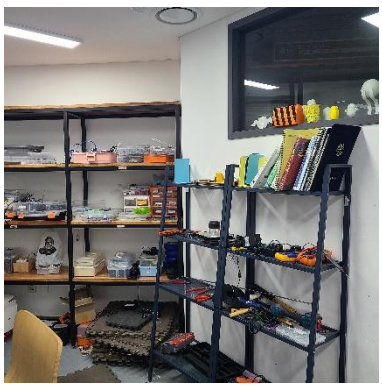
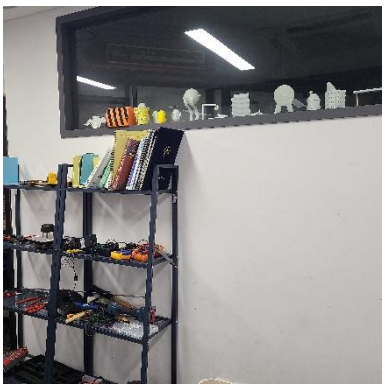
디지털 영상처리 설계 실습#13

12181774 박다혜

1. **직접 촬영한 영상 세장으로 panorama stitching을 수행해 볼 것. (실습 2가지 방법 각각 적용 및 분석)**

```
void ex_panorama() {
    Mat matImage1 = imread("center.jpg", IMREAD_COLOR);
    resize(matImage1, matImage1, Size(), 0.3, 0.3);
    Mat matImage2 = imread("left.jpg", IMREAD_COLOR);
    resize(matImage2, matImage2, Size(), 0.3, 0.3);
    Mat matImage3 = imread("right.jpg", IMREAD_COLOR);
    resize(matImage3, matImage3, Size(), 0.3, 0.3);
    if (matImage1.empty() || matImage2.empty() || matImage3.empty()) {
        cout << "Empty image!₩n";
        exit(-1);
    }
    Mat result;
    flip(matImage1, matImage1, 1);
    flip(matImage2, matImage2, 1);
    result = makePanorama(matImage1, matImage2, 3, 60); //60, 2:50
    flip(result, result, 1);
    result = makePanorama(result, matImage3, 3, 60);

    imwrite("my_ex_panorama_result2.jpg", result);
    resize(result, result, Size(), 0.8, 0.8);
    imshow("ex_panorama_result", result);

    waitKey();
}
```





left / center / right

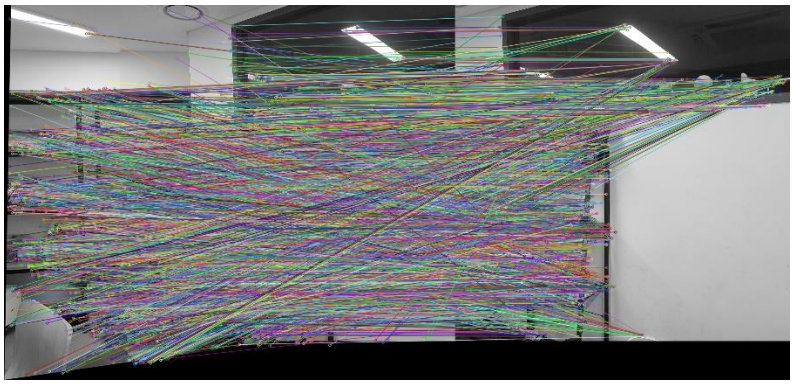
파노라마는 두 장 이상의 영상을 서로 겹쳐서 하나의 영상으로 만드는 기법이다. 이미지 3장을 각각 읽어 온 후 오른쪽 사진에 이어서 파노라마로 붙일 수 있도록 왼쪽과 가운데 이미지를 flip 한다. 그 다음 실습 때 만들었던 makePanorama()에 먼저 왼쪽과 가운데 사진을 넣어 이어 붙인다. 그 결과에 오른쪽 사진을 이어 붙인다. makePanorama()를 자세히 살펴보면

```
Mat makePanorama(Mat img_l, Mat img_r, int thresh_dist, int min_matches) {
    // < gray scale로 변환 >
    Mat img_gray_l, img_gray_r;
    cvtColor(img_l, img_gray_l, CV_BGR2GRAY);
    cvtColor(img_r, img_gray_r, CV_BGR2GRAY);

    // < 특징점(key point) 추출 >
    Ptr<SurfFeatureDetector> Detector = SURF::create(300);
    vector<KeyPoint> kpts_obj, kpts_scene;
    Detector->detect(img_gray_l, kpts_obj);
    Detector->detect(img_gray_r, kpts_scene);

    // < 특징점 시각화 >
    Mat img_kpts_l, img_kpts_r;
    drawKeypoints(img_gray_l, kpts_obj, img_kpts_l, Scalar::all(-1), DrawMatchesFlags::DEFAULT);
    drawKeypoints(img_gray_r, kpts_scene, img_kpts_r, Scalar::all(-1), DrawMatchesFlags::DEFAULT);
```

이미지를 gray scale로 변환하고 두 장의 이미지에서 각각 특징점을 찾는다.

```
// < 기술자(descriptor) 추출 >
Ptr<SurfDescriptorExtractor> Extractor = SURF::create(100, 4, 3, false, true);
Mat img_des_obj, img_des_scene;
Extractor->compute(img_gray_l, kpts_obj, img_des_obj);
Extractor->compute(img_gray_r, kpts_scene, img_des_scene);

// < 기술자를 이용한 특징점 매칭 >
BFMatcher matcher(NORM_L2); //Brute Force 매칭
vector<DMatch> matches;
matcher.match(img_des_obj, img_des_scene, matches);

// < 매칭결과 시각화 >
Mat img_matches;
drawMatches(img_gray_l, kpts_obj, img_gray_r, kpts_scene, matches, img_matches, Scalar::all(-1), Scalar::all(-1), vector<char>(), DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS);
imwrite("img_matches.png", img_matches);
```

그 다음 두 개의 이미지에서 동일한 특징점을 찾아 매칭해 주는데, 이러한 특징점 매칭 기법 중 하나는 Brute force이다. 하나의 이미지에서 발견한 특징점을 다른 또 하나의 이미지의 모든 특징점과 비교해 가장 유사한 것이 동일한 특징점이라고 판별하는 것이다.



하지만 이러한 방법은 너무 많은 매칭 결과를 얻으므로 매칭 거리가 가장 작은 우수한 매칭 결과만을 정제할 필요가 있다.

```
//매칭결과 정제
//매칭거리가 작은 우수한 매칭 결과를 정제하는 과정
//최소 매칭 거리의 3배 또는 우수한 매칭 결과 60이상까지 정제

double dist_max = matches[0].distance;
double dist_min = matches[0].distance;
double dist;
for (int i = 0; i < img_des_obj.rows; i++) {
    dist = matches[i].distance;
    if (dist < dist_min)dist_min = dist;
    if (dist > dist_max)dist_max = dist;
}
printf("max_dist :  %f\n", dist_max); //max는 사실상 불필요
printf("min_dist :  %f\n", dist_min);

vector<DMatch> matches_good;
do {
    vector<DMatch> good_matches2;
    for (int i = 0; i < img_des_obj.rows; i++) {
        if (matches[i].distance < thresh_dist * dist_min)
            good_matches2.push_back(matches[i]);
    }
    matches_good = good_matches2;
    thresh_dist -= 1;
} while (thresh_dist != 2 && matches_good.size() > min_matches);

// < 우수한 매칭 결과 시각화 >
Mat img_matches_good;
drawMatches(img_gray_l, kpts_obj, img_gray_r, kpts_scene, matches_good, img_matches_good, Scalar::all(-1), Scalar::all(-1), vector<char>(), DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS);
imwrite("img_matches_good.png", img_matches_good);
```

매칭 거리 중 가장 작은 최소 매칭 거리를 구한다. 그 다음 thresh_dist=3이었으므로 최소매칭 거리의 3배보다 작은 거리만을 따로 저장하여 시각화 하였다

```
// < 매칭 결과 좌표 추출 >
vector<Point2f>obj, scene;
for (int i = 0; i < matches_good.size(); i++) {
    obj.push_back(kpts_obj[matches_good[i].queryIdx].pt); //img1
    scene.push_back(kpts_scene[matches_good[i].trainIdx].pt); //img2
}

// < 매칭 결과로부터 homograpy 행렬을 추출 >
Mat mat_homo = findHomography(scene, obj, RANSAC); //이상치(outlier)제거를 위해 RANSAC추가

// < Homograpy 행렬을 이용해 시점 역변환 >
Mat img_result;
warpPerspective(img_r, img_result, mat_homo, Size(img_l.cols * 2, img_l.rows * 1.2), INTER_CUBIC); //영상이 잘리는 것을 방지하기 위해 여유공간을 부여

// < 기준 영상과 역변환된 시점 영상 합체 >
Mat img_pano;
img_pano = img_result.clone();
Mat roi(img_pano, Rect(0, 0, img_l.cols, img_l.rows));
img_l.copyTo(roi);
```

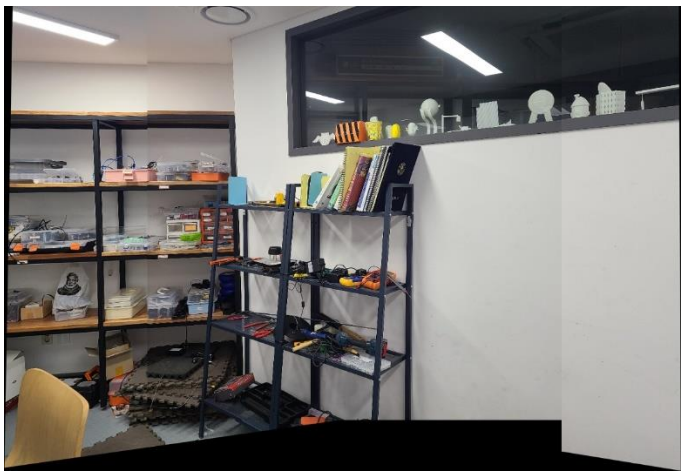
정제된 결과의 좌표들을 각각 추출한다. 시점을 역변환 하기 위해서 매칭 결과로부터 homograpy 행렬을 추출하였다. 이 때 RANSAC을 사용하여 outlier를 제거한다. 마지막으로 기준 영상과 역변환된 시점의 영상을 합친다.

두번째 방법

```
void ex_panorama_simple() {
    Mat img;
    vector<Mat> imgs, imgs2;
    img = imread("left.jpg", IMREAD_COLOR);
    if (img.empty()) {
        cout << "Empty image!₩n";
        exit(-1);
    }
    imgs.push_back(img);

    img = imread("center.jpg", IMREAD_COLOR);
    if (img.empty()) {
        cout << "Empty image!₩n";
        exit(-1);
    }
    imgs.push_back(img);

    Mat result, final_result;
    Ptr<Stitcher> stitcher = Stitcher::create(Stitcher::PANORAMA, false); //try_use_gpu=false
    Stitcher::Status status = stitcher->stitch(imgs, result);
    if (status != Stitcher::OK) {
        cout << "Can't stitch images, error code" << int(status) << endl; //공통되는 영역이 너무 적으면 eror code 1을 출력
        exit(-1);
    }
    imgs2.push_back(result);
    img = imread("right.jpg", IMREAD_COLOR);
    if (img.empty()) {
        cout << "Empty image!₩n";
        exit(-1);
    }
    imgs2.push_back(img);
    status = stitcher->stitch(imgs2, final_result);
    imshow("ex_panorama_simple 3", final_result);
    imwrite("ex_panorama_simple.jpg", final_result);
    waitKey();
}
```

openCV의 stitcher을 활용하여 파노라마 사진을 만들 수 있다. 3장을 이은 파노라마 사진을 만들기 위해서는 먼저 stitcher를 적용하여 두장의 이미지로 파노라마를 만든다. 그 결과에 다시 세번째 이미지를 stitcher를 적용하여 붙이면 된다.

결과



첫번째 : ex_panorama_simple의 결과

오른쪽 위의 장식품은 가지런히 잘 이어졌으며 왼쪽의 선반과 천장은 약간의 차이가 있었지만 전체적으로 각각의 특징이 잘 이어졌다.

 두번째 : ex_panorama_simple의 결과

위쪽 천장 부분의 약간의 왜곡이 발생하였지만 왼쪽 선반과 오른쪽 벽의 이음새가 보이지 않을 만큼 잘 이어졌다. 첫번째 방법은 선반이 정면을 향하고 있었지만 opencv를 이용한 방법은 선반의 시점이 정면을 향하고 있지 않았다.

2. Book1, Book2, Book3이 주어졌을 때 Scene에서 이것들을 찾아 윤곽을 찾아 그려주는 프로그램을 구현할 것

opencv_contrib의 surf_matcher.cpp을 참고하여 작성되었다.

```
struct SURFDetector
{
    Ptr<Feature2D> surf;
    SURFDetector(double hessian = 800.0)
    {
        surf = SURF::create(hessian);
    }
    template<class T>
    void operator()(const T& in, const T& mask, std::vector<cv::KeyPoint>& pts, T& descriptors, bool useProvided = false)
    {
        surf->detectAndCompute(in, mask, pts, descriptors, useProvided);
    }
};

template<class KPMatcher>
struct SURFMatcher
{
    KPMatcher matcher;
    template<class T>
    void match(const T& in1, const T& in2, std::vector<cv::DMatch>& matches)
    {
        matcher.match(in1, in2, matches);
    }
};
```

SURF 특징점 추출 방법을 사용하였다. 실습시간에는 detector와 descriptor를 차례대로 적용하였지만 detectAndCompute()를 사용하여 동시에 진행하였다. 그 다음 추출된 특징들을 매칭한다.

```
static Mat drawGoodMatches(
    const Mat& img1,
    const Mat& img2,
    const std::vector<KeyPoint>& keypoints1,
    const std::vector<KeyPoint>& keypoints2,
    std::vector<DMatch>& matches,
    std::vector<Point2f>& scene_corners_
])
{
    //-- Sort matches and preserve top 10% matches
    std::sort(matches.begin(), matches.end());
    std::vector< DMatch > good_matches;
    double minDist = matches.front().distance;
    double maxDist = matches.back().distance;

    const int ptsPairs = std::min(GOOD_PTS_MAX, (int)(matches.size() * GOOD_PORTION));
]   for (int i = 0; i < ptsPairs; i++)
    {
        good_matches.push_back(matches[i]);
    }
    std::cout << "\nMax distance: " << maxDist << std::endl;
    std::cout << "Min distance: " << minDist << std::endl;

    std::cout << "Calculating homography using " << ptsPairs << " point pairs." << std::endl;

    // drawing the results
    Mat img_matches;

    drawMatches(img1, keypoints1, img2, keypoints2,
        good_matches, img_matches, Scalar::all(-1), Scalar::all(-1),
        std::vector<char>(), DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS);
```

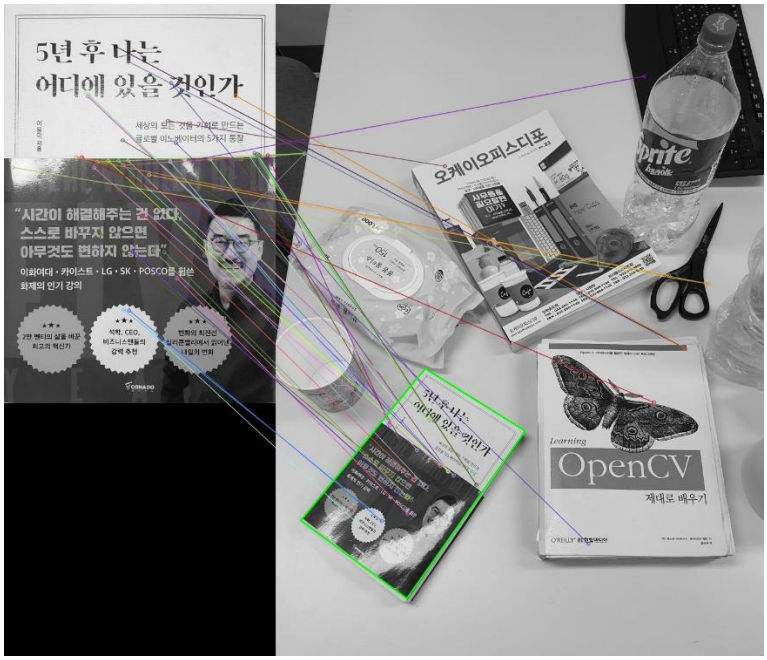
GOOD_PTS_MAX = 50으로 거리의 15%가 50보다 작은 결과만을 정제한다.

```
//-- Localize the object
std::vector<Point2f> obj;
std::vector<Point2f> scene;

for (size_t i = 0; i < good_matches.size(); i++)
{
    //-- Get the keypoints from the good matches
    obj.push_back(keypoints1[good_matches[i].queryIdx].pt);
    scene.push_back(keypoints2[good_matches[i].trainIdx].pt);
}
//-- Get the corners from the image_1 ( the object to be "detected" )
std::vector<Point2f> obj_corners(4);
obj_corners[0] = Point(0, 0);
obj_corners[1] = Point(img1.cols, 0);
obj_corners[2] = Point(img1.cols, img1.rows);
obj_corners[3] = Point(0, img1.rows);
std::vector<Point2f> scene_corners(4);

Mat H = findHomography(obj, scene, RANSAC);
perspectiveTransform(obj_corners, scene_corners, H);

scene_corners_ = scene_corners;

//-- Draw lines between the corners (the mapped object in the scene - image_2 )
line(img_matches,
    scene_corners[0] + Point2f((float)img1.cols, 0), scene_corners[1] + Point2f((float)img1.cols, 0),
    Scalar(0, 255, 0), 2, LINE_AA);
line(img_matches,
    scene_corners[1] + Point2f((float)img1.cols, 0), scene_corners[2] + Point2f((float)img1.cols, 0),
    Scalar(0, 255, 0), 2, LINE_AA);
line(img_matches,
    scene_corners[2] + Point2f((float)img1.cols, 0), scene_corners[3] + Point2f((float)img1.cols, 0),
    Scalar(0, 255, 0), 2, LINE_AA);
line(img_matches,
    scene_corners[3] + Point2f((float)img1.cols, 0), scene_corners[0] + Point2f((float)img1.cols, 0),
```

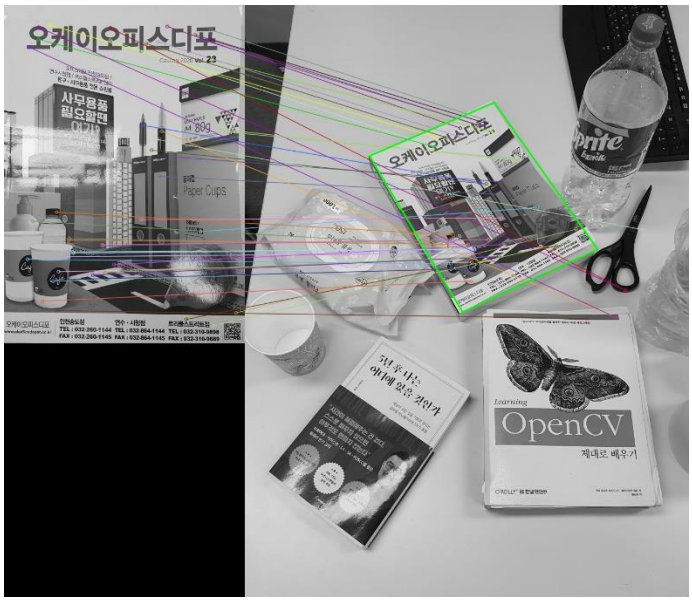
이 정제된 결과로 두개의 이미지에서 공통된 특징점의 좌표를 구하고 findhomograpy()와 RANSAC을 이용해서 outlier를 제거한 homography 행렬을 구하고 시점을 역변환 한다.

마지막으로 코너를 찾고 line()을 이용해 코너를 이은 직선을 그린다.

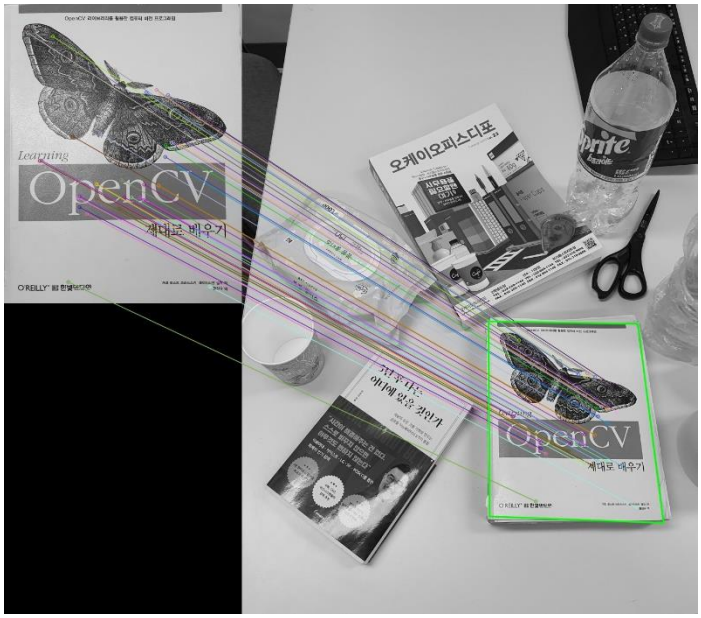
결과



book1



book2



book3