1. SIFT.cpp

코드 목적:

SIFT descriptor 를 이용해 특징점이 되는 keypoint 를 찾고, feature matching 을 수행한다

코드 흐름:

1) 이미지 파일을 가져와 흑백으로 전환한다.
2) detector에 사용되는 값들을 정의한다.
3) 두 입력 이미지를 옆으로 붙여서 하나의 큰 이미지를 만든다.
4) detect 함수와 compute함수를 이용해 keypoints를 찾고, 원을 그린다.
5) findPairs()함수를 이용해 가장 가까운(matching되는) keypoint를 반환한다.
6) 대응하는 점들을 빨간 선으로 연결한다.

함수 설명

euclidDistance(Mat& vec1, Mat& vec2)

vec1과 vec2 사이의 거리(둘이 얼마나 다른지)를 반환한다. 유사할수록 값이 작다.

nearestNeighbor(Mat& vec, vector<KeyPoint>& keypoints, Mat& descriptors)

keypoints를 가지는 descriptors위의 점 중 vec와 가장 유사한, matching되는 점의 인덱스를 반환한다.

SecondNearestNeighbor(Mat& vec, vector<KeyPoint>& keypoints, Mat& descriptors, int first)

keypoints를 가지는 descriptors위의 점 중 vec와 두번째(first 다음으로)로 유사한 점의 인덱스를 반환한다.

void findPairs(vector<KeyPoint>& keypoints1, Mat& descriptors1, vector<KeyPoint>& keypoints2, Mat& descriptors2, vector<Point2f>& srcPoints, vector<Point2f>& dstPoints, bool crossCheck, bool ratio_threshold);

descriptor1 의 keypoints1 에 matching 되는 descriptor2 의 keypoints2 를 찾아 srcPoints (descriptor 1 의 점)와 dstPoints(descriptor 2 의 점)에 저장한다.

crossCheck: cross check 여부, ratio_threshold: threshold ratio 사용 여부

OpenCV 제공 함수

detect (InputArray image, std::vector< KeyPoint > &keypoints)

image에서 keypoint를 detect해 keypoints변수에 저장한다.

compute (InputArray image, std::vector< KeyPoint > &keypoints, OutputArray descriptors)

image에서 detect된 keypoints들에 대해 계산해 descriptors행렬에 계산해 반환한다.

circle(img, Point(x, y), radius, Scalar(b,g,r), thickness, lineType, shift)

img의 Point(x, y)들에 radius의, Scalar색의, thickness와 lineType을 가지는 원을 그린다.

line(img, from, to, Scalar(b,g,r))

img에 from부터 to까지 Scalar색의 선을 그린다.

설명

keypoint 와 descriptor 를 찾는 과정은 OpenCV 가 제공하는 함수를 이용한다.

```
FeatureDetector* detector = new SiftFeatureDetector(
        0,          // nFeatures
        4,          // nOctaveLayers
        0.04,       // contrastThreshold
        10,         // edgeThreshold
        1.6         // sigma
);

DescriptorExtractor* extractor = new SiftDescriptorExtractor();
vector<KeyPoint> keypoints1;
Mat descriptors1;
detector->detect(input1_gray, keypoints1);
extractor->compute(input1_gray, keypoints1, descriptors1);
```

두 이미지에 대해 keypoints 와 descriptor 가 찾아졌다. 이 findPairs()를 호출해 유사한 점들끼리 matching 해보자

1) Nearest neighbor 을 이용해 feature matching을 한다.

```
for (int i = 0; i < descriptors1.rows; i++) {
        KeyPoint pt1 = keypoints1[i];
        Mat desc1 = descriptors1.row(i);

        int nn = nearestNeighbor(desc1, keypoints2, descriptors2);
```

```
int nearestNeighbor(Mat& vec, vector<KeyPoint>& keypoints, Mat& descriptors) {
    int neighbor = -1;
    double minDist = 1e6;

    for (int i = 0; i < descriptors.rows; i++) {
        Mat v = descriptors.row(i);
        double dist = euclidDistance(vec, v);
        if (dist < minDist) {
                minDist = dist;
            neighbor = i;
        }
    }

    return neighbor;
}
```

```
        KeyPoint pt2 = keypoints2[nn];
        srcPoints.push_back(pt1.pt);
        dstPoints.push_back(pt2.pt);
}
```

각 keypoint에 대해 nearest neighbor keypoint를 찾는다.
nearestNeighbor함수는 descriptors2를 모두 돌며 img1의 keypoints와의 distance가 가장 작은, 즉 가장 유사한 행을 찾고 그 index를 반환한다.
img1의 keypoints중 하나인 pt1에 대응하는 img2위의 keypoint pt2를 찾았다면 이를 각각 scrPoints와 dstPoints에 push해준다.

두 keypoints의 유사도를 알아내기 위해 euclidDistance() 함수를 사용한다.
이 함수에서는 vec1을 돌며 vec2와 얼마나 차이가 나는지 계산한다.

```
double euclidDistance(Mat& vec1, Mat& vec2) {
    double sum = 0.0;   int dim = vec1.cols;
    for (int i = 0; i < dim; i++) {
        sum += (vec1.at<uchar>(0, i) - vec2.at<uchar>(0, i)) * (vec1.at<uchar>(0, i) - vec2.at<uchar>(0, i));
    }
    return sqrt(sum);
}
```

2) cross-checking

feature matching의 정확성을 높이기 위해 cross checking을 사용할 수 있다. 이는 findPairs의 매개변수 crossCheck가 true일 때 수행한다.

img1의 keypoint f가 img2의 g에 matching될 때, 반대로 g 또한 f에 매칭되는지 확인하는 작업이다. 서로가 서로에게 가장 유사한 keypoint일 때만 매칭시켜준다.

```
for (int i = 0; i < descriptors1.rows; i++) {
        KeyPoint pt1 = keypoints1[i];
        Mat desc1 = descriptors1.row(i);

        int nn = nearestNeighbor(desc1, keypoints2, descriptors2);

    if (crossCheck) {
            Mat desc2 = descriptors2.row(nn);
            int nn2= nearestNeighbor(desc2, keypoints1, descriptors1);
            if(nn2!=i) continue;
        }

        KeyPoint pt2 = keypoints2[nn];
        srcPoints.push_back(pt1.pt);
        dstPoints.push_back(pt2.pt);
}
```

nearesrNeighbor함수를 이용해 keypoints1[i]에 matching되는 img2위의 점의 인덱스 nn을 구한다. 다시 nearesrNeighbor함수에 nn과 img1의 정보를 넣어, nn과 매칭되는 img1위의 점의 인덱스 nn2를 구한다. nn2가 i와 다르다면 continue해준다. 즉, img1의 keypoint pt1과 가장 유사한 nn이 있고, img2위의 nn과 가장 유사한 점이 다시 pt1이 되는지 확인한다. 아닐 경우, 매칭시켜주지 않고 다음 keypoints에서 연산을 이어나간다.

3) ratio-based thresholding 추가

feature matching의 정확성을 높이기 위해 ratio-based thresholding 추가하자. 이는 findPairs의 매개변수 ratio_threshold가 true일 때 수행한다.

가장 유사한 keypoint와의 차이와 두번째로 가장 유사한 keypoint와의 차이의 비율이 어느정도 클 때만 매칭된다고 인정해주는 것이다. 여기선 예시로 0.65를 그 비율의 threshold로 이용하였다. 즉, 매칭될 keypoint와의 유사성이 다른 어떤 keypoint들보다도 차이나게 유사해야만 한다는 것이다.

이 연산을 위해, 가장 유사한 점의 인덱스 nn을 제외하고 가장 유사한 점을 찾아내는 SecondNearestNeighbor() 함수를 정의한다.

nearestNeighbor()함수와 동일하지만 매개변수로 nn(가장 유사한 점의 인덱스)를 함께 보내 nn이 아닌지를 확인해준다.

```
if (ratio_threshold) {
                int nn2=SecondNearestNeighbor(desc1, keypoints2, descriptors2,nn);

                Mat v1=descriptors2.row(nn);
                double dist1 = euclidDistance(desc1, v1);

                Mat v2=descriptors2.row(nn2);
                double dist2 = euclidDistance(desc1, v2);

                if ((dist1/dist2) > 0.65) continue;
            }
```

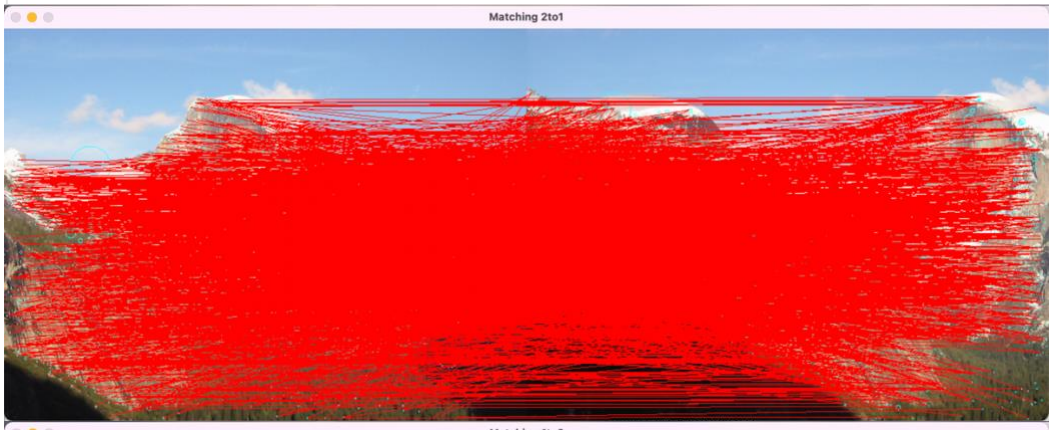
```
int SecondNearestNeighbor(Mat& vec, vector<KeyPoint>& keypoints, Mat& descriptors,int first) {
    int neighbor = -1;
    double minDist = 1e6;

    for (int i = 0; i < descriptors.rows; i++) {
        Mat v = descriptors.row(i);
        double dist = euclidDistance(vec, v);
        if (dist < minDist) {
            if(i==first)continue;
            minDist = dist;
            neighbor = i;
        }
    }

    return neighbor;
}
```

위 코드의 설명은 이미지 2 를 이미지 1 에 featuring matching 하는 과정이었다. main 함수에 새로운 변수를 선언하고 kepoint1 과 2 의 위치를 바꿔 findPairs 함수를 호출하여 이미지 1 을 이미지 2 에 matching 하는 작업 또한 수행해준다.

실행 결과:

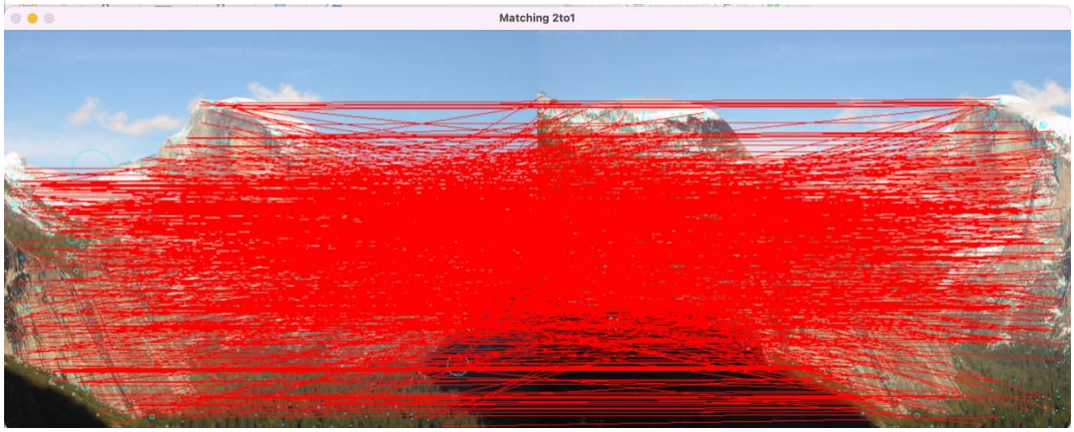
1) bool crossCheck = false; bool ratio_threshold = false;

```
input1 : 2865 keypoints are found.
input2 : 2623 keypoints are found.
2623 keypoints are matched.2to1
2865 keypoints are matched.(1to2)
```



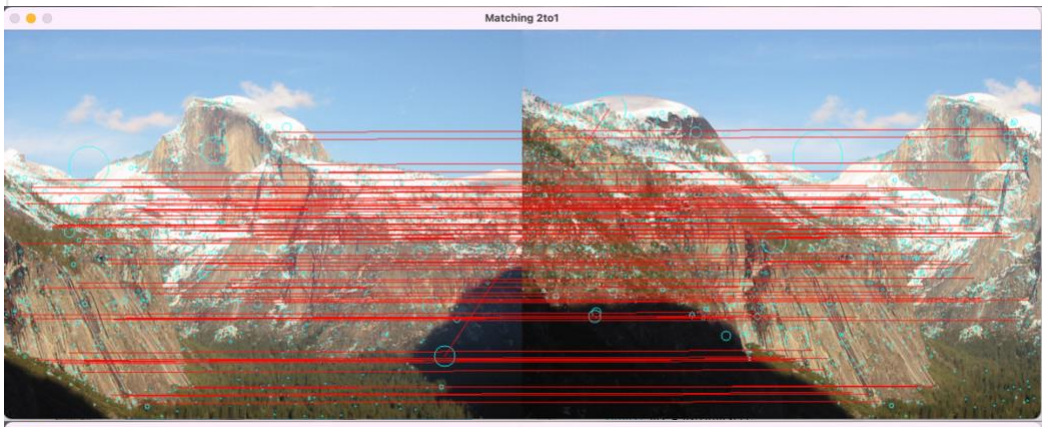
2) bool crossCheck = true; bool ratio_threshold = false;

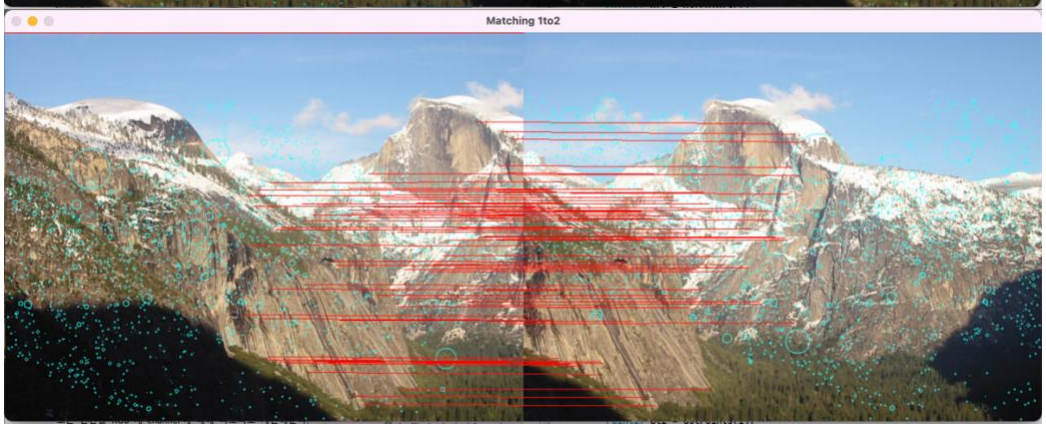
```
input1 : 2865 keypoints are found.
input2 : 2623 keypoints are found.
1050 keypoints are matched.2to1
1050 keypoints are matched.(1to2)
```



3)bool crossCheck = true; bool ratio_threshold = true;

```
input1 : 2865 keypoints are found.
input2 : 2623 keypoints are found.
69 keypoints are matched.2to1
68 keypoints are matched.(1to2)
```





2. feature_homography.cpp

코드 목적: img1 이 포함된 img2 에 대해, matching 되는 keypoints 를 찾고, object 의 테두리를 알아낸다

SURF 는 Speeded Up Robust Features 로, Integral images 를 이용해 SIFT 보다 연산량을 줄일 수 있다.

```
Mat img_object = imread("img1.png", CV_LOAD_IMAGE_GRAYSCALE);
Mat img_scene = imread("img2.png", CV_LOAD_IMAGE_GRAYSCALE);
//-- Step 1: Detect the keypoints using SURF Detector
    int minHessian = 400;
    SurfFeatureDetector detector(minHessian);
    std::vector<KeyPoint> keypoints_object, keypoints_scene;

    detector.detect(img_object, keypoints_object);
    detector.detect(img_scene, keypoints_scene);
```

detect()함수를 이용해 img_obj 와 img_scene 의 keypoints 를 찾는다.

```
//-- Step 2: Calculate descriptors (feature vectors)
    SurfDescriptorExtractor extractor;
    Mat descriptors_object, descriptors_scene;

    extractor.compute(img_object, keypoints_object, descriptors_object);
    extractor.compute(img_scene, keypoints_scene, descriptors_scene);
```

compute()함수를 이용해 descriptor 를 계산한다.

```
//-- Step 3: Matching descriptor vectors using FLANN matcher
    FlannBasedMatcher matcher;
    std::vector< DMatch > matches;
    matcher.match(descriptors_object, descriptors_scene, matches);

    double max_dist = 0; double min_dist = 100;
```

FLANN 은 Fast Library for Approximate Nearesr Neighbors 의 약자로, 큰 이미지에서 특성들을 매칭할 때 성능을 위해 최적화된 라이브러리 모음이다. img_obj 와 img_scene 의 feature matching 으로 FLANN 을 사용한다.

```
//-- Quick calculation of max and min distances between keypoints
    for (int i = 0; i < descriptors_object.rows; i++) {
        double dist = matches[i].distance;
        if (dist < min_dist) min_dist = dist;
        if (dist > max_dist) max_dist = dist;
    }
    printf("-- Max dist : %f ₩n", max_dist);
    printf("-- Min dist : %f ₩n", min_dist);
```

계산된 대응점들 사이의 거리(차이)중 최댓값과 최솟값을 구한다.

```
//-- Draw only "good" matches (i.e. whose distance is less than 3*min_dist )
    std::vector< DMatch > good_matches;

    for (int i = 0; i < descriptors_object.rows; i++) {
        if (matches[i].distance < 3 * min_dist) {
            good_matches.push_back(matches[i]);
        }
    }
```

두 대응점 사이의 거리가 3 x 최솟값 보다 작은 경우만 good matches 로 판단한다.

```
    Mat img_matches;
    drawMatches(img_object, keypoints_object, img_scene, keypoints_scene,
        good_matches, img_matches, Scalar::all(-1), Scalar::all(-1),
        vector<char>(), DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS);
```

openCV 제공 함수 drawMatches()으로 특정점 매칭 영상 생성한다.

```
//-- Localize the object from img_1 in img_2
    std::vector<Point2f> obj;
    std::vector<Point2f> scene;

    for (int i = 0; i < good_matches.size(); i++)
    {
        //-- Get the keypoints from the good matches
        obj.push_back(keypoints_object[good_matches[i].queryIdx].pt);
        scene.push_back(keypoints_scene[good_matches[i].trainIdx].pt);
    }
```

img_2 에 있는 img_1 object 를 localize. obj 와 scene 에 각 대응하는 점을 차례로 집어넣는다.

```
Mat H = findHomography(obj, scene, CV_RANSAC);
```

img_1 의 obj 가 img_2 의 scene 으로 변환될 수 있도록 해주는 변환 행렬 H 선언

```
//-- Get the corners from the image_1 ( the object to be "detected" )
std::vector<Point2f> obj_corners(4);
obj_corners[0] = cvPoint(0, 0);
obj_corners[1] = cvPoint(img_object.cols, 0);
obj_corners[2] = cvPoint(img_object.cols, img_object.rows);
obj_corners[3] = cvPoint(0, img_object.rows);
std::vector<Point2f> scene_corners(4);

perspectiveTransform(obj_corners, scene_corners, H);
```

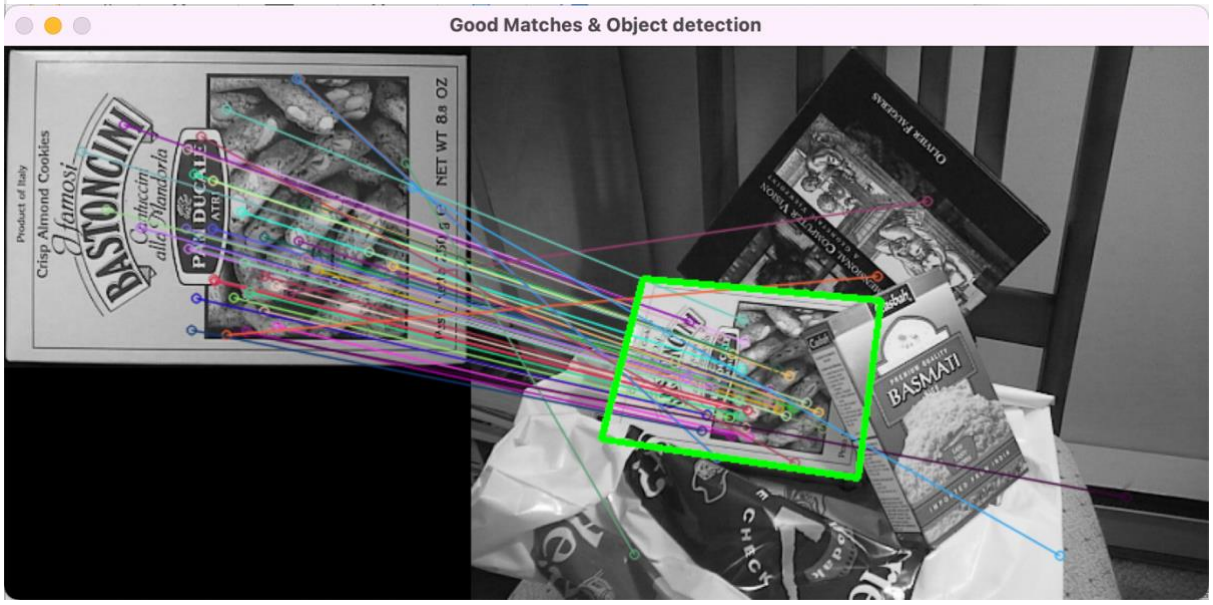
object_corner 을 H 에 따라 변환하여 scene_corners 에 저장한다. 즉, object 를 img_2 의 scene 로 변환할때의 모서리가 scene_corners 에 저장되어있다.

```
//-- Draw lines between the corners (the mapped object in the scene - image_2 )
line(img_matches, scene_corners[0] + Point2f(img_object.cols, 0), scene_corners[1] +
Point2f(img_object.cols, 0), Scalar(0, 255, 0), 4);
line(img_matches, scene_corners[1] + Point2f(img_object.cols, 0), scene_corners[2] +
Point2f(img_object.cols, 0), Scalar(0, 255, 0), 4);
line(img_matches, scene_corners[2] + Point2f(img_object.cols, 0), scene_corners[3] +
Point2f(img_object.cols, 0), Scalar(0, 255, 0), 4);
line(img_matches, scene_corners[3] + Point2f(img_object.cols, 0), scene_corners[0] +
Point2f(img_object.cols, 0), Scalar(0, 255, 0), 4);

//-- Show detected matches
imshow("Good Matches & Object detection", img_matches);
```

계산한 scene 의 corner 를 이용해 scene 에 포함된 object 에 테두리 선을 그어준다.
결과 이미지를 새로운 창에 띄운다.

실행 결과

```
-- Max dist : 0.671161
-- Min dist : 0.070356
```



참고자료:

오픈 SW 프로젝트 Lec08 수업자료