1. Mean-shift OpenCV와 low-level 구현 2개 적용해보고 결과 분석

Mean-shift를 OpenCV에서 주어진 함수와 내가 직접 다른 함수들을 가지고 구현하는 것 두가지를 지금부터 수행해본다.

먼저 OpenCV에서 주어진 함수를 가지고 구현한 Mean-shift함수를 보면 다음과 같다.

```
void exCVMeanShift() {
    Mat img = imread("fruits.png");// fruits 이미지 읽어오기
    if (img.empty()) exit(-1);
    cout << " exCvMeanShift() " << endl;//함수 호출 메시지를 출력

    resize(img, img, Size(256, 256), 0, 0, cv::INTER_AREA);// 이미지를 크기 256x256으로 사이즈 재조정
    imshow("src", img);
    imwrite("exCVMeanShift.jpeg", img);

    pyrMeanShiftFiltering(img, img, 8, 16);
    // OpenCV에서 있는 pyrMeanShiftFiltering 함수를 사용하여 Mean Shift 필터링을 수행
    // 여기서 8은 공간 대역폭(spatial bandwidth)이고 16은 색상 대역폭(color bandwidth)이다

    imshow("Dst", img);
    waitKey();
    destroyAllWindows();
    imwrite("exCVMeanShift_dst.jpeg", img);
}
```

OpenCV의 함수인 pyrMeanShiftFiltering함수를 사용하였는데 이는 OpenCV에서 제공하는 Mean-Shift 필터링 기능을 수행하는 함수이다. Mean-Shift 필터링은 이미지의 평활화와 segmentation 작업을 하며 주로 색상 공간에서 밀집된 영역을 찾고 이러한 영역을 확대하여 객체 경계를 강조하는 역할을 한다. 평활화를 통해 입력 이미지의 노이즈를 줄이고 segmentation을 통해 객체의 경계를 더 잘 구분할 수 있게 한다. 이때 이 함수는 공간대역폭과 색상대역폭을 정해야 하는데 공간대역폭이 클수록 더 넓은 영역의 픽셀들이 고려되고 작으면 더 좁은 영역들의 픽셀이 고려된다. 색상대역폭은 밀집도를 고려하여 평균을 계산하는데 색상대역폭이 크면 더 다양한 색상이 고려되고 작으면 비슷한 색상만 고려한다. 결국 이 함수를 통해 Mean-Shift 알고리즘은 데이터 포인트의 밀집된 영역을 찾고 이러한 밀집된 영역을 중심으로 데이터 포인트를 이동시켜 클러스터가 고정될 때까지 수행한다.

이제 low-level 구현인 My Mean-Shift 함수를 구현해보자. 다음과 같다.

```cpp
//lower level의 My Mean-Shifting 함수
void exMyMeanShift() {
    Mat img = imread("fruits.png");// fruits 이미지 읽어오기
    if (img.empty()) exit(-1);
    cout << " exMyMeanShift() " << endl;//함수 호출 메시지를 출력

    resize(img, img, Size(256, 256), 0, 0, cv::INTER_AREA);// 이미지를 크기 256x256으로 사이즈 재조정
    imshow("src", img);
    imwrite("exMyMeanShift.jpeg", img);

    cvtColor(img, img, cv::COLOR_RGB2Luv);// RGB 색상 공간에서 Luv 색상 공간으로 변환

    MeanShift MSProc(8, 16, 0.1, 0.1, 10);
    // Mean Shift 객체를 생성
    // 공간 대역폭 8, 색상 대역폭 16, 최소 이동값 0.1, 최대 반복 횟수 10으로 설정

    MSProc.doFiltering(img);// Mean Shift 필터링을 수행

    cvtColor(img, img, cv::COLOR_Luv2RGB);// 필터링된 이미지를 Luv 색상 공간에서 다시 RGB 색상 공간으로 변환

    imshow("Dst", img);
    waitKey();
    destroyAllWindows();
    imwrite("exMyMeanShift_dst.jpeg", img);
}
```

직접 만드는 My Mean-Shift함수는 Mean Shift 객체를 생성하고 doFiltering 함수를 호출하여 필터링을 수행한다. Mean Shift과정이 일어나는 부분은 doFiltering함수 부분이기 때문에 이를 제대로 살펴보자

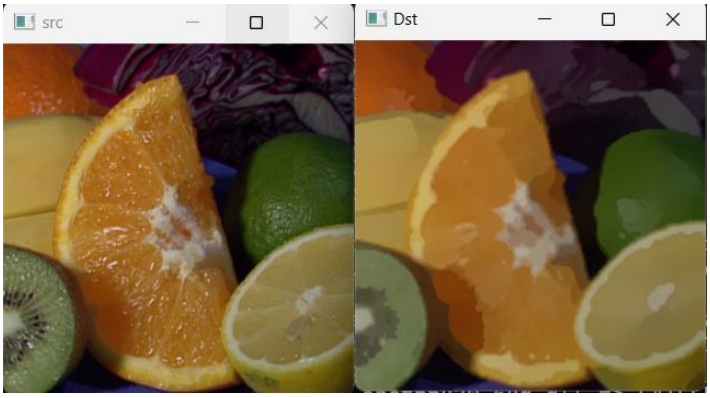
```cpp
// Mean Shift 필터링 수행
void MeanShift::doFiltering(Mat& img) {
    int height = img.rows;
    int width = img.cols;
    split(img, img_split);

    Point5D pt, pt_prev, pt_cur, pt_sum;

    int pad_left, pad_right, pad_top, pad_bottom;
    size_t n_pt, step;

    for (int row = 0; row < height; row++) {
        for (int col = 0; col < width; col++) {
            pad_left = (col - bw_spatial) > 0 ? (col - bw_spatial) : 0;
            pad_right = (col + bw_spatial) < width ? (col + bw_spatial) : width;
            pad_top = (row - bw_spatial) > 0 ? (row - bw_spatial) : 0;
            pad_bottom = (row + bw_spatial) < height ? (row + bw_spatial) : height;

            pt_cur.setPt(row, col, (float)img_split[0].at<uchar>(row, col), (float)img_split[1].at<uchar>(row, col), (float)img_split[2].at<uchar>(row, col));
            step = 0;
            do {
                pt_prev.copyPt(pt_cur);
                pt_sum.setPt(0, 0, 0, 0, 0);
                n_pt = 0;
                for (int hx = pad_top; hx < pad_bottom; hx++) {
                    for (int hy = pad_left; hy < pad_right; hy++) {
                        pt.setPt(hx, hy, (float)img_split[0].at<uchar>(hx, hy), (float)img_split[1].at<uchar>(hx, hy), (float)img_split[2].at<uchar>(hx, hy));
                        if (pt.getColorDist(pt_cur) < bw_color) {
                            pt_sum.accumPt(pt);
                            n_pt++;
                        }
                    }
                }

                pt_sum.scalePt(1.0 / n_pt);
                pt_cur.copyPt(pt_sum);
                step++;
            } while ((pt_cur.getColorDist(pt_prev) > min_shift_color) &&
                (pt_cur.getSpatialDist(pt_prev) > min_shift_spatial) &&
                (step < max_steps));
            img.at<Vec3b>(row, col) = Vec3b(pt_cur.l, pt_cur.u, pt_cur.v);
        }
    }
}
```

doFiltering함수는 이미지 크기와 가져오고 이미지를 채널별로 분리하여 저장하고 현재 픽셀 주변의 패딩 영역을 설정한다. 그 다음 현재 포인트를 설정하고 반복 과정을 위해서 변수를 초기화한다. Mean Shift 반복 과정은 패딩 영역 내의 모든 픽셀에 대해서 색상 거리 기준을 만족하는 포인트들을 더하여 평균을 계산해서 포인트를 이동시킨다. 이렇게 반복을 계속 하다가 결국 바뀌지 않는 지점에서 종료한다. 이렇게 함수를 설정하고 이를

My Mean Shift 함수에서 이용하여 Mean Shift과정을 수행한다.

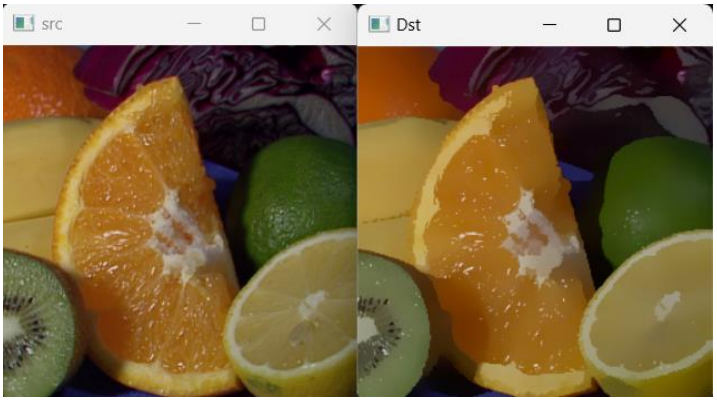
```
int main(int argc, const char* argv[]) {
    exCVMeanShift();
    exMyMeanShift();

    return 0;
}
```

Main 함수로 각각의 함수를 실행하면 다음과 같은 결과를 얻을 수 있다.



이는 OpenCV함수를 사용하여 실행한 결과이다. 두번째 사진이 MeanShift과정을 거쳐서 나온 사진인데 OpenCV에서 주어지는 함수를 사용하였다.

pyrMeanShiftFiltering(img, dst, 8, 16)는 fruits.png 이미지를 필터링하여 dst에 저장하였고 공간 대역폭을 8, 색상 대역폭을 16으로 설정하여 필터링을 수행한 결과 이미지의 평활화와 세그멘테이션이 향상되어 객체 경계가 더욱 뚜렷해지고 스무딩 또한 일어난 것을 알 수 있다.



다음은 직접 doFiltering함수를 통해 구현한 결과이다.

이 함수를 통해 구현한 Mean Shift는 각 데이터 포인트에서 시작하여 중심점을 초기화하고 중심점 주변의 데이터 포인트들의 평균을 계산하여 중심점을 이동한다. 이동된 새로운 중심점에서 다시 평균을 계산하여 중심점을 이동시키는 과정을 반복하는데 결국 중심점의 이동이 미미할 때까지 반복하여 최종 중심점을 구한다. 이런 식으로 알고리즘을 구현하면 결국 이미지의 잡음이 줄어들고 색상이 균일하게 보이고 객체의 클러스터링으로 경계가 명확하게 보인다. 이때 실행을 해보면 OpenCV에서 주어진 함수를 이용한 함수보다 직접 doFiltering하는 함수가 계속 Mean을 이동하고 직접 계산하여서 그런지 실행시

간이 더 오래 걸린다는 것을 실습을 진행해보면 알 수 있다.

2. Grab cut의 처리가 잘 되는 영상과 잘 되지 않는 영상을 찾아 실험해보고 그 이유를 서술할 것(성의 없는 분석, 완전히 틀리거나 의미 없는 분석은 감 점)

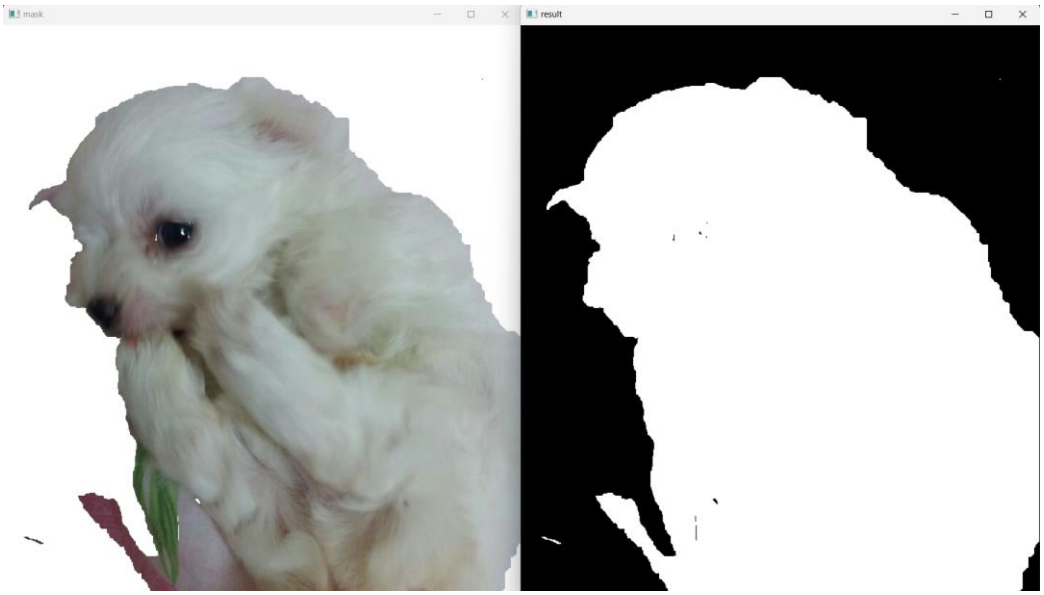
이번 실습에서 하는 grab cut은 그래프 기반의 이미지를 분할해주는 알고리즘으로 사용자가 지정한 사각형 영역 내에서 객체와 배경을 분리해준다. 알고리즘을 통해 이미지의 전경을 추출하고 객체가 포함된 사각형 영역을 좌표를 통해 지정하면 GrabCut 함수가 해당 영역을 바탕으로 객체 배경을 분리해준다.

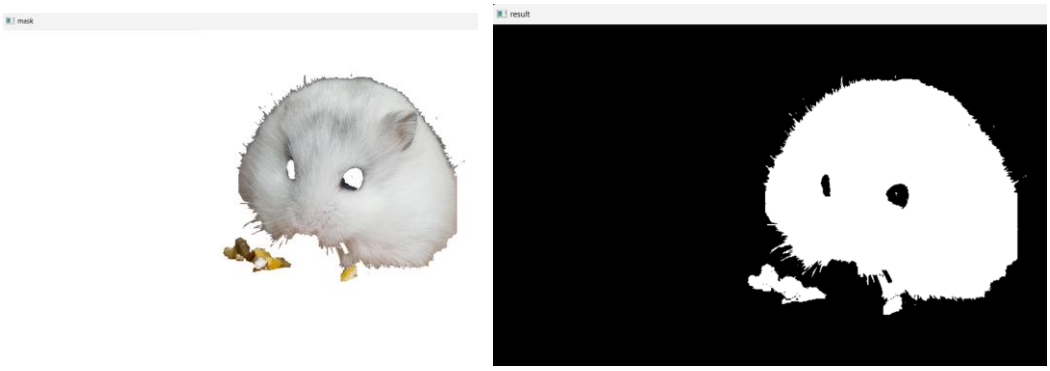
```
void DogGrabCut() {
    Mat img = imread("MyDog.jpg", 1);
    imshow("src_img", img);

    Rect rect = Rect(Point(29.3, 55.56), Point(719.3, 952.77));
    Mat result, bg_model, fg_model;
    grabCut(img, result, rect, bg_model, fg_model, 5, GC_INIT_WITH_RECT);
    compare(result, GC_PR_FGD, result, CMP_EQ);
    Mat mask(img.size(), CV_8UC3, cv::Scalar(255, 255, 255));
    img.copyTo(mask, result);

    imshow("mask", mask);
    imshow("result", result);

    waitKey(0);
}


void HamgrabCut() {
    Mat img = imread("hamster.jpg", 1);
    imshow("src_img", img);

    Rect rect = Rect(Point(374, 76), Point(802, 433));
    Mat result, bg_model, fg_model;
    grabCut(img, result, rect, bg_model, fg_model, 5, GC_INIT_WITH_RECT);
    compare(result, GC_PR_FGD, result, CMP_EQ);
    Mat mask(img.size(), CV_8UC3, cv::Scalar(255, 255, 255));
    img.copyTo(mask, result);

    imshow("mask", mask);
    imshow("result", result);

    waitKey(0);
}
```

```
void CatgrabCut() {
    Mat img = imread("cat1.png", 1);
    imshow("src_img", img);

    Rect rect = Rect(Point(375, 30), Point(632, 464));
    Mat result, bg_model, fg_model;
    grabCut(img, result, rect, bg_model, fg_model, 5, GC_INIT_WITH_RECT);
    compare(result, GC_PR_FGD, result, CMP_EQ);
    Mat mask(img.size(), CV_8UC3, cv::Scalar(255, 255, 255));
    img.copyTo(mask, result);

    imshow("mask", mask);
    imshow("result", result);

    waitKey(0);
}




int main() {


    //DogGrabCut();
    //HamgrabCut();
    CatgrabCut();
    return 0;
}
```

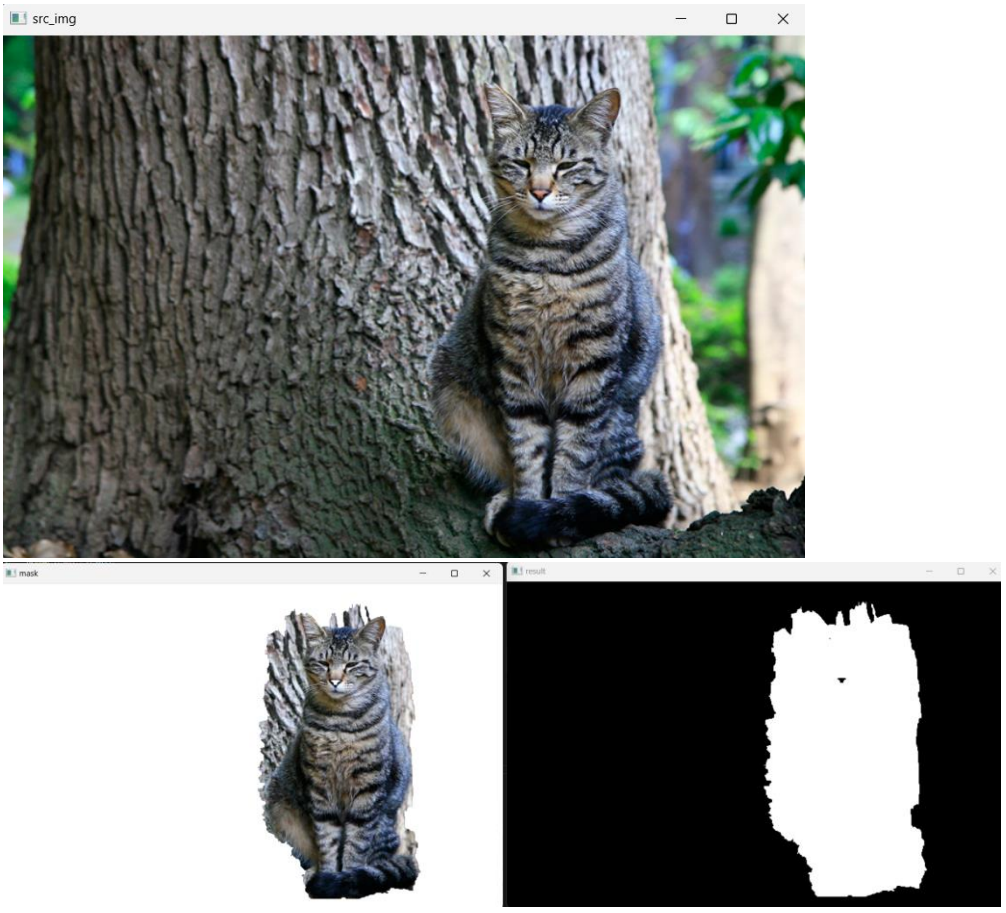
이와 같이 세개의 grabcut함수를 만들어봤다. 이 함수는 객체의 경계를 구분하는데 주변 화소와 RGB값을 비교해서 유사하면 같은 segment로 하고 화소값의 차이가 크면 다른 segment로 한다. 이렇게 다르게 분류하여 cluster가 아닌 부분을 없애주는 것이다.
이때 함수안에는 실습에서 진행할 때도 사용한 OpenCV grabcut함수를 쓴다. 이미지에서 좌표의 설정을 통해 내가 원하는 범위의 직사각형을 설정하였고 이렇게 설정한 직사각형 범위 안에서 전경과 배경의 분리가 일어나게 한다.

처음에는 강아지 사진으로 잘 분리될 것이라 생각했고 두번째는 실습때 사용한 사진으로 이 사진도 잘 분리될 것이라 생각했다. 하지만 마지막 사진은 화소값의 차이가 크지 않고 경계가 모호하기 때문에 분류하는 것이 어려울 것이라 생각했다.
실제 실습 결과를 살펴보면





첫번째와 두번째 사진에서 왼쪽사진이 mask이고 오른쪽 사진이 결과인데 설정한 사각형 내에서 객체의 경계가 주변의 화소값과 차이가 크기 때문에 mask가 제대로 추출된 것을 결과를 통해 확인할 수 있다.



마지막 사진은 앞에 두사진들과는 달리 제대로 고양이의 모양이 온전히 결과로 나오지 않았다. 이는 사진속에 나무 부분에서 나무와 고양이를 제대로 구분하지 못해서 출력되기를 원했던 고양이의 모습을 제대로 출력하지 못한 것이다. 왜냐하면 고양이와 나무와의 화소 값의 차이가 크지 않아서 고양이의 모습에서의 경계를 제대로 알아채지 못했다.

이렇게 세개의 사진을 통해 실습해본 결과 grabcut이 잘 처리되는 사진과 처리되지 않는 차이는 객체와 배경의 구분이 매우 잘 드러나느냐 아니냐의 차이이다. 객체와 배경의 화

소값의 차이가 뚜렷할 경우에는 즉, 색조값도 다르고 밝기 값의 차이도 명확할 시에는 grabcut이 제대로 일어나지만 아니고 모호할 시에는 grabcut이 객체를 제대로 구분하지 못해서 올바르지 않은 결과값을 준다는 것을 알 수 있다.