Computer Vision Assignment Report

Assignment 3 – Hough Trasform

2019742071 – 이민규

1. 과제 개요

1. : **캐니 에지 검출 알고리즘**

- 가우시안 필터 적용
- 밝기값의 기울기, 크기, 방향계산
- Non-maximum suppression
- Two-level threshold
- edge tracking

2. : **허프변환**

- (a, b, r) 투표방법
- 그래디언트 기반 방법

>> 입력이미지에 캐니 에지 검출 알고리즘을 적용해 엣지를 검출하고 허프변환을 사용해 원을 검출

2. 과제 수행 방법

1. : 캐니 에지 검출 알고리즘

1-1: Gaussian_Blur()

```
void Gaussian_Blur(const cv::Mat& InputImage, cv::Mat& OutputImage, int kernelSize)
{
    OutputImage = cv::Mat::zeros(InputImage.size(), InputImage.type());
    int radius = kernelSize / 2;
    double sigma = 1; // 표준편차

    // 가우시안 분포를 따르는 값들로 2차원 필터를 만든다.
    std::vector<std::vector<double>> kernel(kernelSize, std::vector<double>(kernelSize));
    double value_sum = 0.0; // 가우시안 커널의 모든 요소의 합을 계산 (정규화때 사용)
    for (int x = -radius; x <= radius; ++x)
    {
        for (int y = -radius; y <= radius; ++y)
        {
            double value = exp(-(x * x + y * y) / (2.0 * sigma * sigma));
            kernel[x + radius][y + radius] = value;
            value_sum += value;
        }
    }

    // 정규화 (커널의 합이 1이 되어 블러를 적용할경우, 이미지의 밝기를 유지시킴)
    for (int i = 0; i < kernelSize; ++i)
    {
        for (int j = 0; j < kernelSize; ++j)
            kernel[i][j] /= value_sum;
    }
```

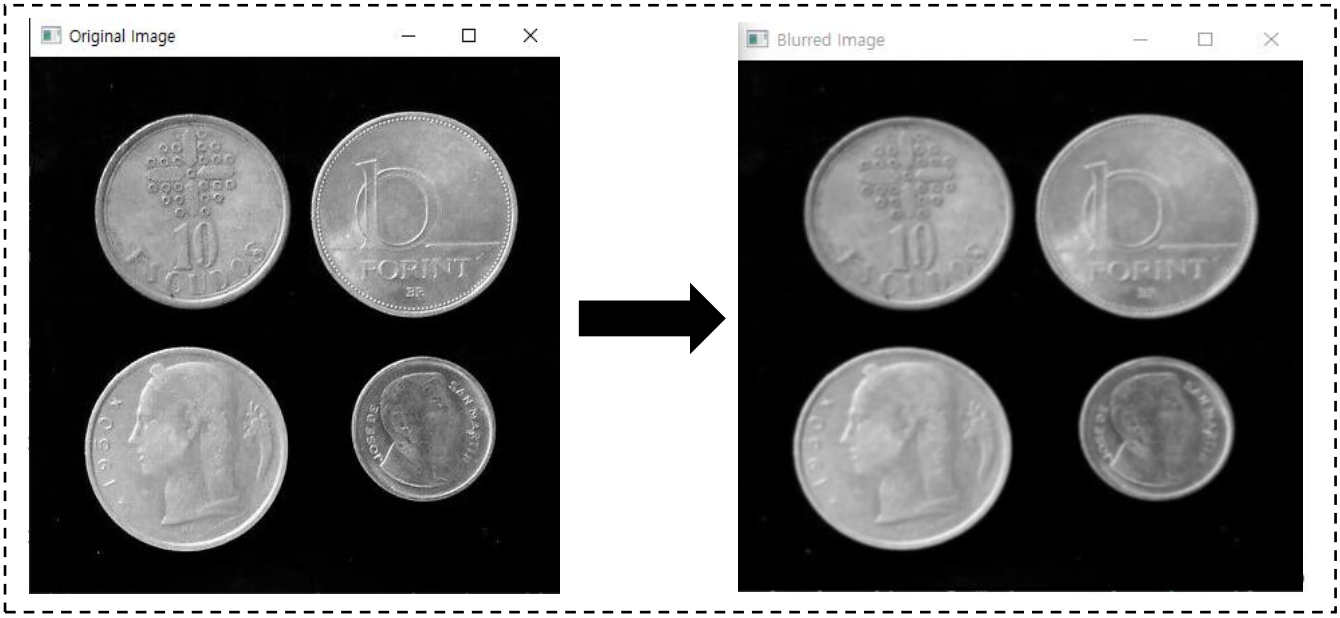
```
    // 가우시안 적용 (컨볼루션)
    for (int i = radius; i < InputImage.rows - radius; ++i)
    {
        for (int j = radius; j < InputImage.cols - radius; ++j)
        {
            double blur_sum = 0.0;
            for (int x = -radius; x <= radius; ++x)
            {
                for (int y = -radius; y <= radius; ++y)
                    blur_sum += InputImage.at<uchar>(i + x, j + y) * kernel[x + radius][y + radius];
            }
            OutputImage.at<uchar>(i, j) = static_cast<uchar>(blur_sum);
        }
    }
}
```

< 그림1. Gaussian_Blur()>

이미지에 가우시안 필터를 적용하면 노이즈가 제거되어 이미지의 엣지를 검출하기에 유리해진다.

이때, 정규화는 이미지의 밝기를 유지시키기 위해서 필요하다.

가우시안필터의 이미지상 적용은 마치 CNN에서 필터가 슬라이딩 하듯이 적용된다.



< 그림2. 가우시안 블러 적용 결과 >

1-2: Mynon_Max_Suppression ()

```
int sobel_x[3][3] =
{
    {-1, 0, 1},
    {-2, 0, 2},
    {-1, 0, 1}
};
int sobel_y[3][3] =
{
    {-1, -2, -1},
    { 0,  0,  0},
    { 1,  2,  1}
};

for (int i = 1; i < HEIGHT - 1; i++)
{
    for (int j = 1; j < WIDTH - 1; j++)
    {
        // 소벨필터를 적용하는 것은 편미분을 하는 것과 같은 효과
        double gradx = 0;
        double grady = 0;
        for (int x = -1; x <= 1; x++)
        {
            for (int y = -1; y <= 1; y++)
            {
                gradx = gradx + sobel_x[1 - x][1 - y] * input_copy[(i + x) * WIDTH + j + y];
                grady = grady + sobel_y[1 - x][1 - y] * input_copy[(i + x) * WIDTH + j + y];
            }
        }
```

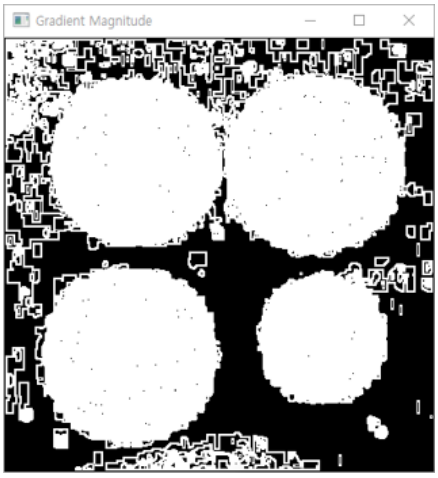
< 그림3. Mynon_Max_Suppression()의 일부 >

본 함수에서는 이미지의 그래디언트와 그래디언트의 방향, 그리고 non maximum suppression까지 한번의 함수로 구현하였다. 그래디언트를 구하는 것은 그림3에서와 같이 소벨필터를 이미지에 슬라이딩하여 gx와 gy, 즉 편미분 값을 먼저 구하고 다음의 공식에 따라 지정된 픽셀에서의 그래디언트를 구한다.

$$\text{value} = \exp\left(-\frac{x^2+y^2}{2\sigma^2}\right)$$

그래디언트의 방향은 아크탄젠트로 구하였다.

이때, 소벨필터를 적용할 때 이미지의 가장자리 픽셀에서는 3 X 3 필터 연산을 할 수 없으므로 인접한 픽셀의 그래디언트 값을 적용한다. 그래디언트의 모양은 다음과 같이 나타난다.

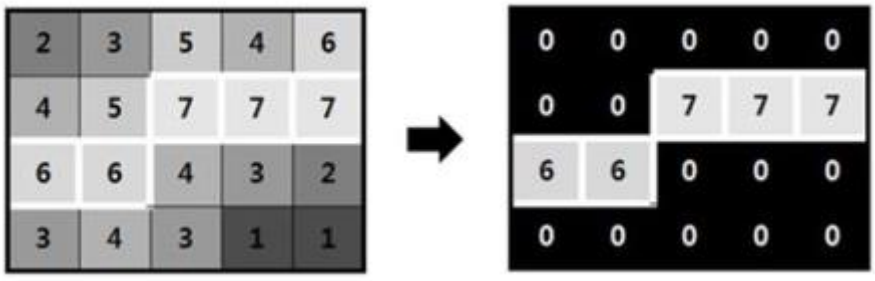


< 그림4. 그래디언트의 모양 >

non-maximum-suppression의 적용은 그래디언트의 두께를 줄여주는 역할을 한다.

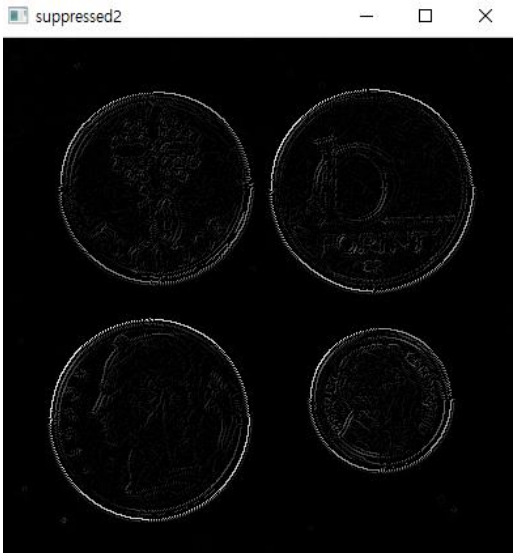
이는 추후 엣지를 검출할 때, 검출되는 엣지의 두께를 줄여주는 역할을 한다.

각 픽셀에서 상하좌우대각선 방향으로 그래디언트의 크기를 비교한다. 만일 주변 그래디언트가 더 크다면, 지정한 픽셀의 그래디언트의 값을 0으로 만든다.



이 과정에서도 정규화가 필요하다. 그래디언트 값은 255, 즉 이미지 밝기 값의 최대 값을 넘을 수 있기 때문이다.

따라서 그 값을 0~255의 범위로 변환하였다.



< 그림5. MynonMaxSuppression2() 적용결과>

1-3: DoubleThreshold_EdgeTracking()

```
void DoubleThresholdAndEdgeTracking(cv::Mat& suppressed, cv::Mat& edges, float lowThreshold, float highThreshold)
{
    edges = cv::Mat::zeros(suppressed.size(), CV_8UC1);
    for (int i = 1; i < suppressed.rows - 1; i++)
    {
        for (int j = 1; j < suppressed.cols - 1; j++)
        {
            if (suppressed.at<uchar>(i, j) >= highThreshold)
                edges.at<uchar>(i, j) = 255; // 이미지의 픽셀값이 highThreshold이상이면 edge로 설정 (255)
            else if (suppressed.at<uchar>(i, j) >= lowThreshold)
            {   // 현재 픽셀 값이 lowThreshold 이상 highThreshold 미만이면, 인접한 픽셀 중 highThreshold를 넘는 픽셀이 있는지 찾는다
                // 만일 하나라도 highThreshold를 넘는 픽셀이 있으면 현재픽셀은 확인한 경계선과 연결된 약한 경계선이다.
                bool has_strong_neighbor = false;
                for (int dx = -1; dx <= 1; dx++)
                {
                    for (int dy = -1; dy <= 1; dy++)
                    {
                        if (suppressed.at<uchar>(i + dx, j + dy) >= highThreshold)
                        {
                            has_strong_neighbor = true;
                            break;
                        }
                    }
                    if (has_strong_neighbor)
                        break;
                }
                if (has_strong_neighbor)
                    edges.at<uchar>(i, j) = 255;
            }
        }
    }
}
```

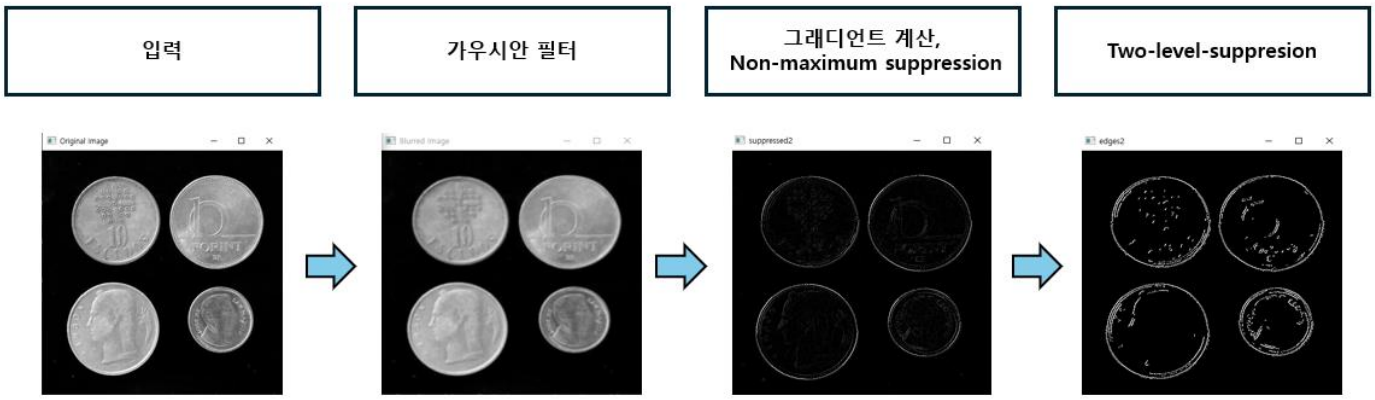
< 그림6. DoubleThreshold_EdgeTracking() 함수정의 >

본 함수에서는 2개의 임계값을 입력으로 받아 이를 엣지구분의 기준으로 사용한다.

높은 임계값은 강한 에지를 결정하고 낮은 임계값은 강한 에지와 연결되어있는 경우만 에지로 판단한다.

이와 같은 과정은 노이즈를 걸러내고 진짜 에지만 남겨두는 효과가 있다.

< 그림7. 최종 엣지 검출결과(左 : 직접 구현한 함수, 右 : 내장 canny 함수)>

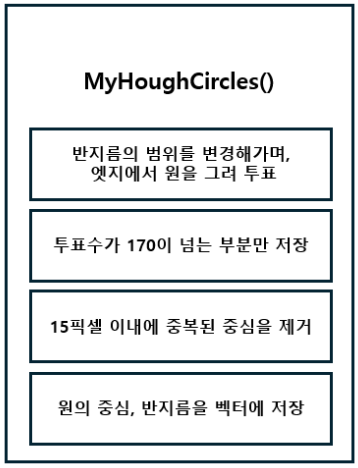


위와 같이 최종엣지 검출결과, 의도한 엣지가 검출되었음을 확인할 수 있다.

실제 캐니 엣지 함수보다 파라미터를 조절하여 원 안의 엣지는 감지되지 않게 하였다. 이는 추후 원을 검출할 때 더욱 유리하게 작동할 것이다.

2. : 허프변환

2-1: MyHoughCircles()



```cpp
void MyHoughCircles(const cv::Mat& edges, std::vector<cv::Vec3f>& circles, int minRad, int maxRad, int minVotes)
{
    std::vector<int> center_x, center_y, circle_rad;

    for (int rad = minRad; rad <= maxRad; rad++) // 반지름의 범위를 알면 투표시간을 단축시킬 수 있다.
    {
        cv::Mat overlap_circle = cv::Mat::zeros(edges.size(), CV_32F); // 결과저장용
        for (int i = 0; i < edges.rows; i++)
        {
            for (int j = 0; j < edges.cols; j++)
            {
                if (edges.at<uchar>(i, j) == 255) // EDGE가 있는 점을 찾는다
                {
                    for (int c = 0; c < 360; c++) // 모든 방향에 대하여 연산
                    {   // 주어진 좌표 i, j를 중심으로 반지름이 rad인 원의 둘레에 위치한 점의 좌표 (a, b)를 계산
                        int a = cvRound(i - rad * std::cos((c * CV_PI) / 180.0)); // 1도 = pi / 180 rad
                        int b = cvRound(j - rad * std::sin((c * CV_PI) / 180.0));
                        if (a >= 0 && a < edges.rows && b >= 0 && b < edges.cols) // 범위 제한
                            overlap_circle.at<float>(a, b)++;
                    }
                }
            }
        }

        int temp = 170; //투표수의 임계점
        for (int i = 0; i < overlap_circle.rows; i++)
        {
            for (int j = 0; j < overlap_circle.cols; j++)
            {
                if (overlap_circle.at<float>(i, j) > temp)
                {
                    center_x.push_back(i); // 전역변수벡터의 끝에 중심의 좌표와 반지름을 append
                    center_y.push_back(j);
                    circle_rad.push_back(rad);
                }
            }
        }
    }

    int k = center_x.size(); // 중복된 중심좌표는 제거필요, 총 중심좌표갯수
    for (int i = 0; i < center_x.size(); i++)
    {
        int checker_x = center_x[i];
        int checker_y = center_y[i];
        for (int j = 0; j < k; j++) // 다른 중심좌표와 비교한다.
        {
            if (i != j)
            {
                if ((center_x[j] + 15 >= checker_x && center_y[j] + 15 >= checker_y) && (center_x[j] - 14 <= checker_x
                { // 15픽셀 이내에 있으면 두 원의 중심이 매우 가까운 것으로 판단하여 제거
                    center_x.erase(center_x.begin() + j);
                    center_y.erase(center_y.begin() + j);
                    circle_rad.erase(circle_rad.begin() + j);

                    k--; // 갯수 감소
                    j = 0; // 초기화
                    i = 0;
                    checker_x = center_x[i];
                    checker_y = center_y[i];
                }
            }
        }
    }

    for (size_t i = 0; i < center_x.size(); i++)
        circles.push_back(cv::Vec3f(center_y[i], center_x[i], circle_rad[i]));
    // circles에는 원의 중심좌표, 반지름이 벡터형식으로 저장
}
```

< 그림8. MyHoughCircles() >

허프변환으로 원을 검출하는 함수를 작성하였다. 원을 반지름, 원의 중심의 두 요소로 변환하여 원을 검출한다.

기존 캐니 엣지 알고리즘의 결과로 edge의 이미지를 검출하였다. 이 엣지점 마다 360도로 돌면서, 일정 반지름길이의 원을 그린다. 이렇게 하면 여러 개의 원이 동전의 엣지를 따라 둘러져있는 형태가 될 것이다. 이 새로 그린 원의 점들이 하나하나 원의 중심의 후보에 대한 투표이다. 이 과정은 사용자가 최소 반지름과 최대 반지름의 범위를 주면 그 범위 안에서 계산이 된다. 이러한 과정이 시간이 오래걸리는 만큼, 반지름의 범위를 줄여놓으면 실행속도는 빨라질 것이다.

본 함수에서 원의 중심으로 정하는 투표값의 임계점은 170으로 정하였으며 이는 실험적으로 여러 번 테스트하여 가장 정답과 근사한 값으로 결정하였다. 임계점을 넘은 점은 원의 중심으로 결정되어 그 좌표가 벡터에 저장된다.

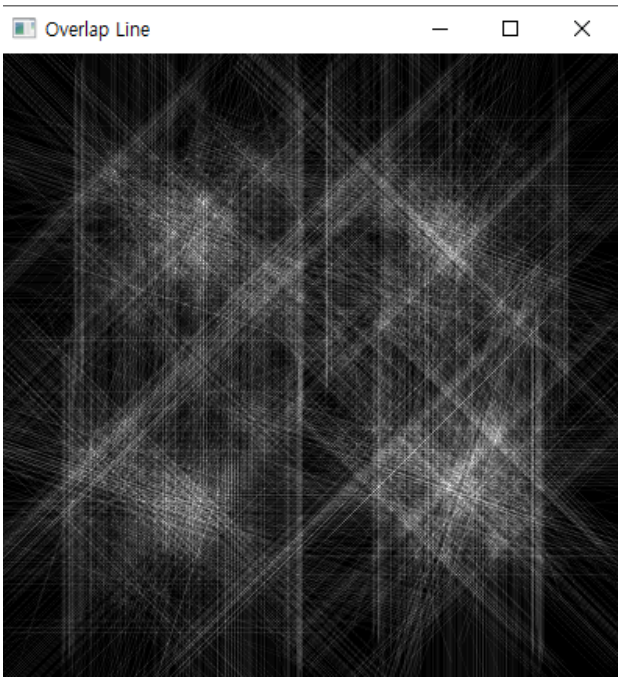
이때, 원의 특성상 실제 원의 중심 부근에 투표가 중첩이 될 가능성이 있다. 이러한 케이스를 제거하기 위해 일정 픽셀내의 겹쳐진 원의 중심을 제거하였다.

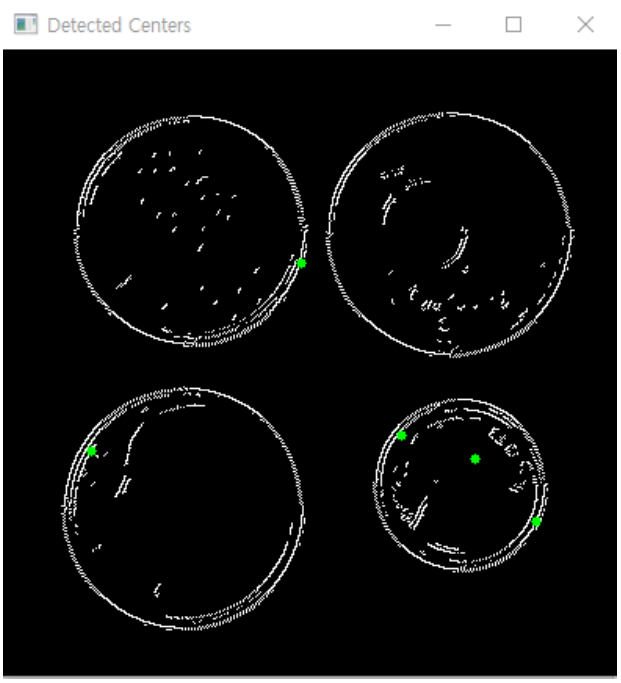
< 그림9. MyHoughCircles() >

위와 같이 원의 검출결과는 정답과 유사함을 확인 가능하다.

2-2 HoughGradientVoting () 그래디언트 기반 허프변환함수 (구현실패)

```
void HoughGradientVoting(const cv::Mat& edges, std::vector<int>& center_x2, std::vector<int>& center_y2)
{
    cv::Mat overlap_line = cv::Mat::zeros(edges.size(), CV_32F); // 투표용 행렬 초기화

    for (int i = 1; i < HEIGHT - 1; ++i)
    {
        for (int j = 1; j < WIDTH - 1; ++j)
        {
            if (edges.at<uchar>(i, j) == 255) // 에지 점 찾기
            {
                // Sobel 필터를 이용하여 그래디언트 계산
                float grad_x = (edges.at<uchar>(i + 1, j - 1) + 2 * edges.at<uchar>(i + 1, j) + edges.at<uchar>(i + 1, j + 1)) -
                    (edges.at<uchar>(i - 1, j - 1) + 2 * edges.at<uchar>(i - 1, j) + edges.at<uchar>(i - 1, j + 1));

                float grad_y = (edges.at<uchar>(i - 1, j + 1) + 2 * edges.at<uchar>(i, j + 1) + edges.at<uchar>(i + 1, j + 1)) -
                    (edges.at<uchar>(i - 1, j - 1) + 2 * edges.at<uchar>(i, j - 1) + edges.at<uchar>(i + 1, j - 1));

                // 그래디언트 방향 계산
                float angle = std::atan2(grad_y, grad_x);

                // 투표 (직선을 따라)
                for (int r = 0; r <= 70; ++r) // 직선그리는 모양을 결정
                {
                    int x = i + r * std::cos(angle);
                    int y = j + r * std::sin(angle);

                    if (x >= 0 && x < HEIGHT && y >= 0 && y < WIDTH)
                    {
                        overlap_line.at<float>(x, y) += 1;
                    }
                }
```





< 그림10. 그래디언트 직선의 모양, 결과 >

HoughGradientVoting 함수로 각 edge의 점에서 그래디언트와 방향을 기반으로 길이가 70인 직선을 그리고 이를 확인한 결과 위와같이 경향성이 있게 모이는 것을 보았다. 그러나, 그 경향성이 너무 넓게 퍼져있어 center를 검출하는 성능이 떨어지는 모습을 보였다. 결론적으로, 그래디언트 기반의 원검출은 실패하였다.

3. 결과분석

		1번 원	2번 원	3번 원	4번 원
Ground truth	원의 중심	(116, 114)	(284, 116)	(118, 290)	(292, 278)
	반지름	70	76	75	56
Hough transform results	원의 중심	**(120, 114)**	**(286, 118)**	**(115, 292)**	**(292, 276)**
	반지름	**73**	**76**	**76**	**54**

```
Center: (292, 276), Radius: 54
Center: (120, 114), Radius: 73
Center: (286, 118), Radius: 76
Center: (115, 292), Radius: 76
```

그래디언트 방식으로 구현하지는 못하였으나 (a, b, r) 기반 투표방식은 상당히 높은 정확도로 원을 검출해 내었다. 그래디언트 방식이 실패한 것은 그래디언트와 그 방향을 구하는 것에서 오류가 있었을 것으로 생각된다. 그림 10에서 확인 할 수 있듯이, 직선이 생각보다 한점으로 모이지 않아서 중심을 정하기가 매우 어려웠다.

	(a, b, r) 투표 방법	그래디언트 기반 방법
메모리 사용량	**많음**	**(a, b, r) 보다는 적음**
실행 시간	**오래걸림, 5초**	**(a, b, r) 보다는 빠름**
결과	**높은 정확도**	-
연산량	**많음**	**(a, b, r) 보다는 적음**

기본적으로 (a, b, r)투표방법은, 각 에지에서 반복문으로 일정범위의 반지름에 대해 그 반지름의 크기를 가지는 모든 원을 그리고 그 원이 지나는 모든 점에 대해서 투표한다. 이는 매우 큰 연산량을 필요로한다. 이에 반해 그래디언트 기반방법은 중심후보를 찾는 방법에서 그래디언트를 사용한다. 따라서 연산량이 상대적으로 적다.

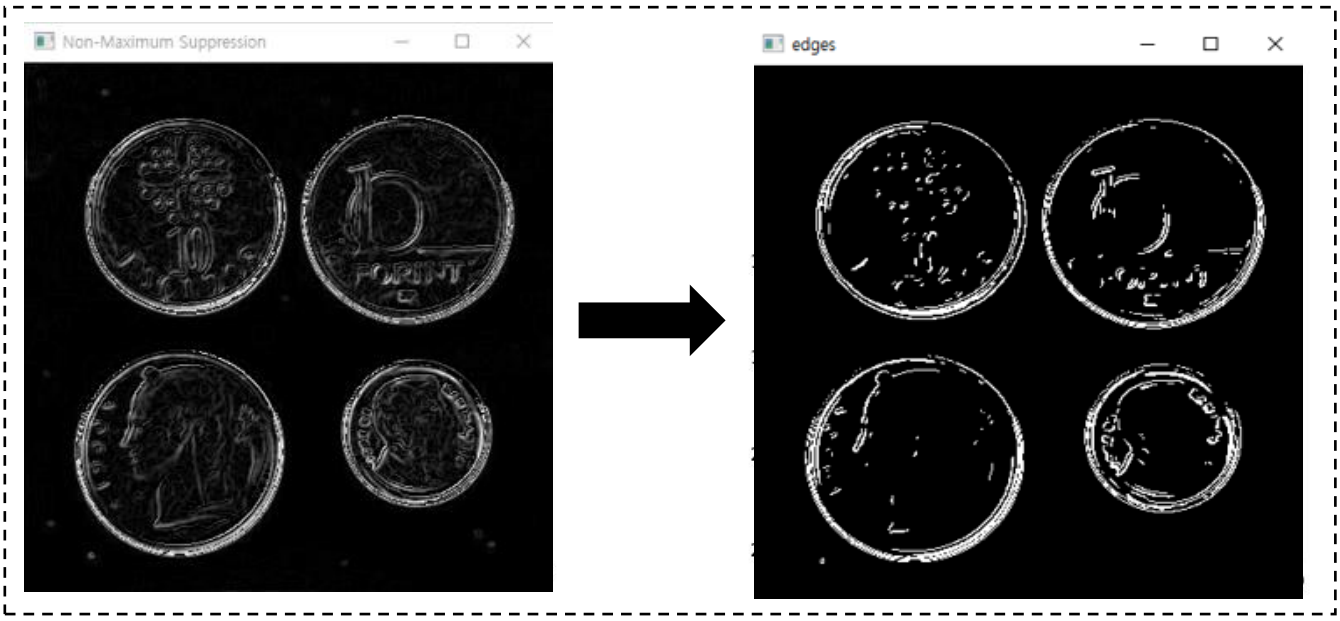
따라서 연산량이 적은 그래디언트 기반방법은 실행시간에서 또한 유리하다.

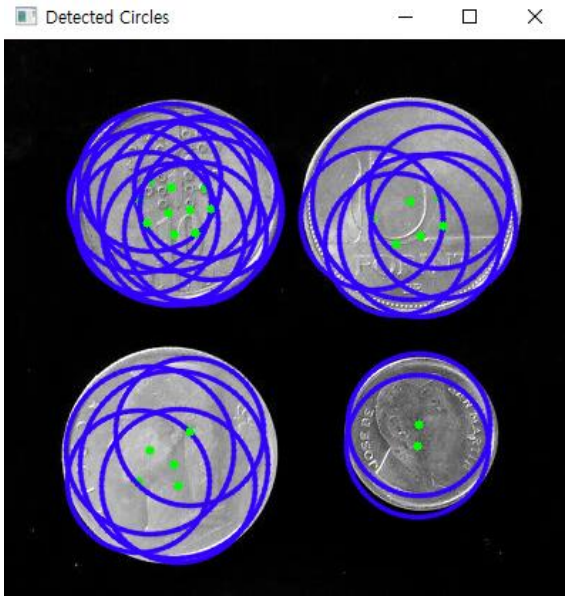
메모리 사용량은 투표를 위한 배열의 크기에 따라서, (a, b, r)투표방법이 더 많은 메모리가 필요하다. 이는 투표공간이 3차원이기 때문이다. 그러나 그래디언트 기반 방법은 한번의 투표로 원의 정보를 얻어내므로 메모리 사용량도 적다.

```
std::time_t time_func_call = std::time(nullptr); // MyHoughCircles 시작 시간 측정
MyHoughCircles(edges2, circles, 20, 100); // (a, b, r)투표 기반의 허프변환
std::time_t time_func_close = std::time(nullptr); // MyHoughCircles 종료 시간 측정
std::cout << "Execution time of  MyHoughCircles(): " << difftime(time_func_close, time_func_call) << " sec" << std::endl; // 소요 시간 출력
```

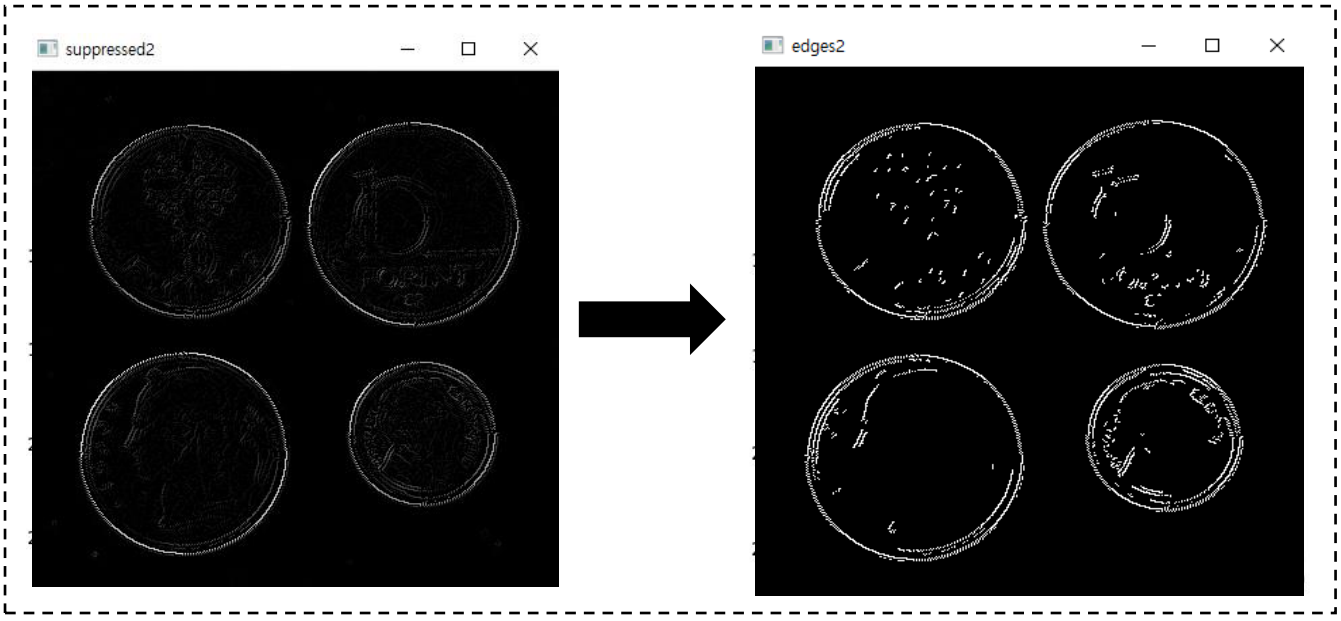
```
Execution time of  MyHoughCircles(): 5 sec
```

함수가 동작하는 시간은 위와 같이 time함수를 사용하여 측정하였다.





< 그림11. 비교1, non-maxumum suppression, 검출결과>



< 그림12. 비교2, non-maxumum suppression, 검출결과>

4. 고찰

본 구현에서는 입력이미지를 캐니 엣지 디텍터 알고리즘으로 엣지를 감지하고, 감지한 엣지를 토대로 원을 감지하는 프로그램을 구현하였다. 프로그램의 각 단계가 직렬구조를 가지고 있기 때문에 이전에 이미지 처리 결과가 좋지 않으면 이후의 이미지의 결과가 나쁘게 출력되기 때문에 매우 까다로운 구현이었다.

예를들어, 본 구현에서 edge의 결과로 뽑아낸 이미지 결과는 non-maxumum suppression을 얼마나 제대로 수행하느냐에 따라 큰 차이가 나는 것을 볼 수 있었다.

위의 그림 11, 그림 12를 비교하면 이전단계의 이미지 처리가 이후 단계에 얼마나 큰 영향을 미치는지 확인 가능하다. 그림 11의 edge를 사용한 경우 원의 검출성능이 매우 떨어진다.

사용자의 파라미터 조절에 따라 결과의 차이가 매우 크게 나타났다. 이는 사용자가 결과를 육안으로 확인하면서 파라미터를 실험적으로 조절 해야함을 의미한다. 적절한 결과를 출력하려면 여러 개의 파라미터를 실험해보고 최적의 값을 찾아내는 것이 중요하다.

실력과 시간의 부족으로 일부 구현의 결과를 내지 못하였다. 이론과 구현의 간극을 메꾸기 위해서는 뛰어난 프로그래머의 코드에서 구현의 논리를 확인하는 훈련이 더 필요할 것으로 사료된다.

원은 매우 기초적인 도형 중 하나로, 실생활에서 많이 등장하는 모양이다. 그러나, 실생활에서 접하는 원중에서 그 모양이 완벽한 원은 없다. 카메라로 이미지를 입력받는 각도나 광원의 방향에 따라 얼마든지 왜곡이 발생할 수 있다. 만일 과제로 제출된 것이 타원을 검출하는 내용이었다면 난이도는 크게 올라갔을 것이다. 실제로 자료를 조사한 바로는, opencv 내장함수인 houghcircle()을 사용하더라도 타원은 잘 검출하지 못하는 양상을 보여주었다.

실생활에서 실제 객체를 검출하기 위해서 추가적인 코드개선과 자료조사가 필요할 것으로 생각된다.