CSE463 Introduction to Computer Vision
HW3 Report

Abdullah Küsgülü
1801042606

Stereo Correspondence Algorithms

General Correspondence

Constructing the disparity image is same for all the methods I will be explaining in the next subsection and it works as follows:

After the matching pixels are found for the stereo images using the methods explained below, a zero-image that has same dimensions with stereo images is created. For every pixel mapping $p_l \rightarrow p_r$ from left stereo image to right stereo image, the value of p_l in the disparity image is found by multiplying the Euclidian distance between p_l and p_r with 8.

Feature Detection and Matching

I have created 6 stereo correspondence algorithms using OpenCV feature detection algorithms.

- Canny Edge Detector only
- Canny combined with Harris Corner Detector
- SIFT detector combined with FAST detector
- SIFT detector combined with FAST detector with BRIEF descriptors
- Canny, SIFT and FAST combined together
- Canny combined with ORB detector

In the first one I applied Canny to both stereo images. After that I generated key points from the non-zero pixels of Canny output images. Then I created descriptors from these key points using BRIEF and matched them using brute force matching.

For the second one I combined the key points calculated as mentioned in previous method using Canny with key points I calculated using Harris Corner Detector to try to increase the number of pixels covered. After combining the key points the rest is the same as previous method.

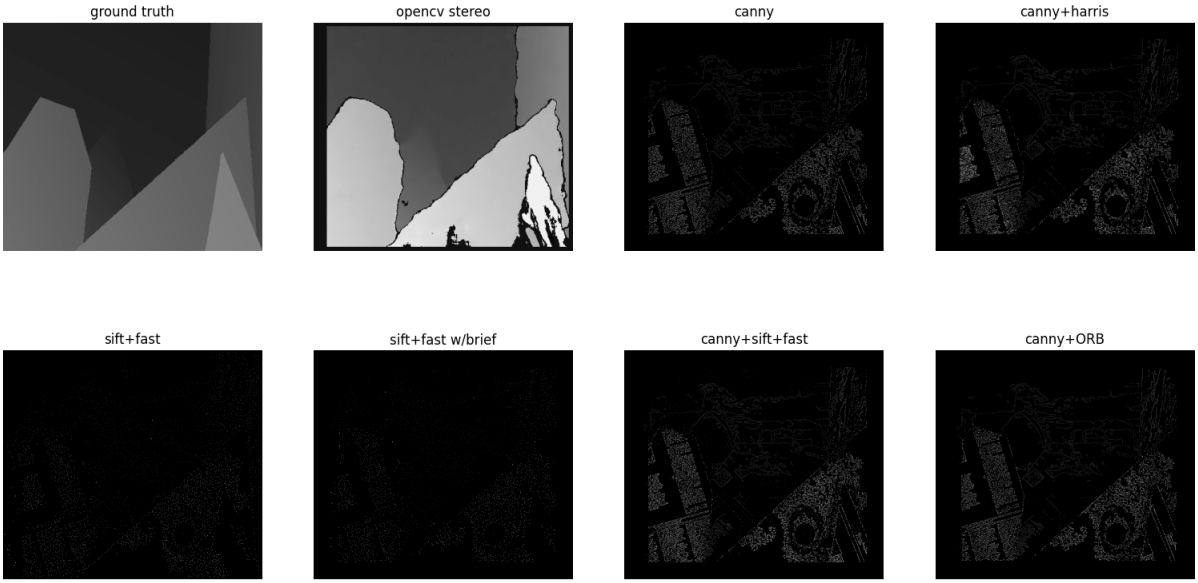
Third one only uses SIFT and FAST detectors. Because of this the pixel coverage is much less than the previous methods, so you might need to zoom in to the images in Tests section. Just like the previous ones I compile a list of key points for both stereo images using SIFT and FAST detect() method. After combining these key points I calculate the descriptors using the compute() method of SIFT. Then I match these key points and descriptors of two images using BF matching.

Fourth one aims to see if changing the way we calculate descriptors effect the disparity image. It is basically the same procedure as the third one with only difference being the way we calculate the descriptors. Instead of using SIFT.compute() to calculate the descriptors I used BRIEF.compute().

Fifth one combines Canny, SIFT and FAST all together. Key points of each image are calculated from Canny output image and with using SIFT and FAST detect() methods. The key points are then combined together and descriptors are created using BRIEF. After this BF matching is applied to find corresponding pixels.

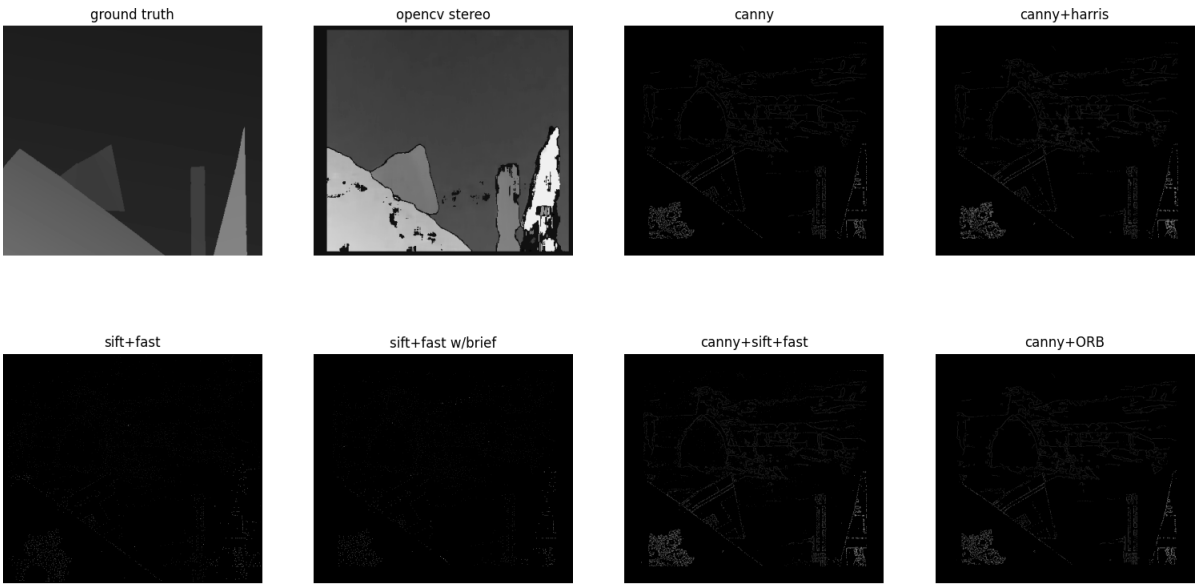
The last one combines Canny with ORB instead of SIFT and FAST to see if that makes any difference. The only differences with the previous one are how we calculate the key points and what flag we use while applying BF matching. Key points are calculated using ORB instead of SIFT and FAST and the flag we use is NORM_HAMMING instead of NORM_L2.

Tests



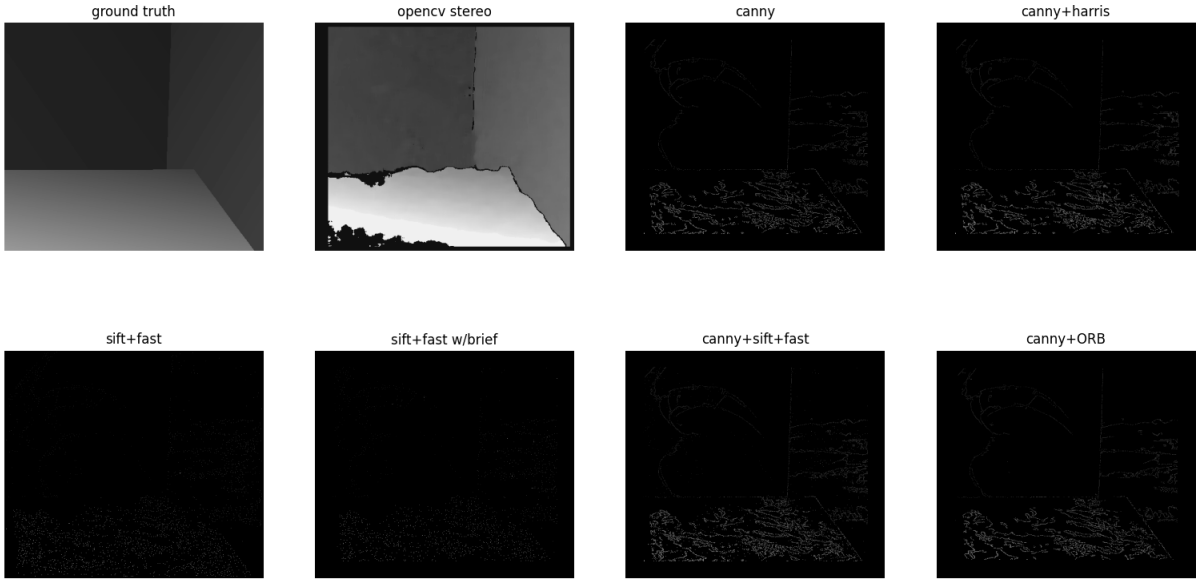

```
Mean Squared Errors and Execution Times for barn1
==================
OpenCV: 109.535415, 0.00400s
Canny: 94.915743, 3.02458s
Canny+Harris: 94.081942, 4.48616s
Sift+Fast: 99.649454, 0.77343s
Sift+Fast w/BRIEF: 100.097459, 0.43960s
Canny+Sift+Fast: 93.446291, 4.90470s
Canny+ORB: 94.437543, 3.27266s
```

Barn 1



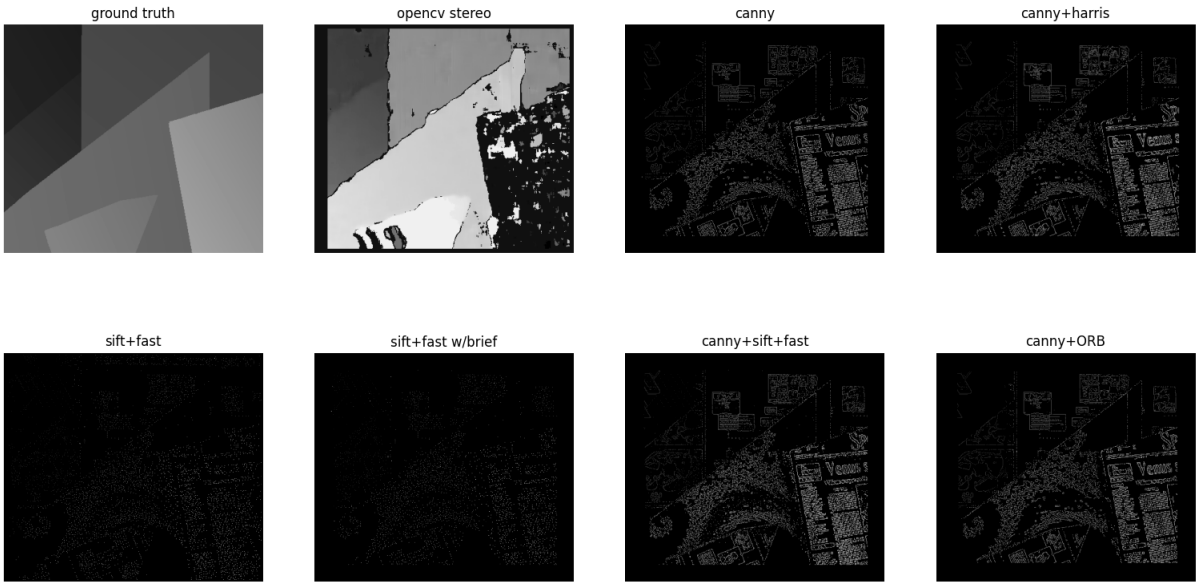

```
Mean Squared Errors and Execution Times for barn2
==================
OpenCV: 113.830458, 0.00500s
Canny: 95.060697, 0.98574s
Canny+Harris: 94.570091, 1.45332s
Sift+Fast: 97.219746, 0.42273s
Sift+Fast w/BRIEF: 97.470646, 0.25266s
Canny+Sift+Fast: 94.109931, 1.75012s
Canny+ORB: 94.939926, 1.21702s
```

Barn 2



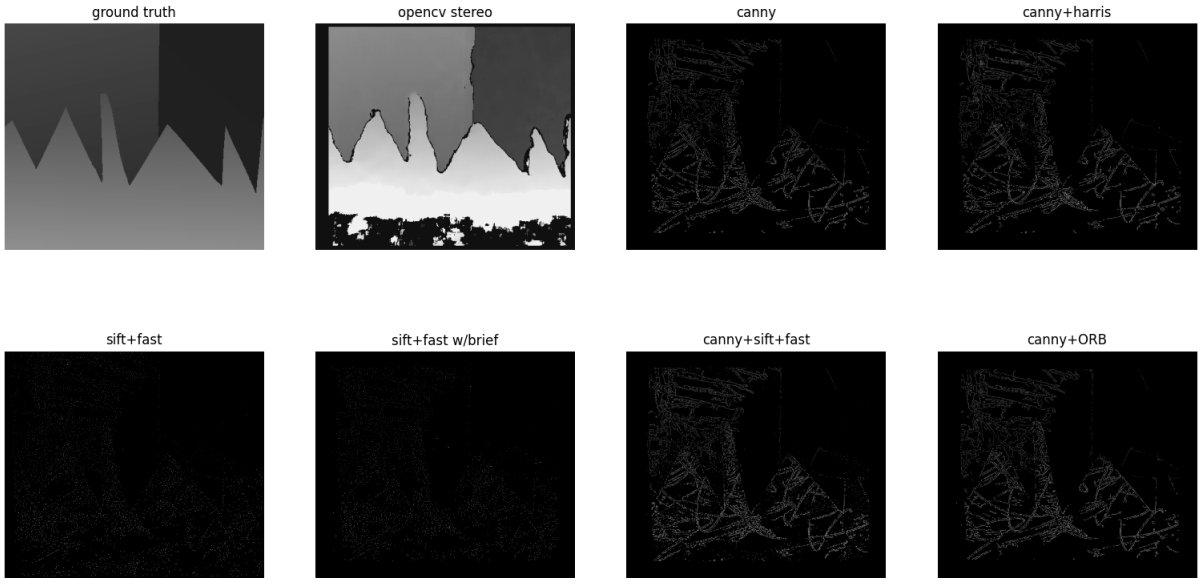

```
Mean Squared Errors and Execution Times for bull
====================
OpenCV: 111.881829, 0.00400s
Canny: 102.814079, 0.68087s
Canny+Harris: 102.473277, 0.94285s
Sift+Fast: 104.573457, 0.37961s
Sift+Fast w/BRIEF: 105.039255, 0.19833s
Canny+Sift+Fast: 102.026071, 1.20904s
Canny+ORB: 102.693210, 0.90534s
```

Bull



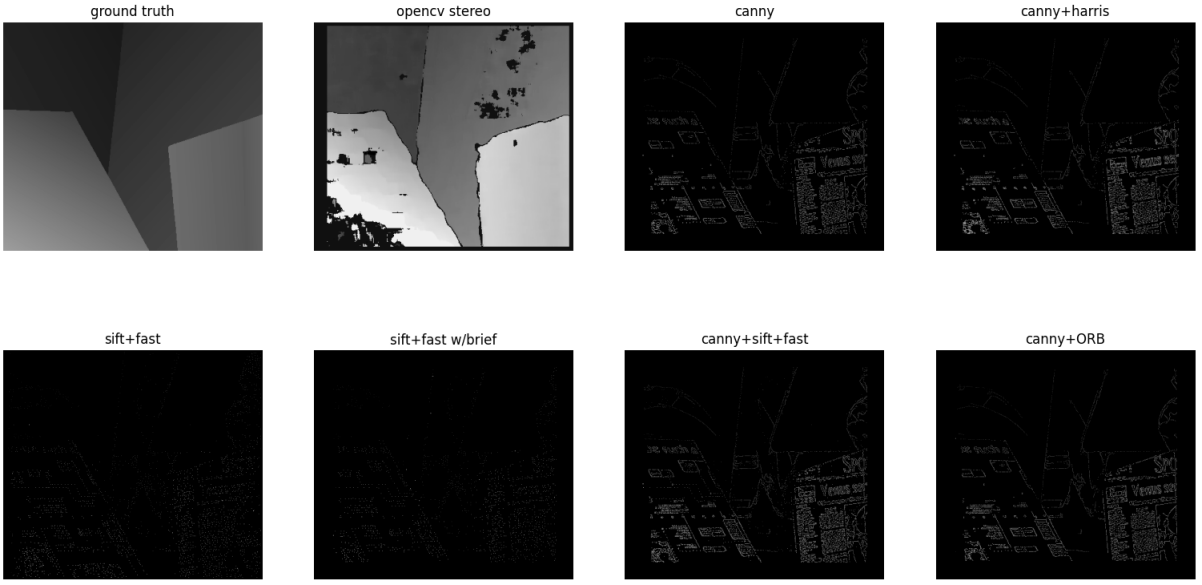

```
Mean Squared Errors and Execution Times for poster
====================
OpenCV: 105.301528, 0.00500s
Canny: 103.587389, 3.85201s
Canny+Harris: 102.869032, 5.09671s
Sift+Fast: 110.865892, 0.86326s
Sift+Fast w/BRIEF: 111.456787, 0.49083s
Canny+Sift+Fast: 101.703700, 6.25988s
Canny+ORB: 103.159353, 4.14543s
```

Poster




```
Mean Squared Errors and Execution Times for sawtooth
====================
OpenCV: 100.733107, 0.00500s
Canny: 83.547441, 1.82868s
Canny+Harris: 83.310660, 2.20851s
Sift+Fast: 85.900079, 0.65608s
Sift+Fast w/BRIEF: 86.330639, 0.35503s
Canny+Sift+Fast: 82.681330, 3.12762s
Canny+ORB: 83.272920, 2.08098s
```

Sawtooth




```
Mean Squared Errors and Execution Times for venus
====================
OpenCV: 108.440670, 0.00500s
Canny: 107.228562, 1.20267s
Canny+Harris: 106.760122, 1.67660s
Sift+Fast: 110.364416, 0.43403s
Sift+Fast w/BRIEF: 110.828332, 0.19901s
Canny+Sift+Fast: 106.381057, 1.92298s
Canny+ORB: 107.209780, 1.43168s
```

Venus

Results

The OpenCV stereo is the most similiar one to the ground truth visually but the MSE is usually higher than other methods. I think that is caused by most of the pixels being brighter than the corresponding pixels in the ground truth.

The methods that use Canny are the most accurate both MSE wise and visually. That is because the Canny Edge Detector gives us the most pixel coverage out of all the feature detection methods I used. But compared to other methods, the methods using the Canny are much slower. Using C++ instead of Python might decrease the difference between these groups but I have not tried it.

The methods that only use SIFT and FAST are much faster but visually they suffer too much. Because the pixel coverage is much less than the methods with Canny, you need to zoom in to see the pixels in the image.

Looking at these results I can say that the most accurate one of my methods was the one that combines Canny, SIFT and FAST. The ones that combine Canny with Harris and Canny with ORB are following in the second place.

All my methods are slower compared to OpenCV method much to my surprise. I expected at least some of my methods such as SIFT with FAST be faster than the OpenCV one because my methods create sparse maps instead of dense maps. But I guess the OpenCV method is optimized pretty thoroughly and my methods pale in comparison.