특징점 매칭

: 2개의 영상에서 추출한 특징점을 비교하여 서로 비슷한 특징점을 찾는다.

- BFMatcher 방법 : 모든 특징점끼리 비교하는 방법
- FLANN 기반 방법 : 근사화 기법을 이용하여 빠른 매칭을 수행하는 방법

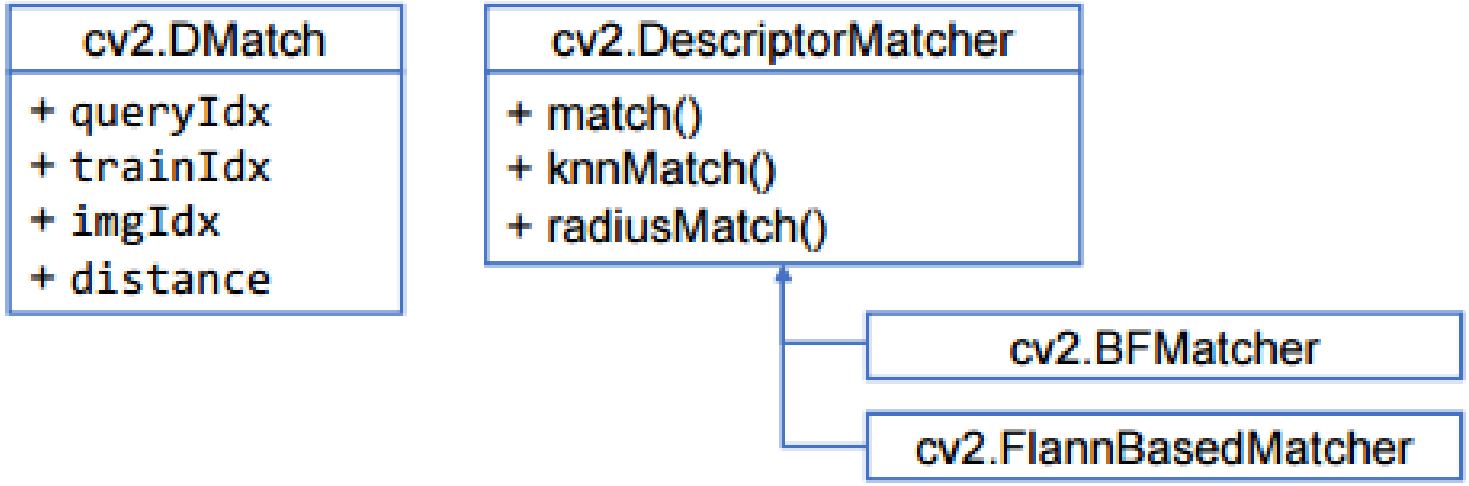
Dmatch 클래스

: 여러 영상에서 추출한 특징점 사이의 매칭 정보를 표현하고 저장할 수 있다.

```
01    class DMatch
02    {
03    public:
04        DMatch();
05        DMatch(int _queryIdx, int _trainIdx, float _distance);
06        DMatch(int _queryIdx, int _trainIdx, int _imgIdx, float _distance);
07
08        int queryIdx;
09        int trainIdx;
10        int imgIdx;
11
12        float distance;
13
14        bool operator<(const DMatch &m) const;
15    };
```

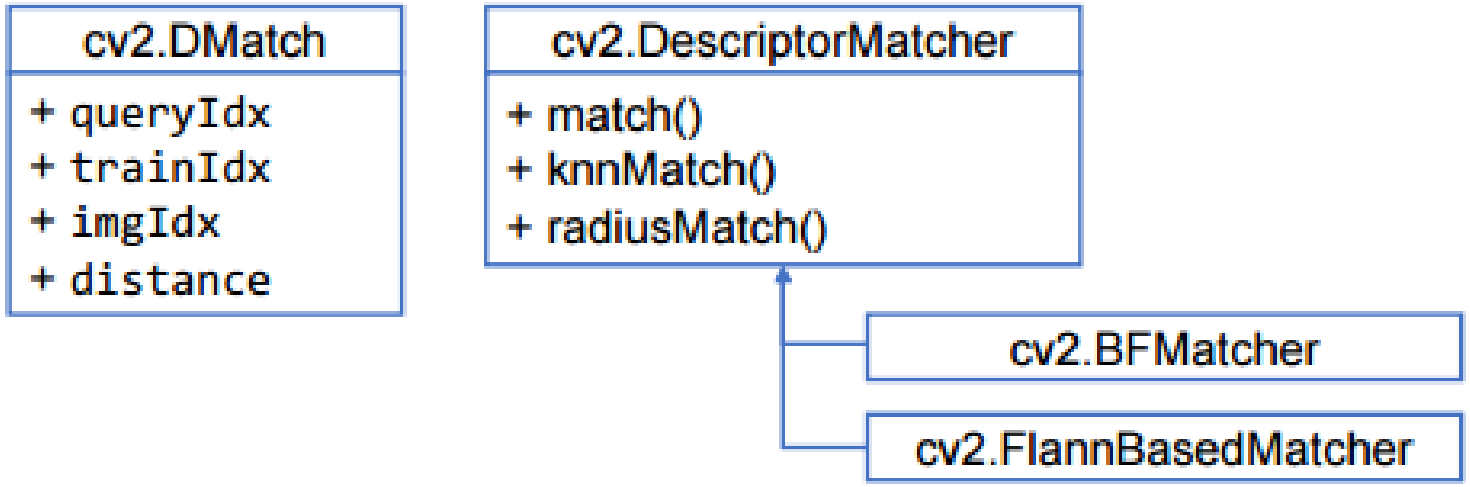
- Distance 멤버 변수 : 두 특징점이 서로 유사하면 distance 값이 0에 가깝고, 서로 다른 특징점이면 distance 값이 크게 나타난다. 유사도라고 생각하면 됨
- queryIdx 멤버 변수 : 1번 이미지의 특징점 번호
- trainIdx 멤버 변수 : 2번 이미지의 특징점 번호
- Imgidx 멤버 변수 : 두 영상이 아닌 여러 개의 영상을 이용할 때 이 값을 이용한다.

- OpneCV 특징점 매칭 클래스 상속 관계



- Dmatch 클래스 객체는 특징점 매칭기 내부에서 생성하여 사용자에게 반환한다. 즉, match(), knnMatch(), radiusMatch() 함수의 반환 결과는 DMatch 객체 리스트이다.
- DescriptorMatcher 클래스 : 매칭기 클래스. Match(), knnMatch(), radiusMatch() 등의 가상 멤버 함수를 가지는 추상 클래스이다. BFMatcher 클래스와 FlannBasedMatchcer클래스는 멤버 함수 기능을 실제로 구현하는 알고리즘 종류들이다.

- OpneCV 특징점 매칭 클래스 상속 관계



- Match()함수는 가장 비슷한 특징자(=포인트) 쌍을 하나 찾는다.
- knnMatch() 함수는 비슷한 특징자 쌍 k개를 찾는다.
- radiusMatch()함수는 지정한 거리 반경 안에 있는 특징자 쌍을 모두 찾아 반환한다.

- BFMatcher클래스

- 첫 번째 영상에 1000개의 특징자가 있고, 두 번째 영상에 2000개의 특징자가 있다면 BFMatcher 방식은 총 2,000,000(1000 x 2000)번의 비교 연산을 수행한다. 모든 특징자 사이의 distance(유사도)를 계산하고, 이 중 가장 distance값이 작은(=유사도가 높은) 특징점을 찾아 매칭하는 방식이다.
- 직관적인 방법이지만 특징점 개수가 늘어날수록 매칭 연산량은 크게 늘어나며, 이러한 경우에는 FlannBasedMatcher 클래스를 사용하는 것이 효율적이다.

- FlannBasedMatcher 클래스

- 모든 특징자들을 전수 조사하기보다 이웃하는 특징자끼리 비교한다. 완전한 distance 최솟값에 매칭을 못할 가능성이 있지만 속도가 빠르다는 장점이 있다.

- drawMatches() 함수

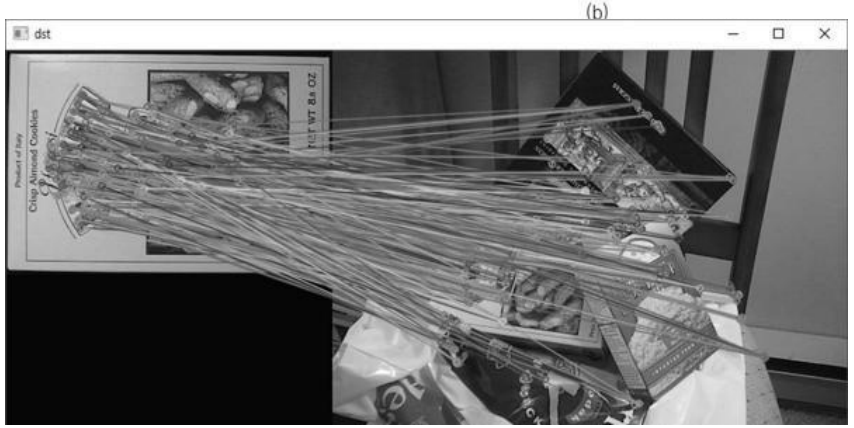
: 두 영상에서 추출한 특징점의 매칭 결과를 한눈에 확인할 수 있도록 두 영상을 가로로 이어 붙이고, 각 영상에서 추출한 특징점과 매칭 결과를 다양한 색상으로 표시한 결과 영상을 생성한다.

```cpp
void keypoint_matching()
{
    Mat src1 = imread("box.png", IMREAD_GRAYSCALE);
    Mat src2 = imread("box_in_scene.png", IMREAD_GRAYSCALE);

    if (src1.empty() || src2.empty()) {
        cerr << "Image load failed!" << endl;
        return;
    }

    Ptr<Feature2D> feature = ORB::create();

    vector<KeyPoint> keypoints1, keypoints2;
    Mat desc1, desc2;
    feature->detectAndCompute(src1, Mat(), keypoints1, desc1);
    feature->detectAndCompute(src2, Mat(), keypoints2, desc2);

    Ptr<DescriptorMatcher> matcher = BFMatcher::create(NORM_HAMMING);

    vector<DMatch> matches;
    matcher->match(desc1, desc2, matches);

    Mat dst;
    drawMatches(src1, keypoints1, src2, keypoints2, matches, dst);

    imshow("dst", dst);

    waitKey();
    destroyAllWindows();
}
```

ORB알고리즘으로 특징점을 검출하고, BFMatcher클래스를 이용하여 서로 유사한 특징점을 찾아 매칭하는 코드.

(a)

(b)



첫번째 영상에서 추출한 모든 특징점에 대해 두번째 영상의 가장 유사한 특징점을 찾아 직선을 그렸기 때문에 매칭 결과가 매우 복잡하게 나타난다.

```cpp
void keypoint_matching()
{
    Mat src1 = imread("box.png", IMREAD_GRAYSCALE);
    Mat src2 = imread("box_in_scene.png", IMREAD_GRAYSCALE);

    if (src1.empty() || src2.empty()) {
        cerr << "Image load failed!" << endl;
        return;
    }

    Ptr<Feature2D> feature = ORB::create();

    vector<KeyPoint> keypoints1, keypoints2;
    Mat desc1, desc2;
    feature->detectAndCompute(src1, Mat(), keypoints1, desc1);
    feature->detectAndCompute(src2, Mat(), keypoints2, desc2);

    Ptr<DescriptorMatcher> matcher = BFMatcher::create(NORM_HAMMING);

    vector<DMatch> matches;
    matcher->match(desc1, desc2, matches);

    Mat dst;
    drawMatches(src1, keypoints1, src2, keypoints2, matches, dst);

    imshow("dst", dst);

    waitKey();
    destroyAllWindows();
}
```

Distance값이 너무 큰 매칭 결과는 무시하고, distance값이 작은 결과만 사용하는 것이 좋다. 따라서 매칭결과를 distance값 기준으로 오름차순 정렬한 뒤 매칭이 잘 된 50개의 매칭 결과만 추출할 수 있다.

추가!

두 영상의 특징점 매칭 결과를 정렬

```cpp
std::sort(matches.begin(), matches.end());
vector<DMatch> good_matches(matches.begin(), matches.begin() + 50);
```

상위 50개 매칭 결과를 good_matches에 저장

```cpp
Mat dst;
drawMatches(src1, keypoints1, src2, keypoints2, good_matches, dst,
    Scalar::all(-1), Scalar::all(-1), vector<char>(),
    DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS);
```

매칭되지 않은 특징점은 화면에 표시X

```cpp
void keypoint_matching()
{
    Mat src1 = imread("box.png", IMREAD_GRAYSCALE);
    Mat src2 = imread("box_in_scene.png", IMREAD_GRAYSCALE);

    if (src1.empty() || src2.empty()) {
        cerr << "Image load failed!" << endl;
        return;
    }

    Ptr<Feature2D> feature = ORB::create();

    vector<KeyPoint> keypoints1, keypoints2;
    Mat desc1, desc2;
    feature->detectAndCompute(src1, Mat(), keypoints1, desc1);
    feature->detectAndCompute(src2, Mat(), keypoints2, desc2);

    Ptr<DescriptorMatcher> matcher = BFMatcher::create(NORM_HAMMING);

    vector<DMatch> matches;
    matcher->match(desc1, desc2, matches);

    Mat dst;
    drawMatches(src1, keypoints1, src2, keypoints2, matches, dst);

    imshow("dst", dst);

    waitKey();
    destroyAllWindows();
}
```

추가!

두 영상의 특징점 매칭 결과를 정렬

```cpp
std::sort(matches.begin(), matches.end());
vector<DMatch> good_matches(matches.begin(), matches.begin() + 50);
```

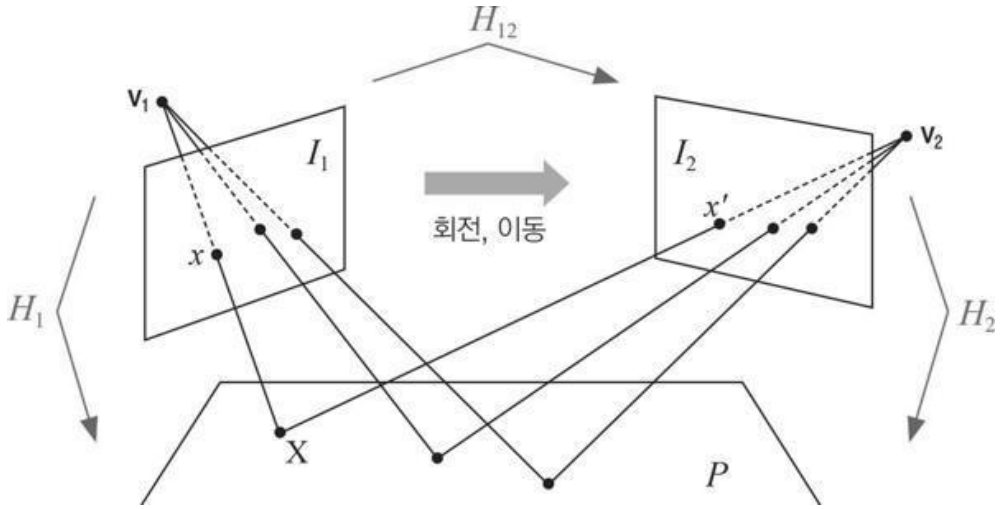
상위 50개 매칭 결과를 good_matches에 저장

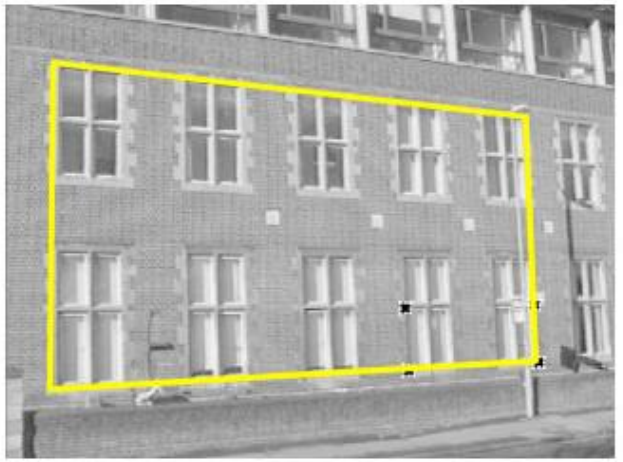
```cpp
Mat dst;
drawMatches(src1, keypoints1, src2, keypoints2, good_matches, dst,
    Scalar::all(-1), Scalar::all(-1), vector<char>(),
    DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS);
```

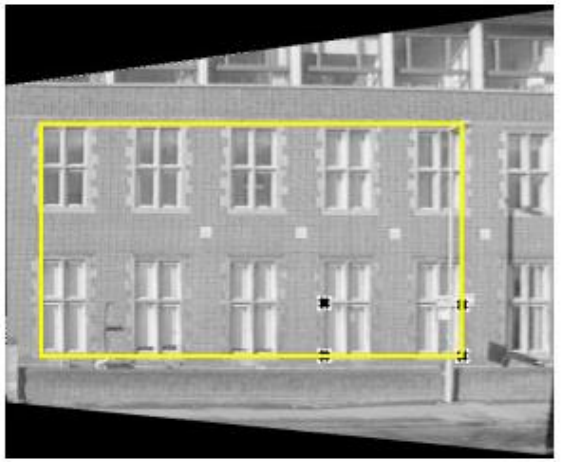
매칭되지 않은 특징점은 화면에 표시X

호모그래피(homography)

: 3차원 공간상의 평면을 서로 다른 시점에서 바라봤을 대 영상 사이의 관계를 나타내는 용어







호모그래피는 3x3실수 행렬로 표현할 수 있다. 4개의 대응되는 점의 좌표 이동 정보가 있으면 호모그래피 행렬을 구할 수 있다. 호모그래피를 계산할 때는 findHomography() 함수를 사용한다.

호모그래피 계산 방법 = RANSAC 알고리즘 사용

```
37      Mat H = findHomography(pts1, pts2, RANSAC);
```

네 모서리 점을 corners1에 저장한 후, 이 점들이 이동하는 위치를 계산하여 corners2에 저장한다.

```
39      vector<Point2f> corners1, corners2;
40      corners1.push_back(Point2f(0, 0));    모서리1
41      corners1.push_back(Point2f(src1.cols - 1.f, 0)); 모서리2
42      corners1.push_back(Point2f(src1.cols - 1.f, src1.rows - 1.f)); 모서리3
43      corners1.push_back(Point2f(0, src1.rows - 1.f)); 모서리4
44      perspectiveTransform(corners1, corners2, H);
```

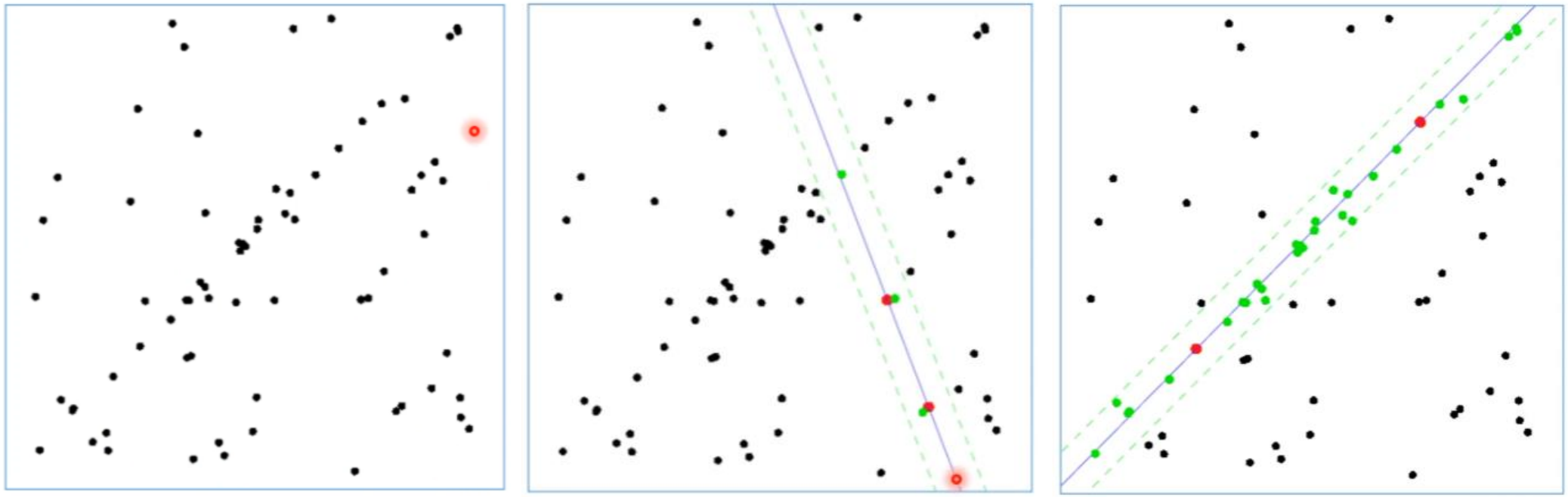
Corner2점들이 위치하는 좌표를 corners_dst에 저장한다.

```
46      vector<Point> corners_dst;
47      for (Point2f pt : corners2) {
48          corners_dst.push_back(Point(cvRound(pt.x + src1.cols), cvRound(pt.y)));
49      }
```

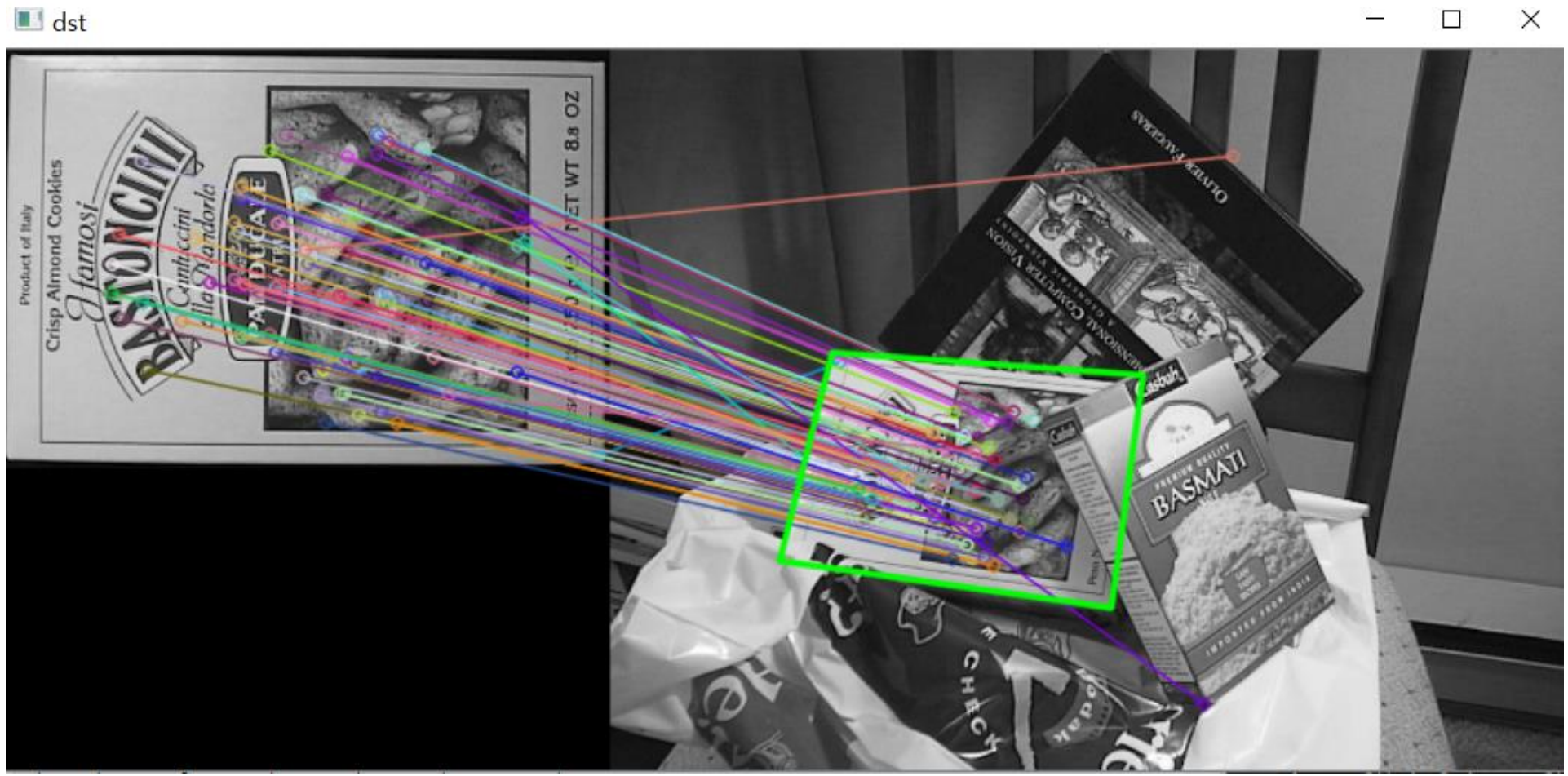
매칭 결과 영상 dst에서 스낵 박스가 있는 위치에 녹색으로 사각형을 그린다.

```
51      polylines(dst, corners_dst, true, Scalar(0, 255, 0), 2, LINE_AA);
52
```

RANSAC알고리즘이란?



임의의 점 2개를 찾고 직선을 구한다. → 마진 안에 들어온 점을 카운팅한다. → 가장 많은 카운팅을 보여준 직선을 구한다.



Box.png 영상의 네 모서리 점의 위치가 어디로 이동하는지 찾아내어 녹색 실선으로 표시했다.

영상 이어 붙이기

: 여러 장의 영상을 서로 이어 붙여 하나의 큰 영상을 만드는 기법. 파노라마 영상이라고 부르기도 한다. 사용할 영상은 서로 일정 비율 이상으로 겹치는 영역이 존재해야 하며, 서로 같은 위치를 분간할 수 있도록 유효한 특징점이 많이 있어야 한다.

- Stitcher클래스

영상 이어붙이기를 위해 입력 영상에서 특징점을 검출하고, 서로 매칭을 수행하여 호모그래피를 구한 다음에, 호모그래피 행렬을 기반으로 입력 영상으로 변형하여 서로 이어붙이는 작업을 수행하고, 이어 붙인 결과가 자연스럽게 보이도록 밝기를 보정하는 블렌딩 처리를 모두 수행하는 클래스

함수 원형 : static Ptr<Stitcher> Stitcher::create(Mode mode = Stitcher::PANORAMA);

(c) SIFT matches 1

(d) SIFT matches 2

(e) RANSAC inliers 1

(f) RANSAC inliers 2

(g) Images aligned according to a homography

① 특징점 검출. 엣지가 있는 부분이 특징점으로 검출된다.

② 양쪽에 공통적으로 검출된 특징점만 남겨둔다.

③ 매칭되는 정보를 이용해 호모그래피 관계를 추출하고 이어 붙인다. 이후 블렌딩 처리를 한다.

```
26      Ptr<Stitcher> stitcher = Stitcher::create();
27
28      Mat dst;
29      Stitcher::Status status = stitcher->stitch(imgs, dst);
30
31      if (status != Stitcher::Status::OK) {
32          cerr << "Error on stitching!" << endl;
33          return -1;
34      }
35
36      imwrite("result.jpg", dst);
37
38      imshow("dst", dst);
39
40      waitKey();
41      return 0;
42  }
```

Stitcher객체를 생성한다.

Imgs에 저장된 입력 영상을 이어 붙여 결과 영상 dst를 생성한다.

영상 이어붙이기가 실패하면 에러 메시지 출력

결과 영상을 result.jpg 파일로 저장