Computer Vision Report – Image Stitching

Xuanming ZHANG
16521855
zy21855@nottingham.edu.cn

School of Computer Science
CS with AI
University of Nottingham Ningbo China

Contents

List of Figures

Chapter 1

Changes from the Proposal

Recall that in the initially written proposal, the proposed features of the program were as follows: 1) stitch any number of given images into one or more panoramas, depending on how many panoramas can be correctly recognized; 2) handle the input set of images in any given order (e.g. images in the left to right order or images in random order); 3) apply Pyramid Blending methods so that two matching images can be blended naturally to create a mosaic.

However, there are multiple changes while implementing this image stitching program. First of all, the whole implementation of stitching the input set of images was conducted in a left-to-right manner, where a pair of images was stitched based on the image order from left to right in each iteration of stitching. Therefore, we did not consider the case, where the input images can be stitched into multiple panoramas (i.e. no recognizing panoramas). Secondly, we added an additional module for image resizing. This can significantly speed up the computation given that the input images may be too large to be processed. Moreover, *Gain Compensation* was included as an option for users in order to render a more smooth panorama. Furthermore, users are also allowed to visualize the inliers while stitching two images in each iteration. Last but not the least, instead of just using *Pyramid Blending*, we implemented several other blending techniques, including *Average Blending*, *Alpha Blending*, and *Feathering Blending*(i.e. ramping).

Chapter 2

Detailed Steps and Computer Vision Techniques Adopted

In this chapter, the steps included in our implementation and particular Computer Vision techniques deployed will be discussed in great details.

2.1 Detailed Steps in Our Implementation

There are in total four steps in our design of implementation: 1) Argument Parsing; 2) Image Resizing; 3) Image Stitching and 4) Output Display. Each of these steps will be illustrated in details.

2.1.1 Argument Parsing

We adopted `argparse` as the main package for parsing the user command. Specifically, we customized the argument options using the code snippet shown in Figure 2.1. As you can see, five options, including `--images`, `--blending`, `--scalePercent`, `--gainCompensation` and `--drawInliers`, are provided for users to run the program. In particular, `--images` enables users to input a sequence of images to be stitched (i.e. the sequence of names of images)[1]; `--blending` allows users to select the blending techniques for blending the overlapping regions between two images; `--scalePercent` specifies the

[1]Note that the range of the number of images to be stitched is between 3 and 5

scale percentage for resizing the images; `--gainCompensation` signifies the deployment of *Gain Compensation* on the final panorama and `--drawInliers` enables the visualization of the intermediate inliers between two images in each iteration of stitching.

```
parser = argparse.ArgumentParser(description="Awesome Image Stitcher")
parser.add_argument("-i", "--images", help="a sequence of images to be stitched", nargs="+",
                    action=required_length(3, 5))
parser.add_argument("-b", "--blending", help="blending techniques to be selected for the final panaroma",
                    choices=['avg', 'alpha', 'feathering', 'pyramid'])
parser.add_argument("-sp", "--scalePercent", help="scale percentage for resizing the images", type=int,
                    choices=[10, 20, 30, 40, 50, 60, 70, 80, 90, 100], default=100)
parser.add_argument("-gc", "--gainCompensation", help="adopts gain compensation for the final output image",
                    action="store_true")
parser.add_argument("--drawInliers", help="draw the inliers while creating the panaroma and store the results",
                    action="store_true")
```

Figure 2.1: Argparse argument customization

The argument parsing step is conducted in `main.py` file. We first check if users input any sequence of images. If not, the program is terminated and the error message will be prompted. Also, we check if users specify any blending techniques. This is required because we want to blend the overlapping region according to the selected method. Then, we sort the input name sequence of images so that they are in increasing order. This is due to the fact that we implemented the stitching procedure in a left-to-right manner. An example user command to run the program is shown below. This command will stitch image sequence, 01.png, 02.png and 03.png, into a panorama using *Alpha Blending* and *Gain Compensation*. Besides, each of the input image is resized to 30% from the original image to speed up the computations.

```
$ python main.py -i 01.png 02.png 03.png -b alpha -sp 30 -gc
```

2.1.2 Image Resizing

The resizing step for the input set of images is mainly used to speed up the computation. Initially, the input images may be too large and they require much longer time to be processed by the program. In particular, we allow users to resize the input images to a customized scale percentage[2]. The function for resizing is shown in Figure 2.2. The *scale_percent* is passed in as an argument in the command line followed by `-sp` option.

[2]Users are allowed to choose from these scales: 10, 20, 30, 40, 50, 60, 70, 80, 90, 100

```python
def resize_img(img_list, scale_percent):
    """
    resize the images in the img_list
    @param img_list: a list containing input images
    @param scale_percent: the scale percentage to which the images are scaled from the original images
    @return: a new list of resized images
    """
    output = []
    for img in img_list:
        # calculate the percent of original dimensions
        width = int(img.shape[1] * scale_percent / 100)
        height = int(img.shape[0] * scale_percent / 100)

        # dsize
        dsize = (width, height)

        # resize image
        new_img = cv2.resize(img, dsize)

        output.append(new_img)

    return output
```

Figure 2.2: Resizing images

2.1.3 Image Stitching

This is the main function of the program. Specifically, the function takes as input: 1) the resized sequence of images; 2) selected blending technique; 3) flag variable for gain compensation and 4) flag variable for drawing the intermediate inliers between two images. The `image_stitching` function is defined in `stitching.py` file and the signature of the function is demonstrated in Figure 2.3. The primary logic of the function lies in the left-to-right stitching pattern. To begin with, the function takes the first two images in the image sequence as the candidate to be stitched together. Then, the leftmost image is marked as the source image and the other is marked as the target image (or dst image in the code implementation). The reason for this design is that in our implementation we always warp the source image so that it can be transformed to the target image in the final output. The general pipeline for stitching the source and target images is as follows:

- **Step 1**: Detect SIFT points for both images.
- **Step 2**: Find matching points among the found SIFT points for both images.
- **Step 3**: Estimate homography based on the matches (Note that the estimated homography transforms the source image to the target image).
- **Step 4**: Compute the offset along x and y direction. The reason for this step is that after transforming the source image to the target image, some parts of the

transformed source image may not be visible in the output because of the resulting negative x and y axis (Note that only the portions with positive x and y axis values can be displayed in the final output). Consequently, after the transformation, the target image needs to be translated according to the calculated offsets along x and y direction.

- **Step 5**: Draw the intermediate inliers between transformed source image and target image only if `--drawInliers` is specified by users.
- **Step 6**: For each matched point on the target image, add the offsets calculated from step 4. This works as translating the target image to the new location according to the offset information.
- **Step 7**: Compute new homography using the new set of coordinates of the target image.
- **Step 8**: Warp the source image according to the new homography calculated in step 7 using *RANSAC* method.
- **Step 9**: Perform gain compensation if requested by users. This is specified by using the `-gc` or `--gainCompensation` argument.
- **Step 10**: Add the translated target image to the final output and blend the overlapping region according to the selected blending method. The code snippet for this step is shown in Figure 2.4. The string `blending` is assigned by users.
- **Step 11**: Repeat Step 1 - 10 if there are images remained to be stitched with the current output. Note that the current output is regarded the source image while the next candidate image to be stitched is the target image.

```
def image_stitching(img_list, blending, gain_comp, drawInliers):
    """
    main function for image stitching
    @param img_list: list of images to be stitched
    @param blending: blending technique selected
    @param gain_compensation: flag indicating whether to adopt gain compensation
    @param drawInliers: flag for drawInliers
    @return: the final stitched image
    """
```

Figure 2.3: Image Stitching main function signature

```
############################################################
# Step 10: Add the translated right image to the final image
# and blend the overlapping region according to the selected
# blending method
############################################################
if blending == 'avg':  # average blending
    img_src = avg_blending(img_final, img_dst, offset_x, offset_y)
elif blending == 'alpha':  # alpha blending
    img_src = alpha_blending(img_final, img_dst, offset_x, offset_y)
elif blending == 'feathering':  # feathering blending
    img_src = feathering(img_final, img_dst, offset_x, offset_y)
elif blending == 'pyramid':  # pyramid blending
    img_src = pyramid_blending(img_src, newM, img_final, img_dst, offset_x, offset_y,
                               max(src_up_right_transform[0], src_down_right_transform[0]))
```

Figure 2.4: Step 10 of Image Stitching

2.1.4 Output Display

The final output panorama will be displayed onto the screen. The code snippet for this step is shown in Figure 2.5. In particular, the variable `result` will contain the final stitched panorama.

```
print('Done stitching!')
plt.figure()
plt.imshow(result)
plt.xticks([]), plt.yticks([])
plt.show()
```

Figure 2.5: Display the output panorama to the screen

2.2 Computer Vision Techniques

In this project, several well-known Computer Vision (CV) techniques were deployed in order to properly render the stitching results (i.e. the final panorama). In this section, all these CV approaches will be articulated in great details.

2.2.1 SIFT Points Detection

Proposed in [1], **S**cale **I**nvariant **F**eature **T**ransform (SIFT) serves as a powerful technique to extract feature points from input images that maintain invariance to scale, translation, rotation and illumination. In particular, we utilized the SIFT implementation from

OpenCV library to detect all SIFT points given two images (i.e. source and target image). The code snippet for detecting SIFT points is shown in Figure 2.6.

```
#####################################################
# Step 1: Detect SIFT points for each of the images
#####################################################
# Initiate SIFT detector
sift = cv2.xfeatures2d.SIFT_create()

# find the keypoints and descriptors with SIFT
kp1, des1 = sift.detectAndCompute(img_src, None)
kp2, des2 = sift.detectAndCompute(img_dst, None)
```

Figure 2.6: SIFT points detection

2.2.2 Finding Matching Points among Two Images

Given the SIFT points computed using `detectAndCompute` function for each image, we try to find the matches among those SIFT points between two images. Besides, due to the fact that some of the matches are not reasonable (i.e. bad matches), all the bad matches are filtered according to *Lowe's ratio test* [1].

2.2.3 Homography Estimation using RANSAC

In order to perform the transformation to the source image so that it can be stitched with the target image, the transformation matrix (i.e. homography) needs to be derived or estimated. In particular, we make use of all the good SIFT point matches to estimate the homography.

Besides, the RANSAC (i.e. Random Sample Consensus) algorithm is deployed to derive the homography. The procedure of RANSAC runs as follows:

- Loop for the following three steps to estimate homography
  - **Step 1**: Select four feature pairs at random
  - **Step 2**: Compute the homography using the four random feature pairs
  - **Step 3**: Compute the inliers according to a pre-defined threshold

- Keep the set with the most inliers
- Recompute the homography using all the inliers from the set. This gives you the final homography estimation.

2.2.4 Image Warping

Upon obtaining the homography estimation for transformation, the source image is shifted (or transformed) according to the homography. The process for the transformation is called *Image Warping*. Specifically, given the computed homography using RANSAC and a source image, the transformed source image can be computed through *Image Warping*. In the code implementation, we adopted `warpPerspective` method defined in *OpenCV* package. Note that the size of the transformed source image can be specified so that the whole transformed image is visible in the final output[3].

2.2.5 Gain Compensation

After warping the source image, the resulting transformed image can be used to be stitched with the target image. However, there might be some obvious differences in brightness within the overlapping region between two images. Therefore, it is helpful to find the overall gains to adjust the brightness for images. Note that this adjustment can be specified by users of the program (by setting `-gc`) depending on their preferences for the gain compensation. In particular, we adopted a simple gain adjustment. The steps are as follows:

- **Step 1**: Compute average RGB intensity of each image in overlapping region, respectively
- **step 2**: Normalize intensity values for the brighter image by ratio of averages

After the aforementioned two steps for gain compensation, the source and target image tend to have similar brightness within the overlapping region.

[3]We added the offsets along x and y direction calculated from step 4 in 2.1.3

2.2.6 Image Blending

Image Blending serves as one of the most important steps in the process of image stitching. It is in charge of blending the overlapping region of the two images together to create a mosaic. In this project, we adopted four well-known blending techniques, including 1) Average Blending; 2) Alpha Blending; 3) Feathering Blending and 4) Pyramid Blending. As mentioned in section 2.1.1, users can alternatively select all the blending techniques. The details for each of the blending approach will be illustrated as follows.

Average Blending

Looping through the pixel locations in the overlapping region between two images, for each particular pixel location, the RGB intensity values are re-calculated by taking the average RGB intensity values between the source and target image.

Alpha Blending

Similar to *Average Blending*, we loop through the pixel locations in the overlapping area. Specifically, for each pixel p, we 1) add up the RGB values, which are alpha pre-multiplied and 2) divide p's accumulated RGB values by its summation of alpha values. In particular, we determine the alpha values according to the contribution of each image to the pixel p. For example, we assign the contribution according to the distance between p and the centers of the two images.

Feathering Blending

In the overlapping region between two images, feathering can be deployed for the purpose of blending. In particular, for the source image, the pixel intensity values within the overlapping region get linearly decreased[4] from left to right and become closer to zero near the boundary of the overlapping area. In contrast, for the target image, the pixel RGB values within the overlapping area get linearly increased from left to right. In our implementation, we adopted two masks, containing values ranging from 0 to 1, for

[4]The speed of decreasing depends on a pre-defined window size. We will probe the effect of window size on the output in later chapters.

both source and target images in the overlapping region. The two masks function as the ramps to either gradually decrease (for source images) or increase (for target images) the intensity values in the overlapping region. After feathering for both the source and target image, we simply add up their intensity values together in the overlapping area to obtain the final blended result.

Pyramid Blending

Another blending technique adopted in our project is *Pyramid Blending*. The general pipeline for *Pyramid Blending* is shown as follows: 1) Form the Gaussian pyramids for both source and target images; 2) Form the Laplacian pyramids using the Gaussian pyramids from last step for both source and target images; 3) Create a new Laplacian pyramid from two Laplacian pyramids in step 2; 4) Reconstruct a new Gaussian pyramid from the laplacian pyramid constructed in step 3 and take the lowest level of the Gaussian pyramid as the final result after blending. All the code implementations for *Pyramid Blending* are encoded in `utils.py` file.

Chapter 3

Results Obtained and Explanations

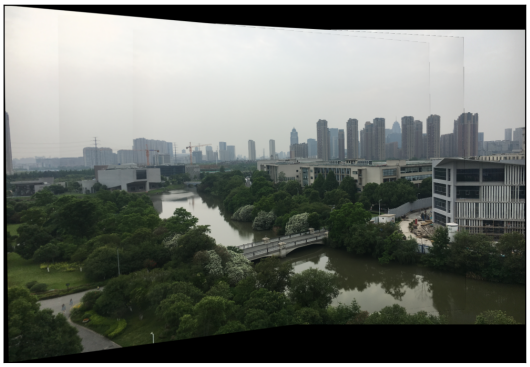
In this chapter, the results obtained associated with each of the feature we proposed will be illustrated. Also, the command for rendering these results will be discussed. Specifically, we provided two sets of images for testing, which are stored in `/images` directory. Users need to specify the location of each of the image to be stitched.

3.1 Stitch any three, four or five images into a panorama

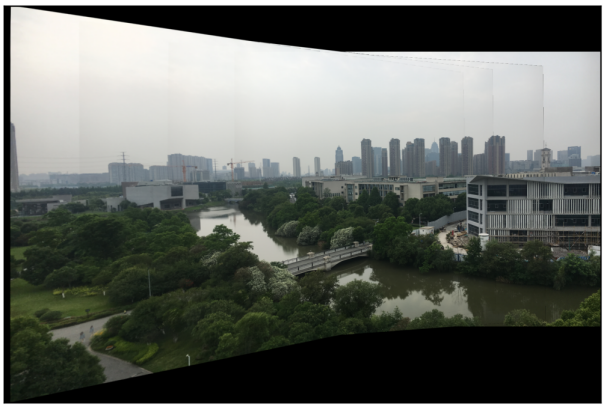
The commands users may use to stitch images are shown below. As you can see, we allow users to select three to five images to be stitched together[1]. The results out of all three commands are shown in Figure 3.1. For demonstration purposes, we choose *Average Blending* for all three panoramas. Also note that if users select less than three or more than five images to be stitched, the program will generate error messages and will be terminated.

```
$ python main.py -i 01.png 02.png 03.png -b avg
$ python main.py -i 01.png 02.png 03.png 04.png -b avg
$ python main.py -i 01.png 02.png 03.png 04.png 05.png -b avg
```

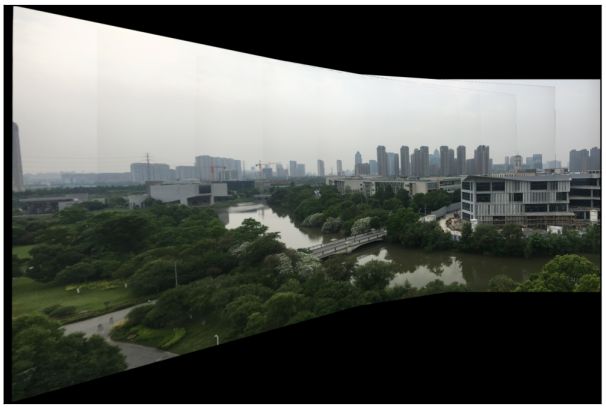
[1]Note that these images can be either from set1 or set2. We use images from set1 for this example



(a) Stitch three images



(b) Stitch four images



(c) Stitch five images

Figure 3.1: Panorama results using images from set1

3.2 Select one of the blending techniques for blending overlapping region between two images

With `-b` or `--blending` option, users can select desired blending techniques for the output panorama. As discussed in section 2.2.6, there are four alternative blending methods to be selected, including *Average Blending*, *Alpha Blending*, *Feathering Blending* and *Pyramid Blending*. The possible commands users may need for choosing any of the four blending techniques are shown below. The results are demonstrated in Figure 3.2. Note that in this example we use images from set2.

```
$ python main.py -i 01.png 02.png 03.png -b avg
$ python main.py -i 01.png 02.png 03.png -b alpha
$ python main.py -i 01.png 02.png 03.png -b feathering
$ python main.py -i 01.png 02.png 03.png -b pyramid
```

3.3 Adopt Gain Compensation

As indicated in section 2.2.5, *Gain Compensation* is deployed to minimize intensity difference of overlapping pixels so that the seam can be reduced in the final panorama. In our program, users can adopt *Gain Compensation* along with any blending techniques. For the purpose of demonstration, we show the results (shown in Figure 3.3) of adopting *Gain Compensation* along with *Average Blending* method. Specifically, `-gc` or `--gainCompensation` is added to the command line to render the output with *Gain Compensation*. As you can visualize from Figure 3.3, the seam on the overlapping region (rightmost) is clearly reduced, suggesting the effect of adopting *Gain Compensation*.

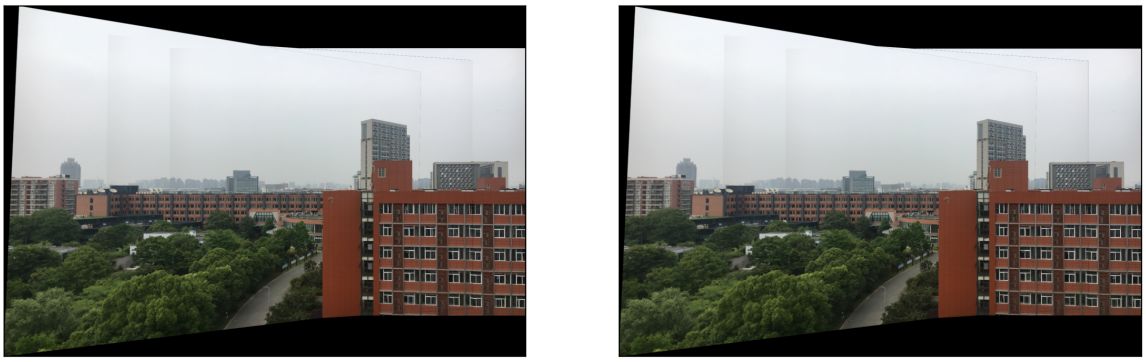
(a) Average Blending

(b) Alpha Blending

(c) Feathering Blending

(d) Pyramid Blending

Figure 3.2: Different blending techniques adopted



(a) With Gain Compensation (b) Without Gain Compensation

Figure 3.3: Average Blending with/without Gain Compensation

3.4 Draw the intermediate inliers while creating the panorama

This feature allows users to visualize the intermediate inliers in the process of creating the final panorama. As introduced in section 2.1.1, users can specify `--drawInliers` option from the command line in order to draw the inliers between two images. The results of the drawing are shown in Figure 3.4 and the command is shown as follows.

```
$ python main.py -i 01.png 02.png 03.png -b avg --drawInliers
```

(a) Inliers between 1st and 2nd image (b) Inliers between stitched 1st-2nd image and 3rd image

Figure 3.4: Visualization of inliers while stitching three images

Chapter 4

Evaluations Based on the Obtained Results

On the basis of the results we obtained from last chapter, we will evaluate our implementations for the images stitching task. In particular, we will discuss the strengths and weaknesses in our approach and feature design.

4.1 Strengths

4.1.1 Allow users to stitch any 3 to 5 images from left to right

Our implementation for image stitching is conducted in a left-to-right manner. One of the strengths for this design is that users can specify any number of images (ranging from three to five) from either set1 or set2 to be stitched together. The use cases have been demonstrated in section 3.1. Also, note that we do not enforce the order of the input image locations. For example, the following two commands will render the same panorama.

```
$ python main.py -i 01.png 02.png 03.png -b alpha
$ python main.py -i 03.png 01.png 02.png -b alpha
```

4.1.2 Allow users to select desired blending approaches

The `-b` or `--blending` option accepts four possible choices, including `avg`, `alpha`, `feathering` and `pyramid`. Users can experiment with different blending techniques and choose the one triggering the best results. Based on our experimentation results, referring to Figure 3.2, *Pyramid Blending* gave us the best results, where seams were almost gone in the overlapping region. Note that the black regions along the boundary of the output are caused by adding the offsets calculated along x and y directions.

4.1.3 Allow Gain Compensation for refining the results

Referring back to Figure 3.3, the effect of adopting Gain Compensation in terms of reducing seams in the overlapping region is pretty obvious. In fact, we also tried to employ *Gain Compensation* together with *Pyramid Blending* for stitching images provided in lab04. The results are demonstrated in Figure 4.1. As you can tell, from figure 4.1(a), the intensity difference of overlapping pixels has been minimized using *Gain Compensation*.

(a) With Gain Compensation (b) Without Gain Compensation

Figure 4.1: Pyramid Blending with/without Gain Compensation for Lab04 images

4.1.4 Allow users to visualize the inliers

As illustrated in Figure 3.4, the intermediate inliers can be visualized when users specify `--drawInliers` option in the command line. This is beneficial for those who wants to

visualize the good matching points between two images for each iteration of stitching.

4.1.5 Allow users to specify the scale percentage to resize input images

Setting `-sp` or `--scalePercent` to any values between 10 to 100 allows the program to resize all the input images before stitching. This is mainly designed to speed up the overall calculations. Users can specify the scale percentage according to certain computational speed requirements.

4.2 Weaknesses

4.2.1 Cannot recognize panoramas given two sets of images

Given the nature of our implementation (i.e. stitch the input set of images from left to right), it would be infeasible to perform panorama recognition if more than one set of images are given as inputs to the program. Specifically, we assume that all the images are from the same set (either set1 or set2). Therefore, if the input images are from different set, the program will fail to work.

4.2.2 Effect of feathering depends on window size

As illustrated in section 2.2.6, feathering enables the pixel intensity values within the overlapping area to constantly change according to a pre-defined window size. In particular, the larger the window size, the more slowly the pixel values get changed. The underline weakness of our design is that users can not explicitly change the window size from command line in order to adjust the feathering results. Instead, users need to refer to the source code implementing *Feathering Blending* and tweak the window size from there. The effect of window size for the feathering results is demonstrated in Figure 4.2. As you can see, larger window size renders the result that is more smooth in the overlapping region, whereas smaller window size triggers obvious seams on the overlapping

boundary.

(a) Smaller window size (b) Larger window size

Figure 4.2: Effects of window size on feathering blending

4.2.3 Effect of Pyramid Blending depends on level of pyramid

According to section 2.2.6, the general pipeline for pyramid blending involves in generating *Gaussian pyramids* and *Laplacian pyramids* for each input image. In particular, the level of pyramids serves as a parameter for the whole *Pyramid Blending* procedure. Nevertheless, similar to how we implement feathering, users can not explicitly assign the level of pyramid for *Pyramid Blending* from command line. Instead, users needs to refer back to the source code of *Pyramid Blending* in our implementation in order to try different values for levels. The effects of number of pyramid levels on *Pyramid Blending* is demonstrated in Figure 4.3. It is noticeable that more levels of pyramid generate more noises in the overlapping region.

4.2.4 Distortion of final output when input more images to be stitched

As shown in Figure 3.1(b) and 3.1(c), as we increase the number of images to be stitched, the left part of the panorama tends to be distorted and stretched. The reason for this is as follows. In each iteration of stitching, we transform the source image according to a

(a) Three levels of pyramids

(b) Five levels of pyramids

Figure 4.3: Effects of number of levels on Pyramid Blending

transformation matrix (i.e. Homography) to stitch with the target image. The stitched result becomes the source image for the next iteration of stitching. Then, taking the next target image, we again transform the source image according to a new Homography to stitch with the new target image. During this whole procedure, the source image for each iteration (except for the first iteration) is the stitched result from last iteration. Consequently, the original source image from the first iteration will be transformed according to multiple intermediate homography, resulting in the distorted final output panorama.

Bibliography

[1] Lowe, D. G. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision 60*, 2 (Nov. 2004), 91–110.