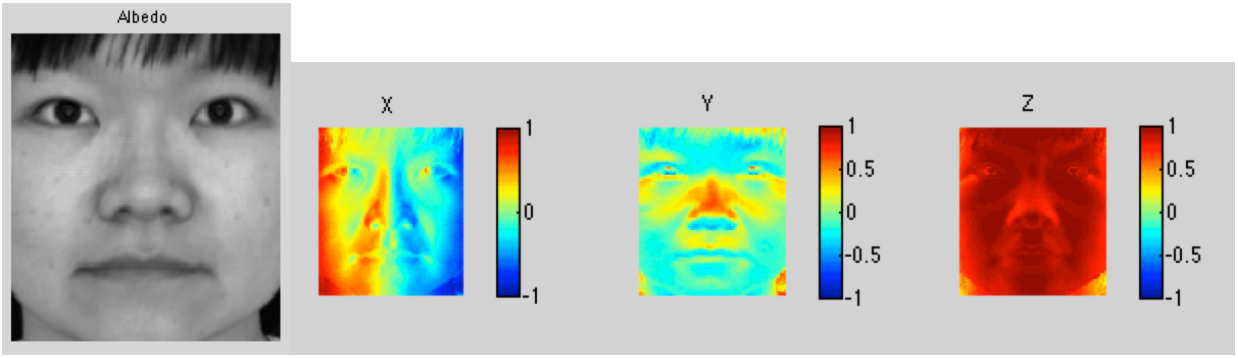
Subject to discuss: 'yaleB05'

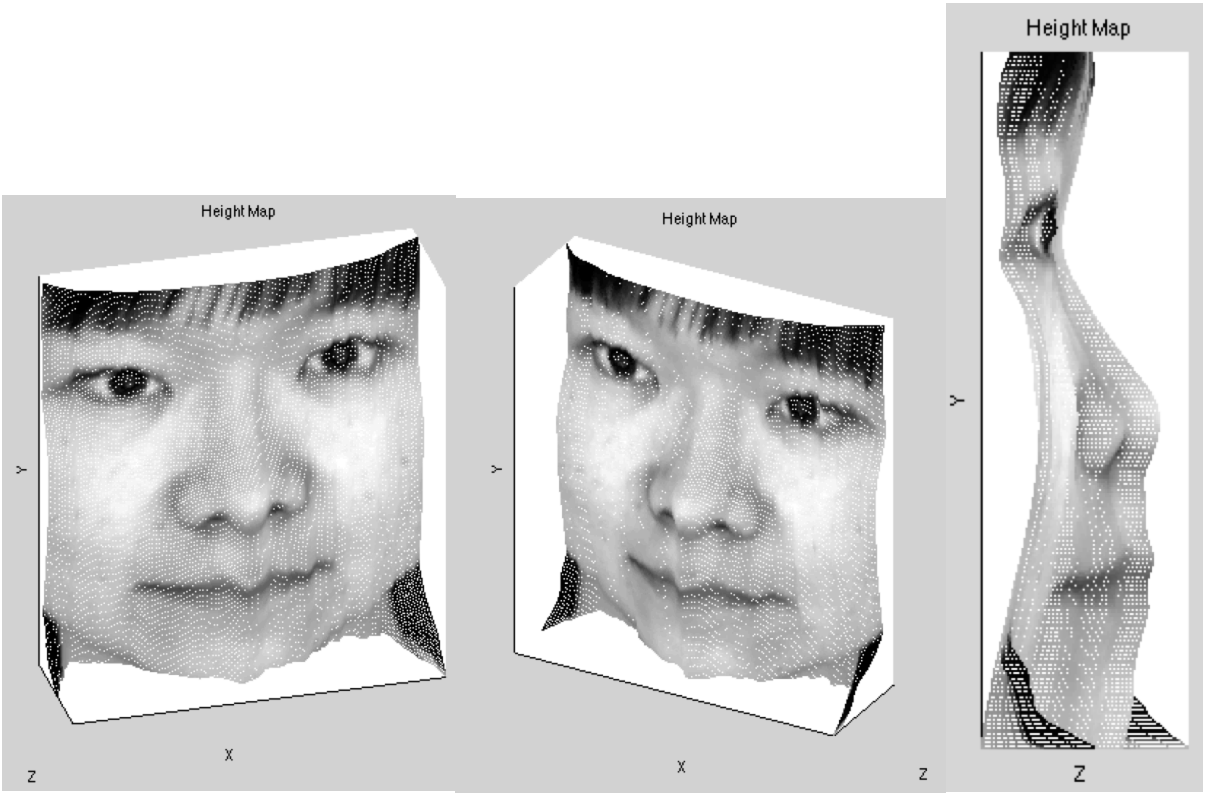
Column: Elapsed time is 1.031394 seconds.



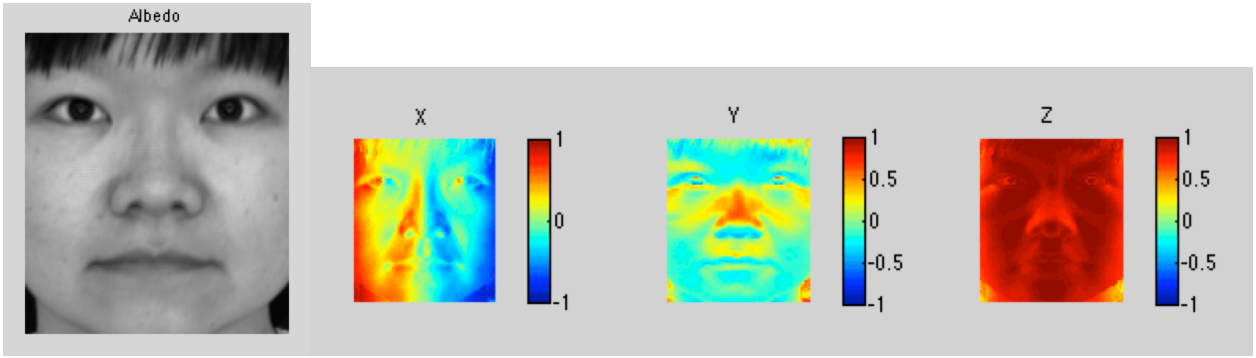


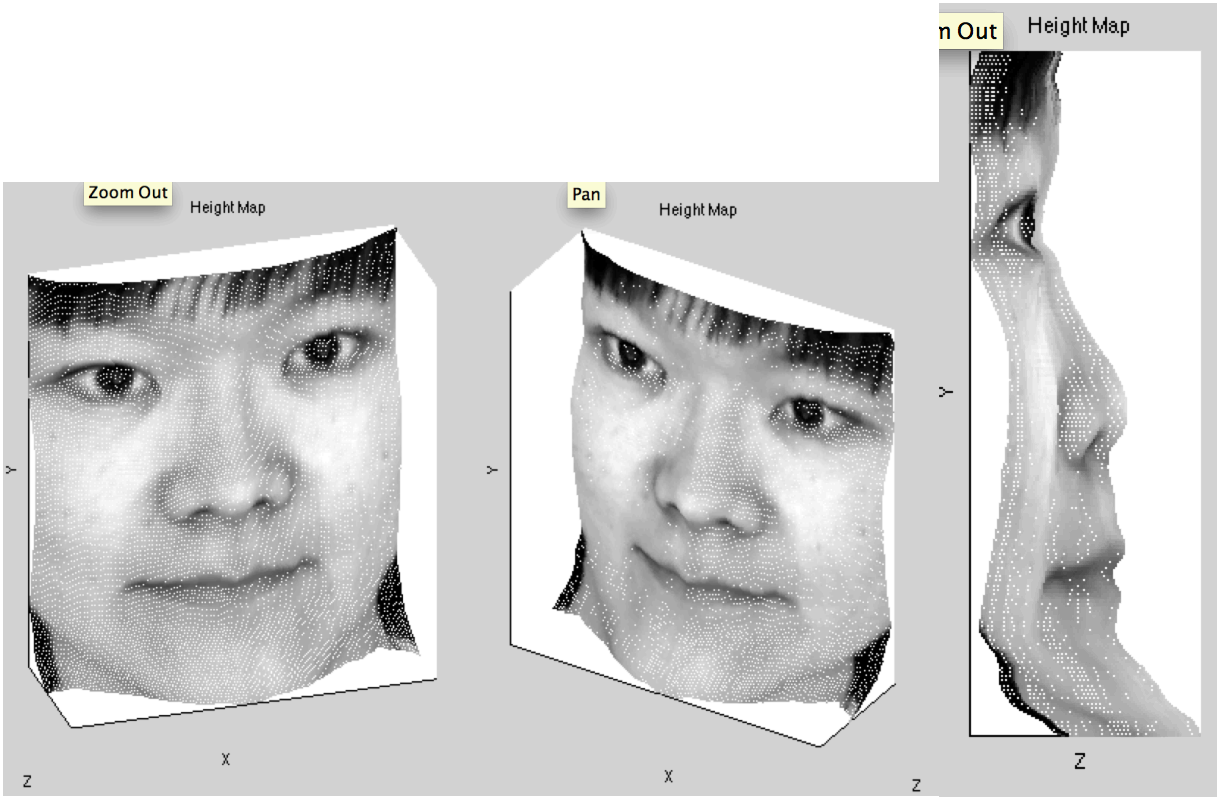
Row: Elapsed time is 1.121275 seconds.





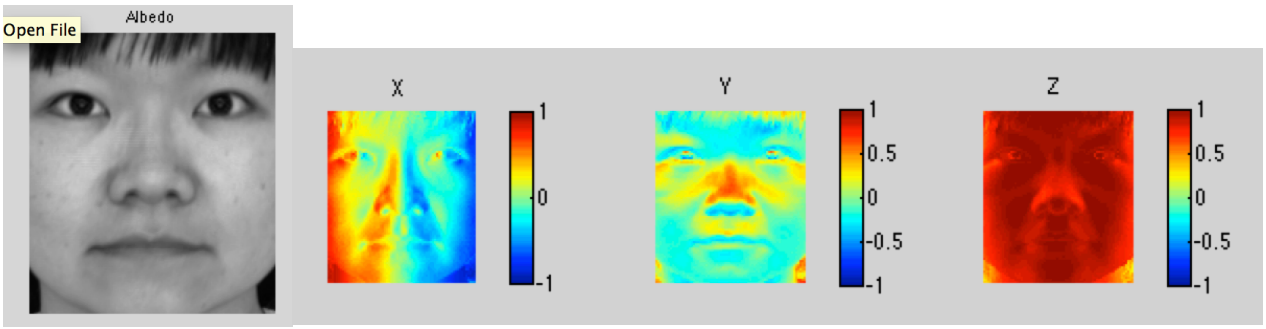
Average: Elapsed time is 1.462112 seconds.

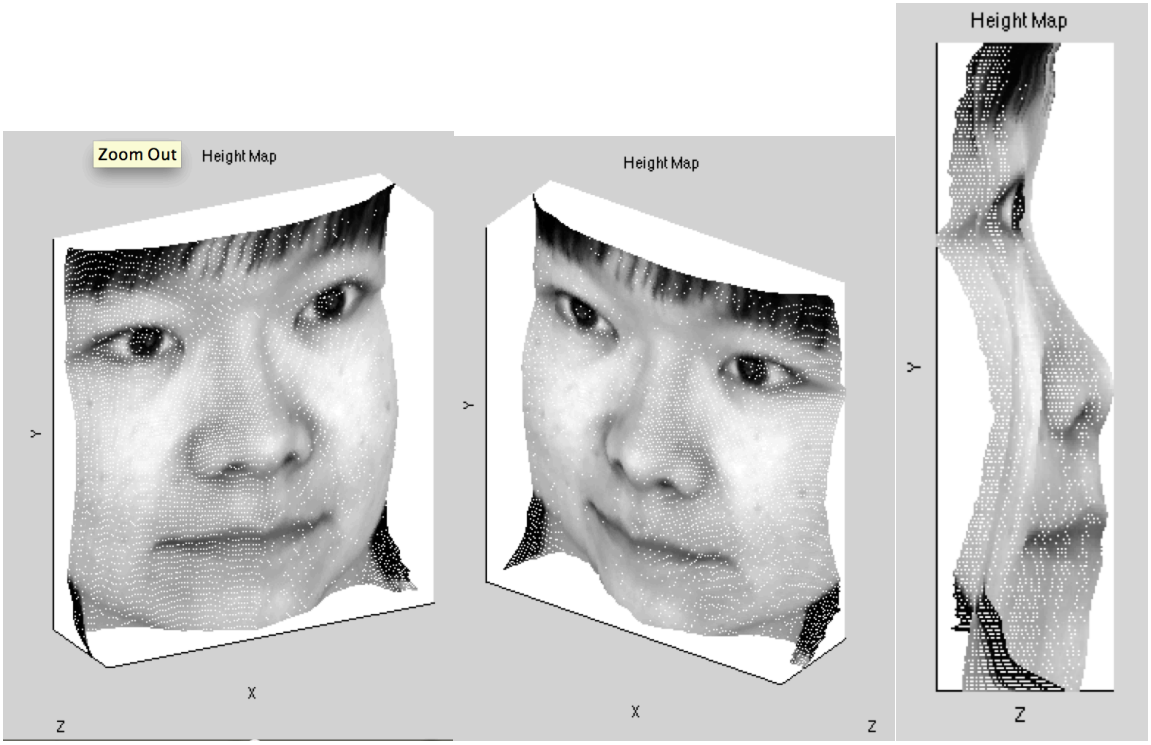




Random(100 times): Elapsed time is 77.153591 seconds for 100 times.

Elapsed time is 2.250512 seconds for 1 time.





Discuss:

I think the random method gives the best result.

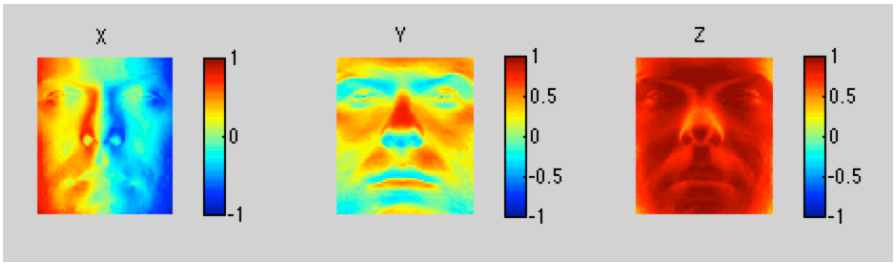
In column approach, we must go through the questionable area because we consider the background as a part of the face, which will lead a very large fy in left lower side. That is also the reason why the lower side of the height map is totally wrong.

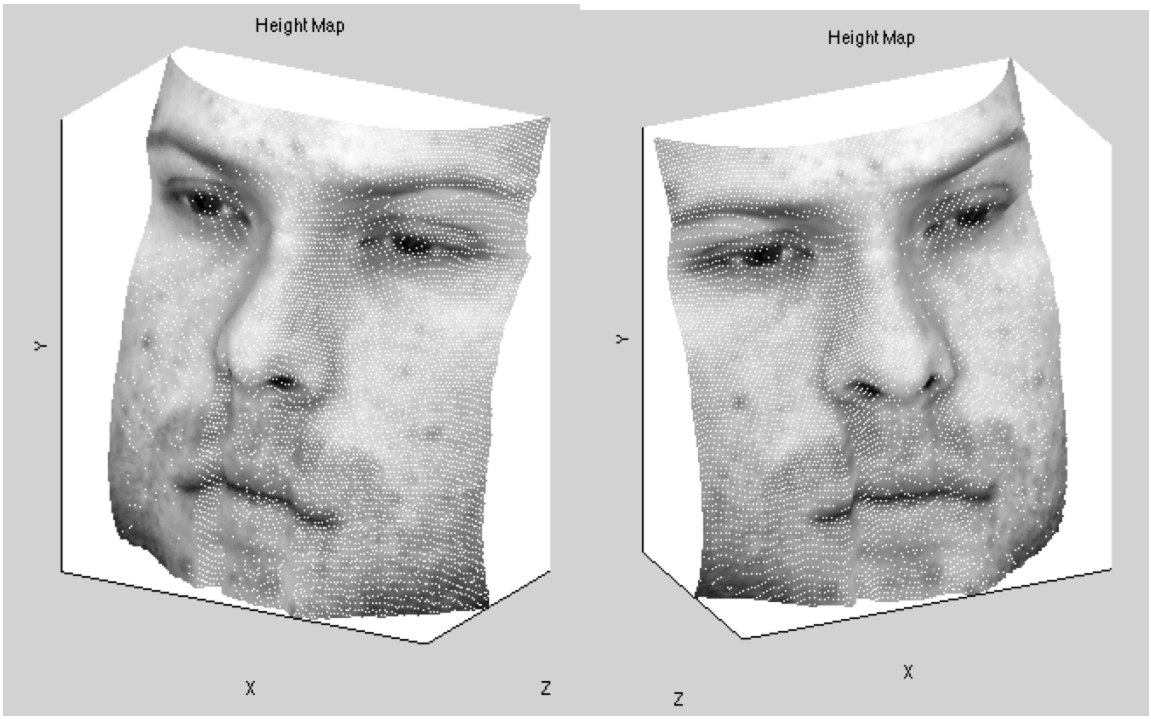
Because there is no absolutely correct path toward the destination point, so by averaging some different random paths we can reduce the variance. So I think random method is the best approach.

Albedo maps and height maps of every subject

Subject: 'yaleB01'

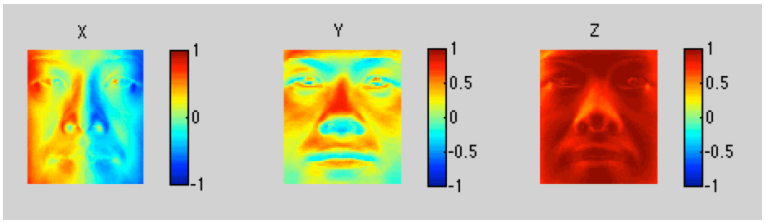
Random(100 times)

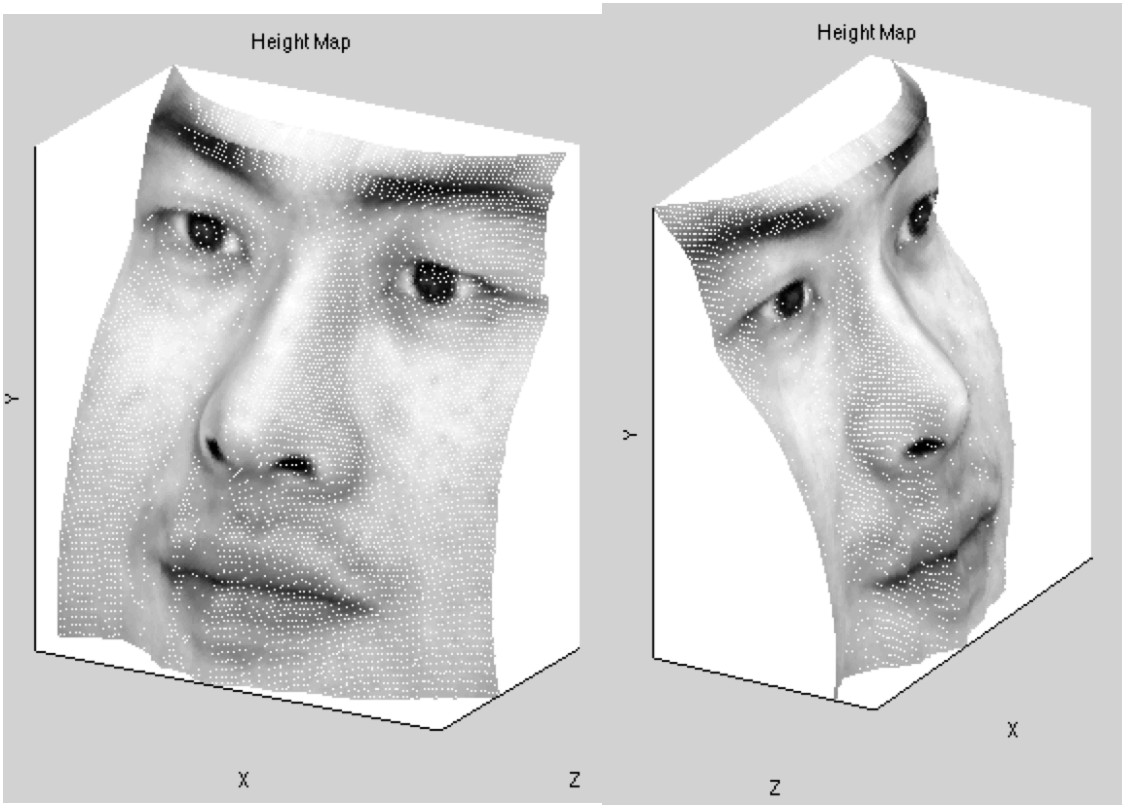


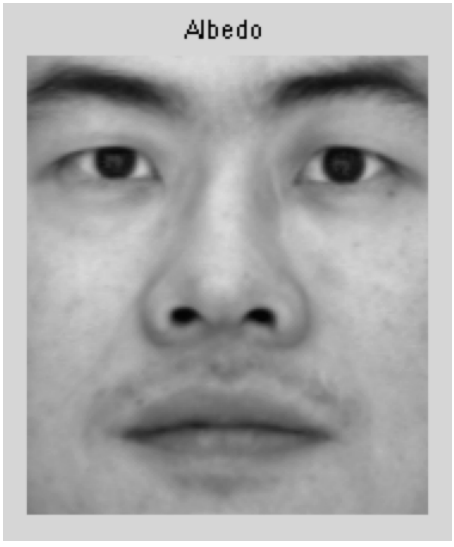


Subject: 'yaleB02'

Random(100 times)

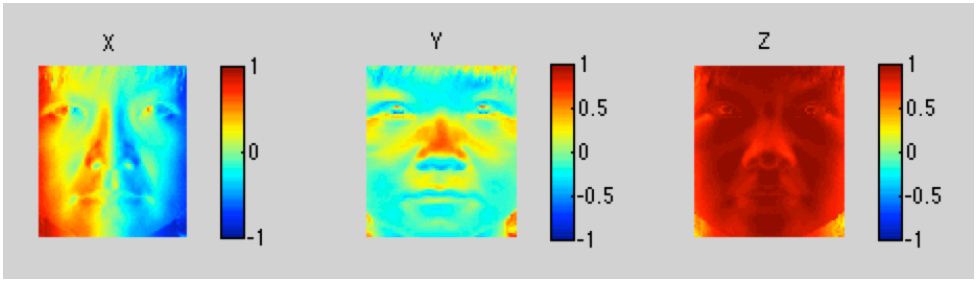


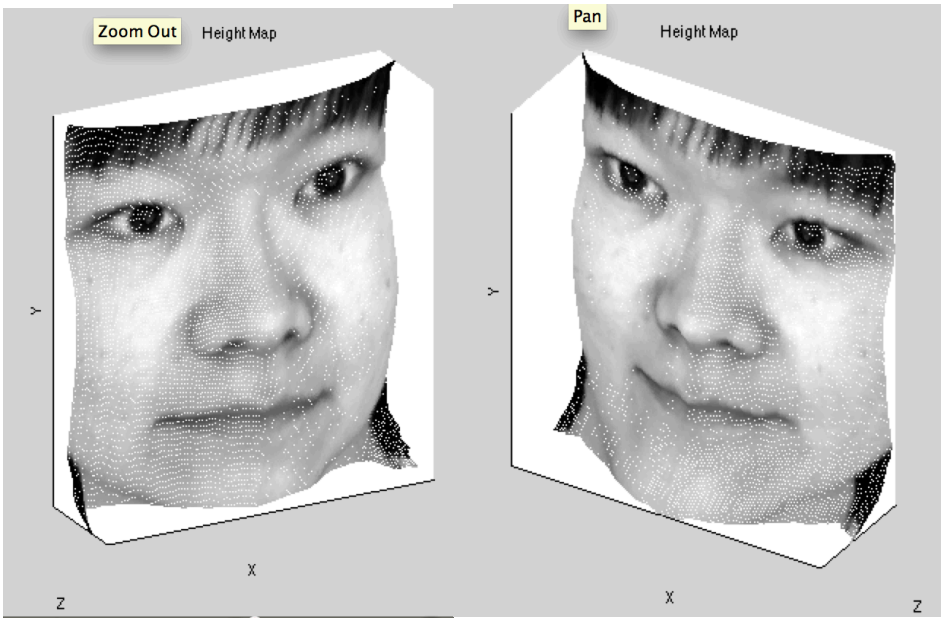


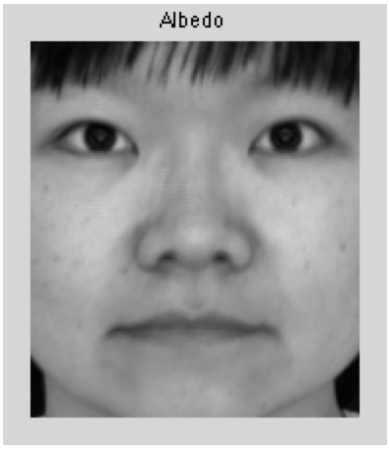


Subject: 'yaleB05'

Random(100 times)

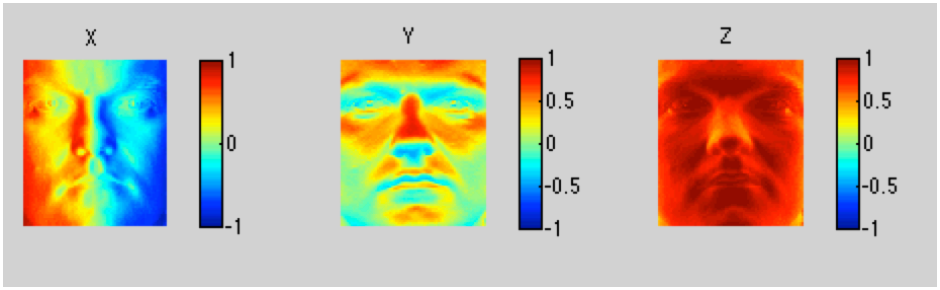


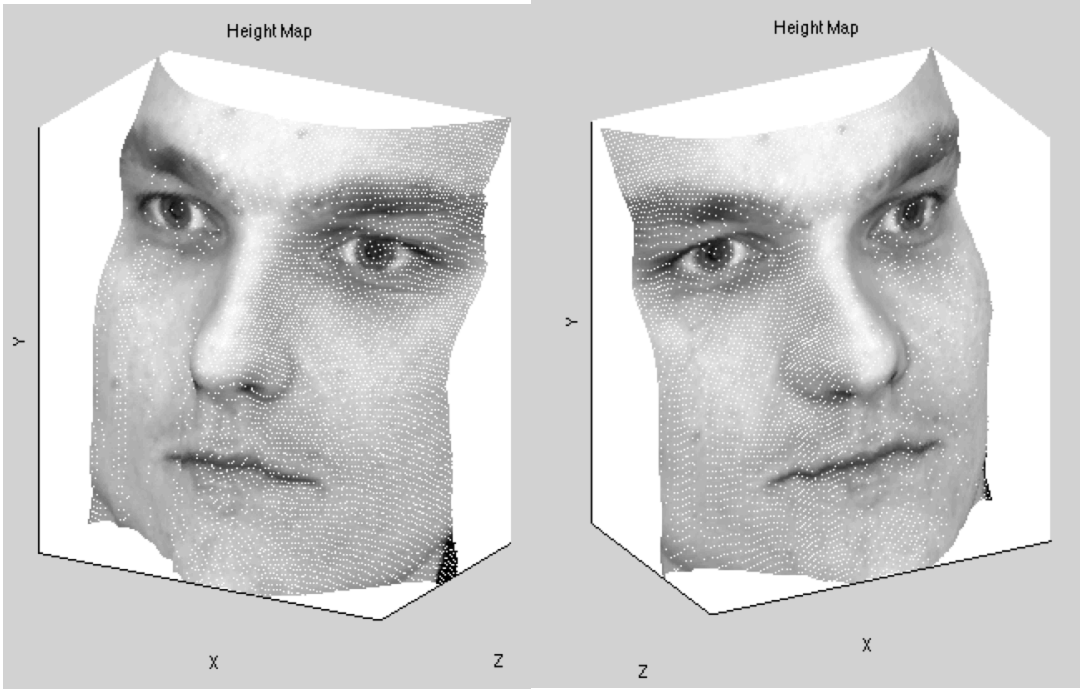




Subject: 'yaleB07'

Random(100 times)





Discuss how the Yale Face data violate the assumption of the method

The Yale Face data violate the following assumption that this method had made

1. A Lambertian object
   It is not realistic because in real world object, we still have to concern a lot of effect such as interreflection.
2. Local shading model
   In the Yale Face data, a lot of images have shadow which violate the local shading model
3. Orthographic projection
   The camera model which create the Yale Face data is not a orthographic camera model.

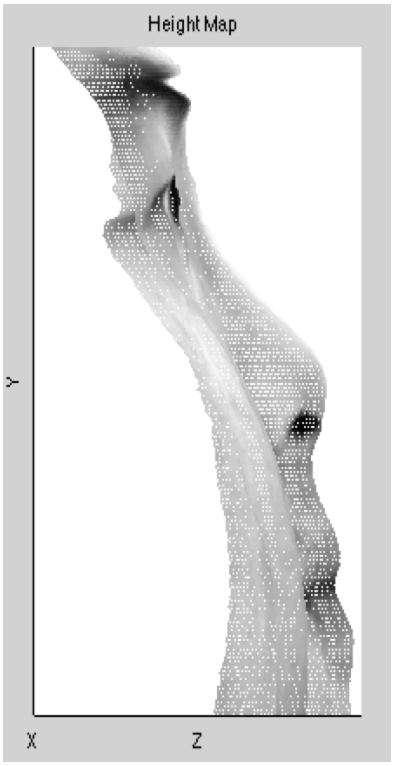
Select a subset that better match the assumption of the method

The first and third assumption is hard to restore in this assignment, so we can only reduce the violation of second one.

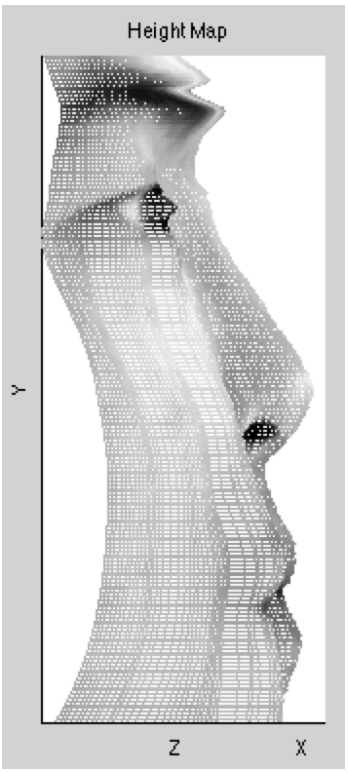
⇨ What we can do is to remove some training images with shadow.

I remove all the images with azimuth degree more than 70 and elevation degree more than 40 in subject `'yaleB02'`

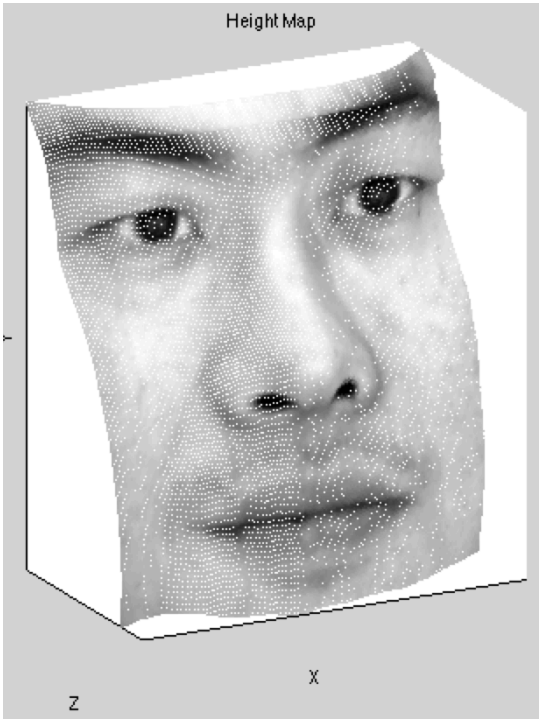
All 64 images



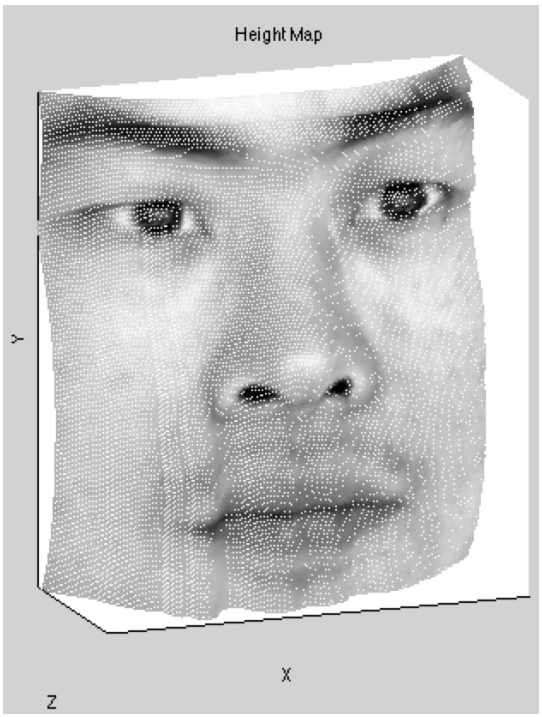
Selected 34 images



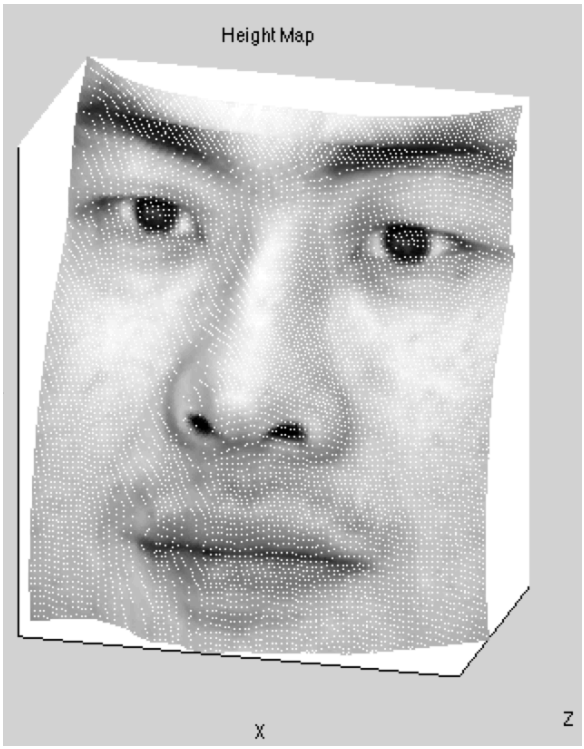
All 64 images



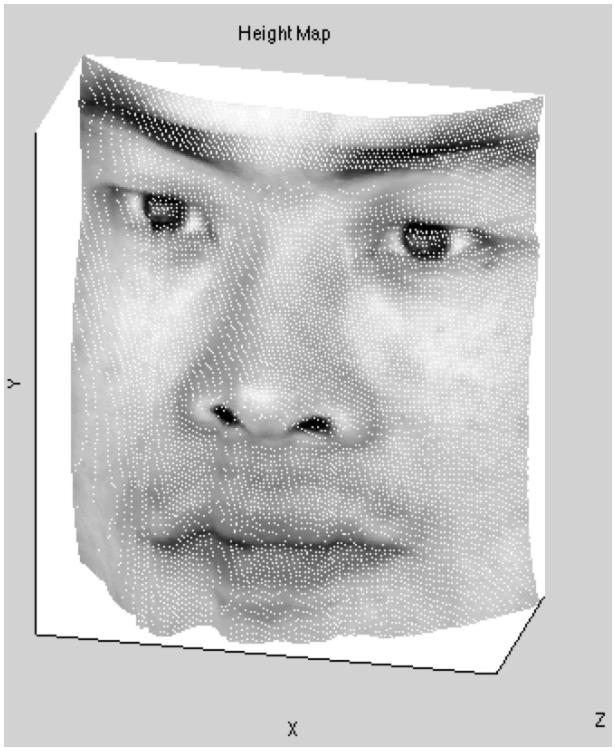
Selected 34 images



All 64 images



Selected 34 images



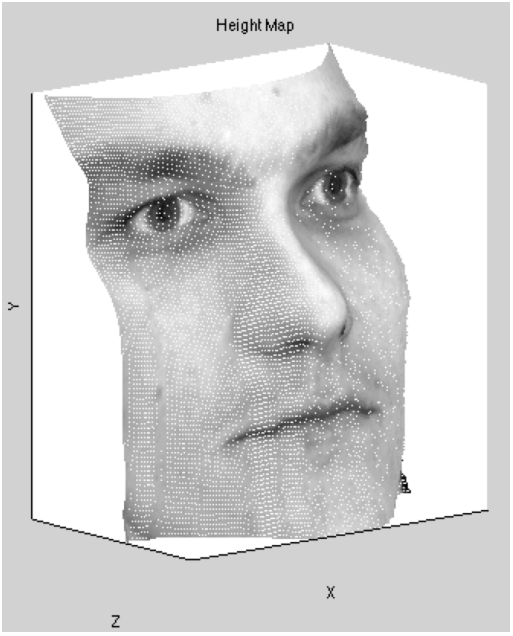
Extra credit: another approach to reconstruct height map by changing the reference point

We reconstruct our height map from the leftmost and uppermost point (1, 1) to any point we want ro reconstruct. But in my opinion, I think the error will propagate, which means the rightmost and lowermost point will have the largest error. In order to improve this, I started my integration from the middle of the image.

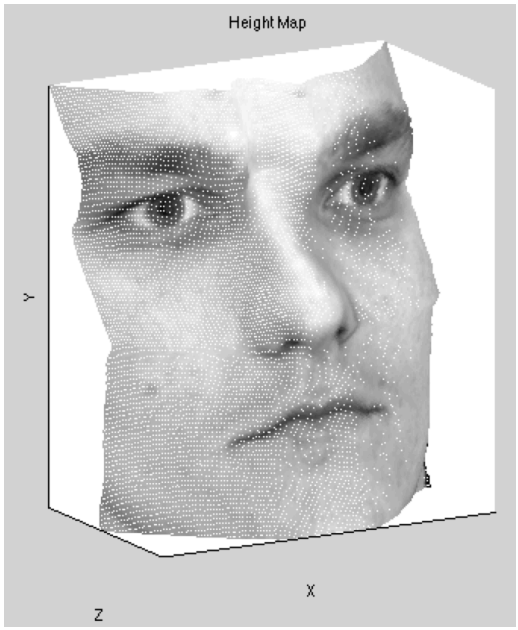
The implementation is not so hard; I just have to compute four different part of image with different operate method.

-fx -fy	+fx -fy
-fx +fy	+fx +fy

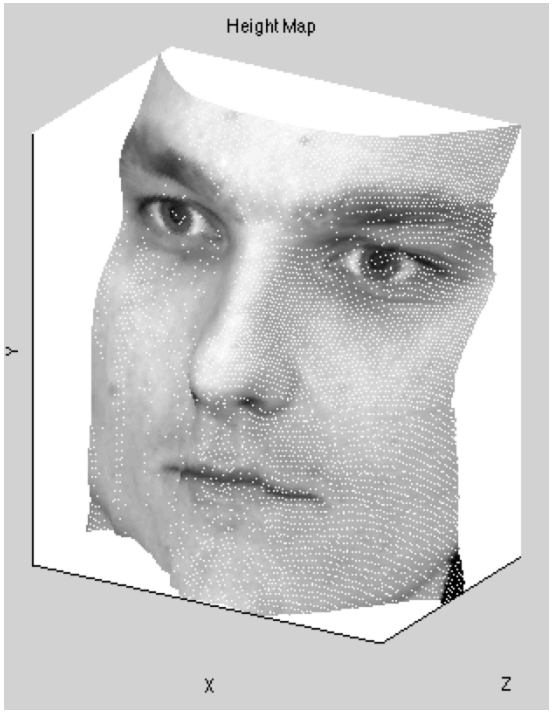
Original



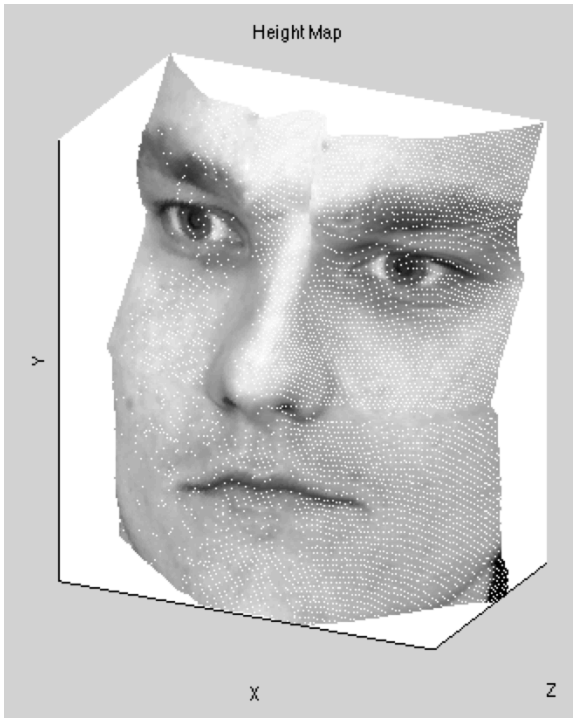
New approach



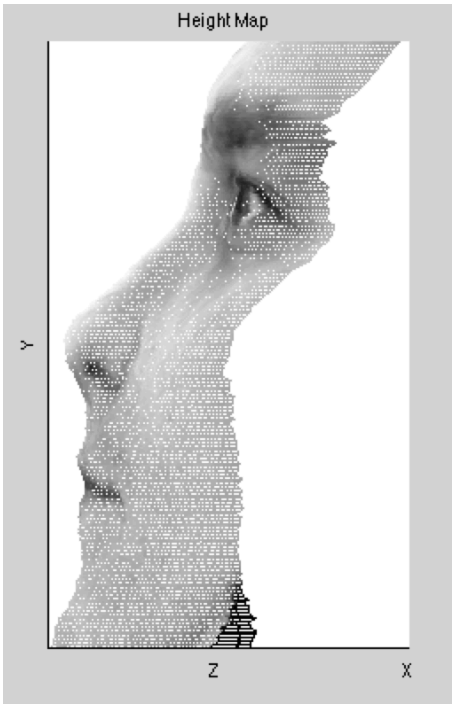
Original



New approach



Original



New approach