Motivation

The topic of my bachelor's thesis was the fundamental matrix, why it's needed in two view geometry, its rigorous mathematical background, and ultimately the steps of the 8-point algorithm which computes this matrix. While I understood this algorithm in depth I hadn't programmed it. This was my motivation for my project this semester. The project of programming the 8-point algorithm ended up being too small for this assignment, so I delved deeper into image processing.

Steps of two view image processing

First step in two view image processing is matching keypoints in the two images. From these matched keypoints the 8-point algorithm gives the fundamental matrix, from which the camera positions are calculated. Up to this point the process is relatively standard. How to continue depends on the goal of the project. One can stitch together pictures, to form a panorama picture, or a 3D surrounding globe, locating objects in the image. I chose a 3D reconstruction of the captured item as the end goal.

Making a 3D reconstruction can be done in many ways: obtaining information from shadows, textures, edges, pictures with different focal lengths. From two pictures with different angles one can calculate the depth of the matching points. As always in real world data processing, more is always better: in this case using more than just two pictures gives more information. Choosing one, then using a set of images of the same scene the depth between the origin and all the other images can be calculated, and the mean depth map can be created giving more accurate results.

At this point a database of three dimensional points has been built! This is what my program does using OpenCV in Python.

Matching points

Matching points happens in two steps: first keypoints are found and descriptors are created, then they are matched. A descriptor of a keypoint can be different depending on which algorithm is used. These are the algorithms I tried:

- **Harris corner detector** detects corners using the gradient of a greyscale picture. This used to be useful if the two images are illuminated differently, because corners and edges stay corners and edges, while little else seems the same. Since then many more complicated and more successful detectors have been built. Its output is just the locations of the points, marked in green on Figure 3.

- **SIFT (Scale Invariant Feature Transform)** A corner is a corner even if you rotate it. But if you zoom in close enough it becomes too smooth to be a corner. This was the motivation for SIFT, which is a complex algorithm that searches for keypoints with several measures in place to attain invariance to rotation, scaling, illumination changes. From the keypoints' neighborhood it constructs descriptors, which store (among other things) an orientation vector.
  I based the later steps of my program on matching keypoints obtained with SIFT.

- **FAST (Features from Accelerated Segment Test)** This one is indeed fast, and returns only keypoints. I only briefly experimented with it, I didn't delve deeper.

- **ORB (Oriented FAST and Rotated BRIEF)** Orb combines two lesser detectors in smart ways. ORB, SIFT and SURF are the three main matching algorithms, of which SURF still has a patent, so I was not able to use it, and the patent of SIFT has expired. But ORB is not and has never been patented.

- **Brute Force Matcher** The descriptors exist in a space, which one depends on their format. A matcher matches them by the distance in this space. It can also return k nearest neighbors.

- **FLANN (Fast Library for Approximate Nearest Neighbors)** This is a library of matching algorithms used in big datasets with high dimensional descriptors. As with most of these algorithms the actual implementation on my end is learning when to use the algorithm and learn to tune the hyperparameters of the python functions.

During my experimentation with these detection and matching algorithms I stumbled upon a very interesting problem when I used SIFT as seen in Figure 1. A lot of keypoints were detected as the dips on the surface of the tatami, but they were matched to a singe point on the other image. I could solve the problem by changing the hyperparameters of SIFT, but only by filtering out the tatami points. The problem is this way I lose a lot of keypoints, not only the ones on the tatami. So I tried different combinations to get better results I tried switching to FLANN, but that didn't help. I also tried switching the detector to ORB (and FAST, though ORB

is an expansion of it) this helps, because an ORB based matcher is less likely to detect the tatami points, but it also has less matches overall in these two pictures.
I did find a way to match tatami points, but only on two other pictures. If the images are closer together as in Figure 2, the tatami points are matched effortlessly. This makes sense, if the images differ less then the keypoint descriptors from one image and the other will be more similar. More similar is closer in the descriptor space thus more likely to be matched to each other than to accidentally similar other points. But this is not a satisfying solution, especially because an obvious way to improve would be to use their relative positions to each other in some way.

Depth map

A depth map is the same size as the source image, but instead of RGB it stores a number representing depth. In Python a **StereoMatcher** calculates the depth map from two images. This can be displayed in 2D as in Figure 4, where there is a gradient representing different depths. More pictures give more accurate results so my program takes a sequence of images and takes one of them and calculates the depth maps relative to the other images. It then transforms these so they are aligned and takes the mean. (OpenCV provides a built-in **getPerspectiveTransform**, but that only uses four points from the input images so I used **findHomography** using a matching).

This is the ultimate depth map my program can construct as seen in Figure 5

An attentive reader might notice that I skipped from matching the points straight to depth maps. Where are the rest of the steps, the 8-point algorithm, the camera reconstruction? In truth the StereoMatcher class is such a powerful tool it has these steps already built in. Moreover it performs the matching algorithm as well. This is why the three main parts that I coded are: the **8-point algorithm** which can use the input from the **matching algorithms** and **depth maps**, which could operate entirely separate from the other two (except for a tiny optional improvement where I used matchings to obtain the homograpy).

Thoughts, from here

This reconstruction works with images that are very close together, which makes sense because in application 3D reconstruction is done from video input with small differences from one frame to the other. A possible improvement would be to look at the given object from a totally different angle, and comupute additional depth maps. This would result in information about previously hidden parts leading to a complete model.

Another possible future for this project is solving the problem found in SIFT matching tatami points. This is a reproducible error (other tatami pictures have the same problem with SIFT matching) delving into the exact mechanics and mathematics of SIFT and trying to improve it is a viable option for continuing my project.

All in all I am satisfied with how my project turned out. I gained a lot of experience programming and am glad that I could learn about these fascinating algorithms.

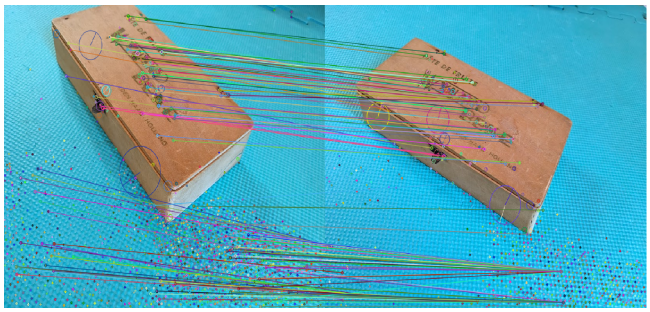

Figure 1: SIFT error

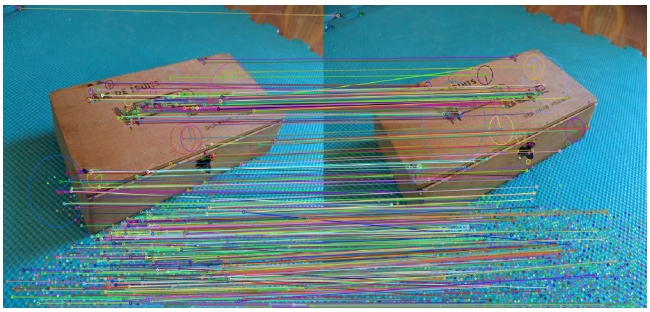

Figure 2: Good SIFT matching

Figure 3: Keypoints

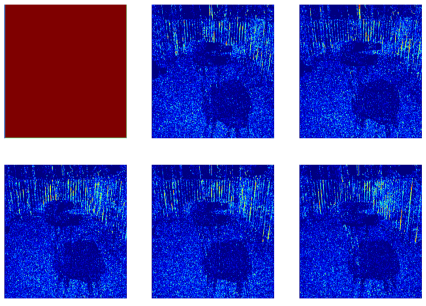

Figure 4: Depth maps relative to the first one

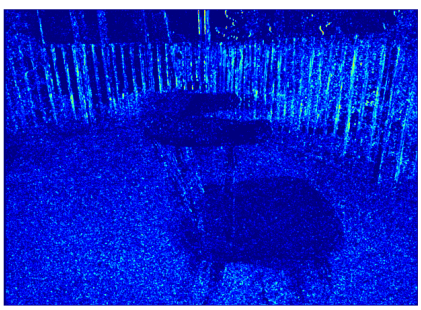

Figure 5: Mean depth mao