Mosaicing by SIFT

Report of lab excercise 3 of Image processing

Anouk Visser (6277209)
Merijn van Wouden (6306632)
University of Amsterdam
The Netherlands

December 4, 2011

Contents

1 Problem **3**

2 Usage **3**

3 Implementation, results and theory **3**

3.1 Introduction . . . 3

3.2 Projectivity . . . 3

3.2.1 . . . 3

3.3 . . . 4

3.3.1 . . . 4

3.3.2 . . . 4

3.3.3 . . . 4

3.3.4 . . . 4

3.3.5 . . . 4

3.3.6 . . . 4

3.3.7 . . . 5

3.3.8 . . . 5

3.3.9 . . . 5

3.3.10 . . . 5

3.3.11 . . . 5

3.4 . . . 6

4 Reflection **7**

5 Conclusion **7**

6 Appendix **8**

6.1 . . . 8

6.2 . . . 8

6.3 . . . 10

6.4 . . . 10

1 Problem

We want to recognize objects from images. For this we first need a representation of how the object should look like, and then find it in the target image. Hence we must somehow match parts of two representations to eachother. We will look at this in quite a closed world where we do include the possibility of still matching while there is a different rotation angle, scale, slope, illumination, noise and little differences in configuration. But in essence it is still clearly almost the same picture. We show how easily this can be done.

2 Usage

The functions shown in our report are included in the attachment, and can be executed from MATLAB.

3 Implementation, results and theory

3.1 Introduction

No questions were asked in this section (the introduction).

3.2 Projectivity

3.2.1

For creating projective transformation matrices we had written the function CreateProjectionMatrix before. This requires 4 correspondence points and makes an exact matrix. But the correspondence points that are used are not necessarily precise. In fact also in automated recognition we will have this problem. When we have more information (more points) we can optimize it. Though we can now not make an exact fit, because of the small differences between the point correspondence offset. So we need to make an optimal estimate with these. For this we use an SVD trick described in section A.3.4.2 in the book 'Multiple View Geometry in Computer Vision' by Hartley and Zisserman.
We changed the last part of CreateProjectionMatrix.m. See 6.1 in the Appendix for the function

For calling the newly changed createProjectionMatrix, the changed lines in demo_mosaic.m are:

```
[xy, xaya] = pickmatchingpoints(f1, f2, 10, 1) % (for instance 10 points)
```

```
T = maketform('projective',createProjectionMatrix(xy', xaya')');
```

After executing the program and having defined the specified amount of correspondence points we get a precise merge show in Figure 1.

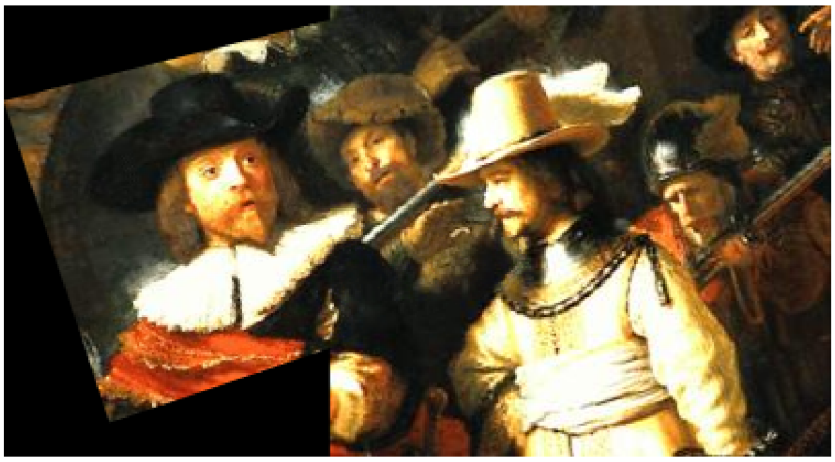

Figure 1: Two cropped parts of the 'Nachtwacht' re-merged together

3.3

3.3.1

Okay.

3.3.2

We can use SIFT in our mosaicing task to automatically detect structures that correspond. This means that we don't have to input the matching points, but that the program will find them itself.

3.3.3

For different scale spaces, we will find images that are blurred with different scales. We will find that blurring at a low scale gives us a lot of small details in the form of 'blobs'. Then, if we use a larger scale we will see that large objects in the image will turn into bigger blobs until everything merges together to just one blob. In these blobs we can find extremas. The moment an extremum disappears we have lost a few details in our image. Using scale spaces and its extrema, we will be able to detect an object.

3.3.4

The kernel of D is $G(x, y, k\sigma) - G(x, y, \sigma)$. So $\sigma^2\nabla^2 G \approx \frac{(\text{kernel of D})}{(k-1)}$

3.3.5

The trace of the matrix is the sum of the elements in the main diagonal. That is the diagonal that starts in the upper left corner. The Hessian matrix is an $n \times n$ matrix, this means we can simply find the *eigenvalues* by using the trace of the matrix.

3.3.6

Lowe uses some numbers and explains why he does: A frequency of 3 scaled per octave, a σ of 1.6, both because these make an optimal performance and not because these necessarily make it work just with these numbers. With finding the extrema in the sample points, when the offset is greater than 0.5 a different point is taken because that means another point is closer to the extremum, but this is not a magic number because this is in the math rules. A magic number here is when the extrema smaller than 0.03, which he chooses to be too small to be interesting (then the difference isn't big enough to make it interesting). An r of 20 is used in the principal curvature equation, without further explanation why. In the Hough transformations he uses 30 degrees for orientation, a factor of 2 for scale, and 0.25 times the maximum projected training image dimension to decrease the large error bounds.

3.3.7

Okay.

3.3.8

We will make a few substitutions to increase the readability of the equations.
$a = \frac{\partial D}{\partial x}$
$H = \frac{\partial^2 D}{\partial x^2}$

This gives us:
$D(x) = D + a^T x + \frac{1}{2} x^T H x$
$x' = (-H^{-1} a)$

We want to find $D(x')$, so:
$D(x') = D + a^T(-H^{-1}a) + \frac{1}{2}(-H^{-1}a)^T H(-H^{-1}a) =$
The transposed of a symmetric matrix is the same matrix. Since H is a symmetric matrix (hessian):
$D(x') = D + a^T(-H^{-1}a) - \frac{1}{2} a^T(-H^{-1}a) =$
$D(x') = D + \frac{1}{2} a^T(-H^{-1}a) =$
$D(x') = D + \frac{1}{2}\frac{\partial D^T}{\partial x} x'$
So, it was certainly not a typo.

3.3.9

We are looking in just one layer in the scale space, but we have three dimnesions because additional to the two orientation axes we have a magnitude of the vector, or the length. So we have three variables, hence we nee trilinear interpolation when we want to interpolate.

3.3.10

We can regard to 'Affine Transformations/changes' as a translation and or multiplication of an input. Affine changes in illumination therefore, mean that the illumination (intensity of the pixels) can shift or change by multiplication.

3.3.11

Okay.

*

SIFT in steps:

1. Take an image.
2. Create scale spaces meaning a stack of images made from this image by blurring it in every step a little bit more.
3. Find the keypoints which are all the points where the substraction of two subsequent layers in the scale space have an optimum. This substraction is called the Difference of Gaussians (DoG).
4. These points are with too many so we discard some. The ones with a low contrast (having a second order taylor expansion below some value, e.g. 0.03), and the ones that are too strong along the edges also. These have a big difference between the principal curvature along the edge and the one perpendicular to the edge. think of a very long blob compared to a more ball-shaped blob. The ball-shaped blob is better. The principal curvatures are the eigenvectors so we compare the eigenvalues. Now we have discarded enough keypoints.

5. Now we give the keypoints that we have left orientations based on the gradient around it. The largest gradient vector is pointed in a fixed direction and the others rotate with it, so that the keypoint is rotation invariant (not subject to errors by rotation). We know that the largest is pointing in a specific direction.

6. We still need to have illumination invariance, small configuration differences invariant and noise, etc. We take sets of pixels around our keypoint, and combine their gradients. We normalize these magnitudes so we have illumination invariance. When we perform feature matching we can see it has invariance to small affine differences.

*

Rotation invariant: Because we rotate the descriptor around the centre to align all largest gradients in the same direction.
Scale invariant: Because we normalize the magnitudes.

3.4

*

We based our implementation of RANSAC upon the pseudocode on Wikipedia. At first instance we were not sure about how the data that we would receive from SIFT was represented, so we made our own assumption, shown in the comments of the code. In a function that will follow later, we show how we preprocess the data to get it in the right shape for our own implementation.
The ransac.m code in 6.2 in the appendix contains enough comments to see what is happening. For more information on the RANSAC algorithm, the Wikipedia page is sufficient.
In 6.3 the error function used in ransac.m follows. We have to use the euclidean distance between the points, and because mostly error functions are squared(e.g. least squares) and making a better performance (a twice as big error is in practice more than twice as bad) we do this aswell.

Appendix 6.4 contains our preprocess function, reading the descriptors from SIFT, and apply vl_ubcmatch and preprocess them in a way to be processed by ransac. A part of the code, on bottom, is a demonstration that already uses this for the 'Nachtwacht' image parts, to automatically produce figure 2.

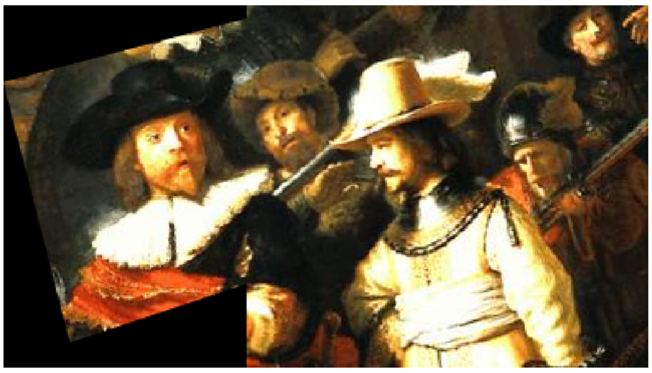

Figure 2: Again two cropped parts of the 'Nachtwacht' re-merged together, but now automatically

*

How can we determine the best amount of iterations k? If we take too little, the chance is too big we get an undesired result. If k is too big, we make this risk unnecessary low. If RANSAC selects just real inliers in an iteration from the model as maybe_inliers, the model will be the best result we will possibly

get. RANSAC will select maybe_inliers n times, so there are n of them. The chance of a maybe_inlier to be an actual inlier is then occuring n times. Giving us the chance of an optimal test set:
$P(\text{succeed for one iteration}) = (\frac{\#\text{actual\_inliers}}{\#\text{total\_points}})^n$
It's convenient if we now express this as the failure:
$P(\text{NO succeed for one iteration}) = 1 - (\frac{\#\text{actual\_inliers}}{\#\text{total\_points}})^n$
So we can give this the exponent k to know the failure chance after k iterations:
$P(\text{NO succeeds after k iterations}) = (1 - (\frac{\#\text{actual\_inliers}}{\#\text{total\_points}})^n)^k$
Now it depends on some factors to determine k. How much a problem is a suboptimal model? What is the amount of inliers and outliers? What is n?
Let's for example say that our model must be optimal with a chance of 0.9999 of succeeding so 0.001 chance of failure, the ratio is 0.5, and n is 5:
$(1 - (0.5)^5)^k = 0.0001$
$k = \frac{log(0.0001)}{log(1-(0.5)^5)} \approx 290$
And for big decreases in risk rate (e.g. factor 100), k grows relatively slow:
$k' = \frac{log(0.000001)}{log(1-(0.5)^5)} \approx 435$
$k'' = \frac{log(0.00000001)}{log(1-(0.5)^5)} \approx 580$
$k''' = \frac{log(0.0000000001)}{log(1-(0.5)^5)} \approx 725$

4 Reflection

It all went quite well and in the end we managed to do all the assignments we had to. With our RANSAC implementation it would be more efficient to first know the data representation, but this way it worked fine aswell.

5 Conclusion

We found that it's easier than expected to implement basic recognition and matching on images. The RANSAC algorithm works surprisingly well. After a lower than expected amount of iterations it makes a much better model than we can make with even 10 correspondence points by hand. This being based on random trials surprises us. But we are glad to have mastered this knowledge that was novel to us.

6 Appendix

6.1

```
function projMatrix = createProjectionMatrix(xy, uv)
%calculation of projection matrix

%get all x and y coordinates from xy
x = xy(:, 1);
y = xy(:, 2);

%get all x and y coordinates from uv
u = uv(:, 1);
v = uv(:, 2);

%initialize ones and zeros
o = ones(size(x));
z = zeros(size(x));

%construct matrix A
Aoddrows = [x, y, o, z, z, z, -u.*x, -u.*y, -u];
Aevenrows = [z, z, z, x, y, o, -v.*x, -v.*y, -v];

A = [Aoddrows; Aevenrows];

%estimate projection matrix
[U, D, V] = svd(A);
m = V(:, end);
projMatrix = reshape(m, 3, 3)';

end
```

6.2

```
function [ best_consensus_set, best_error, best_model ] = ransac(data, n, k, t, d)

% input:
%     data - a set of observations
%     model - a model that can be fitted to data
%     n - the minimum number of data required to fit the model
%     k - the number of iterations performed by the algorithm
%     t - a threshold value for determining when a datum fits a model
%     d - the number of close data values required to assert that a model
%     fits well to data
% output:
%     best_model - model parameters which best fit the data (or nil if no
%     good model is found)
%     best_consensus_set - data points from which this model has been estimated
%     best_error - the error of this model relative to the data

%how the data is represented in this function (after possible preprocessing)
%data = [x1,y1,u1,v1;
%        x2,y2,u2,v2;
%        x3,y3,u3,v3]

%initial best error value is infinite
best_error = Inf;

for iterations= 1:k

    %initialize errors
```

```
this_error = Inf;
errortotal = 0;

%split data in 2 random parts, first permute
datafrac = (data(randperm(size(data,1)),:));

%define input inliers/outliers and ouput inliers
%the inliers and outliers are split at the nth entry
%of our randomly sorted data
maybe_inliers = datafrac([1:n],:);
maybe_outliers = datafrac([(n+1):size(datafrac,1)],:);

maybe_inliers_in = maybe_inliers(:,[1 2]);
maybe_inliers_out = maybe_inliers(:,[3 4]);

maybe_outliers_in = maybe_outliers(:,[1 2]);
maybe_outliers_out = maybe_outliers(:,[3 4]);

%create model and update consensus set.
maybe_model = createProjectionMatrix(maybe_inliers_in, maybe_inliers_out);
consensus_set = maybe_inliers;

%preprocess the maybe_outlier input points
maybe_outliers_inT = homT(maybe_outliers_in);

%testset_chek contains output points based on the model and input
%points.
testset_check = maybe_model * maybe_outliers_inT;

%get rid of the extra homogeneous coordinate.
testset_check =
[testset_check(1,:)./testset_check(3,:);
 testset_check(2,:)./testset_check(3,:)];

%if the error is smaller than t for all elements calculated
%append the input/output points to the consensus set.
%keep track of the total error occuring using this consensus set.
for b = 1:size(testset_check,2)

    %find sum of squared errors
    error_outliers = error1(testset_check, maybe_outliers_out, b);

    if  error_outliers < t
        consensus_set =
        [consensus_set;maybe_outliers_in(b,:),maybe_outliers_out(b,:)];
    end

    errortotal = errortotal + error_outliers;

end

%if the size of the consensus set meets the required d
%also compute the error for the inliers.
if(size(consensus_set,1)>d)

    %split data in input- and output
    consensus_set_in = consensus_set(:,[1 2]);
    consensus_set_out = consensus_set(:,[3 4]);
    %find the projection matrix to optimally approximate in to out
    this_model =
    createProjectionMatrix(consensus_set_in, consensus_set_out);
```

```
        %preprocess the maybe_inlier input points
        inliers_in = homT(consensus_set_in);

        %apply the model to be tested on the possible inliers
        testset_check = maybe_model * inliers_in;

        %get rid of the homogeneous coordinate
        testset_check =
        [testset_check(1,:)./testset_check(3,:);
         testset_check(2,:)./testset_check(3,:)];

        %now test the error for the inliers and add to the previous error
        for b = (1:size(testset_check, 2))
            this_error = error1(testset_check, consensus_set_out, b);
        end

        %keep this error and this model as the best if it is (lowest error)
        if this_error < best_error
            best_model = this_model;
            best_consensus_set = consensus_set;
            best_error = this_error;
        end

    end

end

end
```

6.3

```
function [ error ] = error1(testset_check, maybe_outliers_out, b)
% square of the eucledian distance as an error
error = sqrt((testset_check(1,b) - maybe_outliers_out(b,1))^2 +
(testset_check(2,b) - maybe_outliers_out(b,2))^2)^2;
end
```

6.4

```
function [ points ] = preprocessDataRansac( )

%brings the image in the right format for sift
im1 = imread('nachtwacht1.jpg');
im1 = rgb2gray(im1);
im1 = single(im1);

im2 = imread('nachtwacht2.jpg');
im2 = rgb2gray(im2);
im2 = single(im2);

% finds the descriptors
[F1, D1] = vl_sift(im1);
[F2, D2] = vl_sift(im2);

% matches the descriptors
% elements in this 2xm matrix
% point to the column of vl_shift
% where further attributes can be found
M = vl_ubcmatch(D1, D2);
```

```
%produces a 4x1 vector: [x, y, something, something]
% x and y are of course the coordinates in the picture
F1 = vl_sift(im1);
F2 = vl_sift(im2);

% constructs two matrices. One with x,y coordinates of first
% picture, one with x,y coordinates of second picture
for i = 1:size(M, 2)
    Points1(i, :) = (F1([1 2], M(1, i)))';
    Points2(i, :) = (F2([1 2], M(2, i)))';
end

% constructs points that can be processed by ransac
points = [Points1, Points2];

% Here is a little extra:
% If you run this schript, you will see
% nachtwacht1 and nachtwacht2
% merged together without any need
% to specify the points yourself

B = ransac(points, 15, 200, 25, 10);

xyT = B(:, [1 2]);
xayaT = B(:, [3 4]);
xy = xyT';
xaya = xayaT';

% this is a modified version of the code of demo_mosaic
f1 = imread('nachtwacht1.jpg');
f2 = imread('nachtwacht2.jpg');

T = maketform('projective',createProjectionMatrix(xy', xaya')');
[x y] = tformfwd(T,[1 size(f1,2)], [1 size(f1,1)]);

xdata = [min(1,x(1)) max(size(f2,2),x(2))];
ydata = [min(1,y(1)) max(size(f2,1),y(2))];
f12 = imtransform(f1,T,'Xdata',xdata,'YData',ydata);
f22 = imtransform(f2, maketform('affine', [1 0 0; 0 1 0; 0 0 1]),
                  'Xdata',xdata,'YData',ydata);
subplot(1,1,1);
imshow(max(f12,f22));

end
```