디지털 영상처리 연구실 연구보고서

정지우

우선 원래의 직선추출이다. (원점으로부터의 거리, 각도)

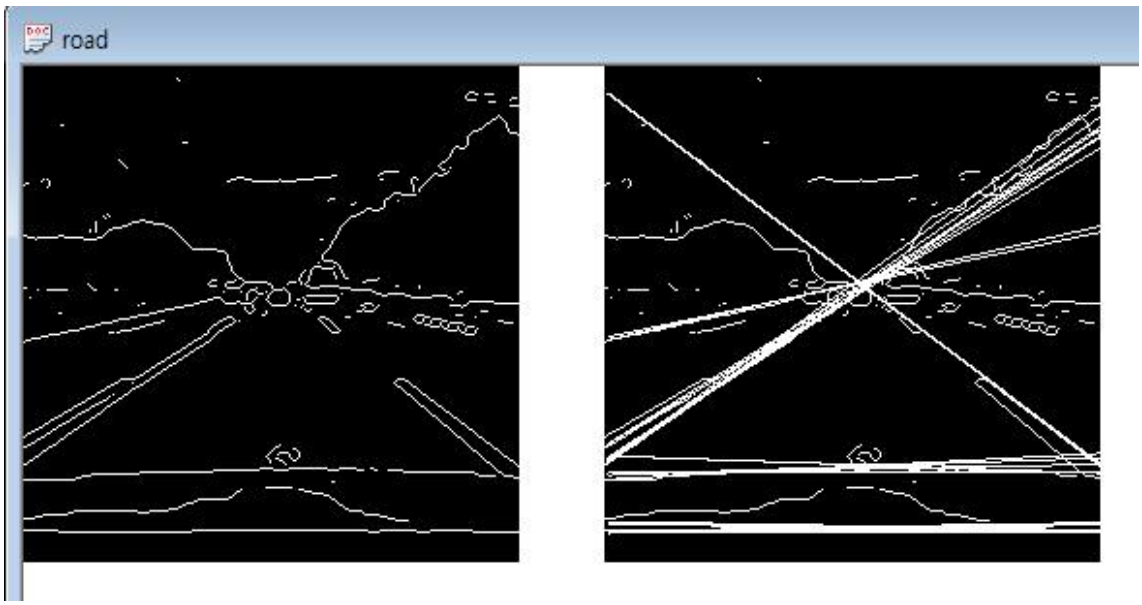
```
register int i, j, k;
int d, index;
float p2d = 3.1415926654f / 180.0f; //60분법( 1도, 24도 ...) 에서 라디안(3.14)같은거로 바꾸려는거임

int H[360][362] = { 0 }; //투표하기위한 배열
int thres = 60;

float* LUT_COS = new float[360];
float* LUT_SIN = new float[360];

//룩업테이블 만드는거임 1도마다 모든 sin,cos 구해서 배열에 저장하는거임
for (i = 0; i < 360; i++)
{
    LUT_COS[i] = (float)cos(i * p2d);
    LUT_SIN[i] = (float)sin(i * p2d);
}

// our 이미지는 im이미지랑 같고 그걸 다시 out에다가 넣어서 한마디로 out=org=in 된거임
for (i = 0; i < height * width; i++)
    outImg[i] = orgImg[i];
// For voting
for (i = 0; i < height; i++)
{
    index = i * width;
    for (j = 0; j < width; j++)
    {
        if (orgImg[index + j] == 255)
        {
            for (k = 0; k < 360; k++)
            {
                d = (int)(i * LUT_COS[k] + j * LUT_SIN[k]);
                if (d ≥ 4 && d ≤ 360) H[k][d]++;
            }
        }
    }
}
// For display
for (d = 4; d ≤ 360; d++)
{
    for (k = 0; k < 360; k++)
    {
        if (H[k][d] > thres) //H배열에다가 히스토그램처럼 투표해놨고 그기다가 th값 (60) 이상 투표되었으면 선 그리는거임
        {
            i = j = 2;

            for (j = 2; j < height; j++) // vertical pixel
            {
                i = (int)((d - j * LUT_SIN[k]) / LUT_COS[k]);
                if (i < height && i>0) outImg[i * width + j] = 255;
            }

            for (i = 2; i < width; i++) // horizontal pixel
            {
                j = (int)((d - i * LUT_COS[k]) / LUT_SIN[k]);
                if (j < height && j>0) outImg[i * width + j] = 255;
            }
        }
    }
}
```

```
// 왜 x,y축을 따로 처리하는가에 관한 대답이다.
//
// 정확성: 각 방정식이 적용되는 좌표 범위가 다를 수 있습니다. 한 방향으로만 계산하면 해상도가 낮은 경우 정확하지 않을 수 있습니다.
// 경우의 수 다루기: cos(θ)나 sin(θ) 값이 0에 가까워질 때, 계산에서 분모가 0에 가까워져 불안정해질 수 있습니다. 이를 방지하기 위해 두 방향을 분리하여 처리합니다.
// 완전성: 이미지의 모든 픽셀을 정확하게 커버하기 위해, 두 방향을 모두 고려하여 처리하는 것이 필요합니다. 이는 모든 가능한 픽셀 좌표를 포함할 수 있도록 보장합니다.
delete[]LUT_COS;
delete[]LUT_SIN;
```

정확성: 각 방정식이 적용되는 좌표 범위가 다를 수 있습니다. 한 방향으로만 계산하면 해상도가 낮은 경우 정확하지 않을 수 있습니다.

경우의 수 다루기: cos(θ)나 sin(θ) 값이 0에 가까워질 때, 계산에서 분모가 0에 가까워져 불안정해질 수 있습니다. 이를 방지하기 위해 두 방향을 분리하여 처리합니다.

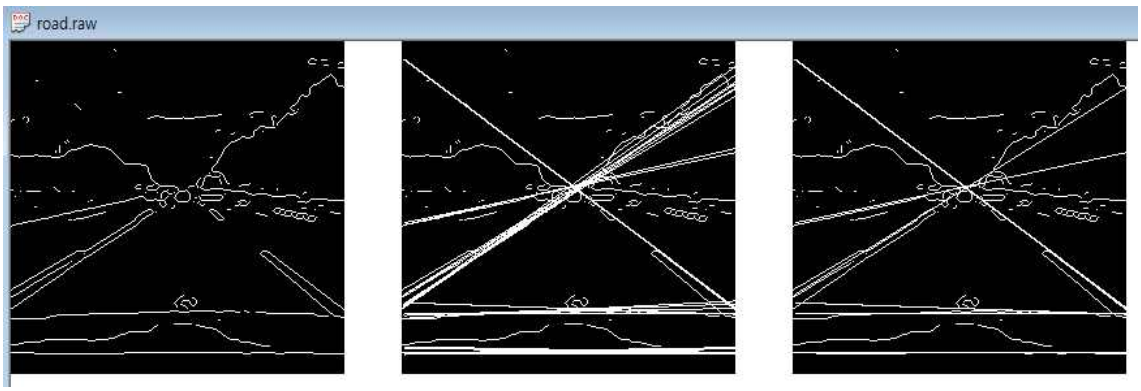
완전성: 이미지의 모든 픽셀을 정확하게 커버하기 위해, 두 방향을 모두 고려하여 처리하는 것이 필요합니다. 이는 모든 가능한 픽셀 좌표를 포함할 수 있도록 보장합니다.



일정범위 안에 한개로 모으기 (+ - 5)

```
// 합치기
for (d = 4; d ≤ 360; d++)
{
    for (k = 0; k < 360; k++)
    {
        if (H[k][d] > thres)
        {
            for (int d2 = d - 5; d2 ≤ d + 5; d2++)
            {
                if (d2 ≥ 4 && d2 ≤ 360)
                {
                    for (int k2 = k - 5; k2 ≤ k + 5; k2++)
                    {
                        if (k2 ≥ 0 && k2 ≤ 360)
                        {
                            if (H[k2][d2] > thres && !(d == d2 && k == k2))
                            {
                                H[k][d] += H[k2][d2];
                                H[k2][d2] = 0;
                            }
                        }
                    }
                }
            }
        }
    }
}
```

(보팅과 디스플레이 사이)



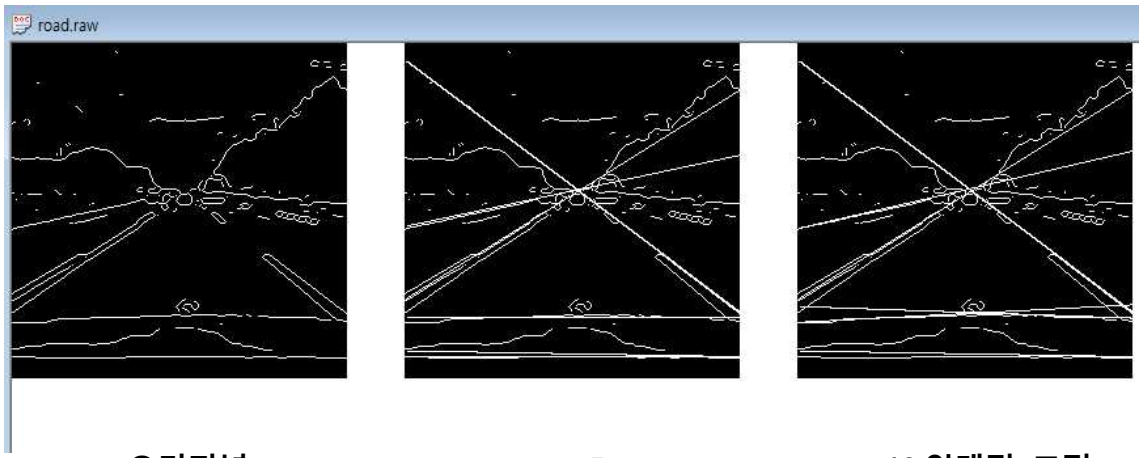

```
int main()
{

    int a[3];
    for (int i = -2; i < 3; ++i) {
        a[i] = i;
        cout << a[i] << " ";
    }
}
```

C:₩Users₩Ok₩Desktop₩C++₩C++₩연습장3₩x64₩Debug₩연습장3.exe

-2 -1 0 1 2

c++에는 이런 에러를 체크하는 기능이 없기 때문에
원래라면 에러가 나야하지만
-2 -1 0 1 2
를 출력하게 되고, 이런 코드를 사용한다면 undefined behaviour로 이어진다.
따라서 배열의 음수 인덱스를 사용할수는 있겠지만, 그게 올바른 사용법은 아니라는 뜻

(+ - 10과 임계점 조정 (64))

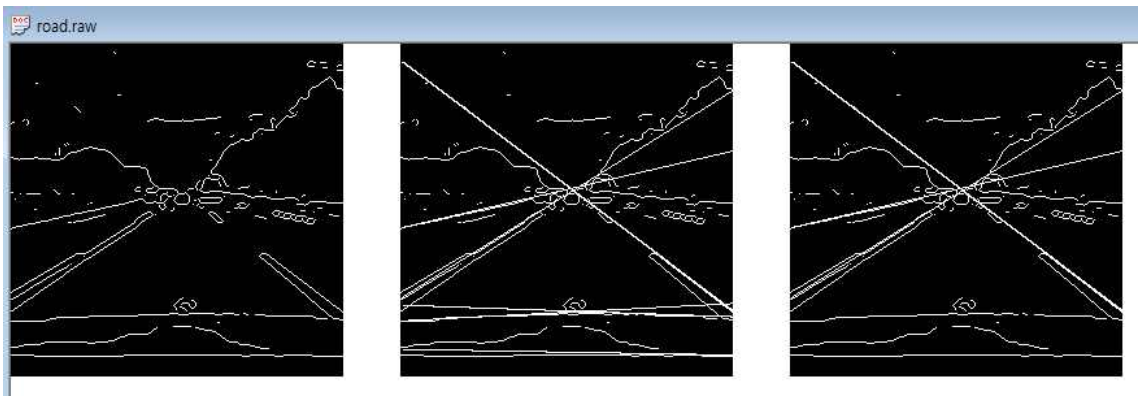


오리지널 +-5 +-10,임계점 조정

```
// 합치기
for (d = 4; d ≤ 360; d++)
{
    for (k = 0; k < 360; k++)
    {
        if (H[k][d] > thres)
        {
            for (int d2 = d - 10; d2 ≤ d + 10; d2++)
            {
                if (d2 ≥ 4 && d2 ≤ 360)
                {
                    for (int k2 = k - 10; k2 ≤ k + 10; k2++)
                    {
                        if (k2 ≥ 0 && k2 ≤ 360)
                        {
                            if (H[k2][d2] > thres && !(d == d2 && k == k2))
                            {
                                H[k][d] += H[k2][d2];
                                H[k2][d2] = 0;
                            }
                        }
                    }
                }
            }
        }
    }
}
```

이미지의 용도에 맞게 수평에 가까운 선은 없앤거임.

```
// For voting
for (i = 0; i < height; i++)
{
    index = i * width;
    for (j = 0; j < width; j++)
    {
        if (orgImg[index + j] == 255)
        {
            for (k = 0; k < 360; k++)
            {
                // 수평에 가까운 각도는 무시
                if (k ≤ 10 || k ≥ 350) continue;

                d = (int)(i * LUT_COS[k] + j * LUT_SIN[k]);
                if (d ≥ 4 && d ≤ 360) H[k][d]++;
            }
        }
    }
}
```



운전화면에서 차선을 위한 영상처리라고 생각했을 때,

이미지의 중심에서 가장 가까운 기울어진 직선이 차선이므로
가장 가까운 직선을 제외하고는 없앤다.

오리지널 코드

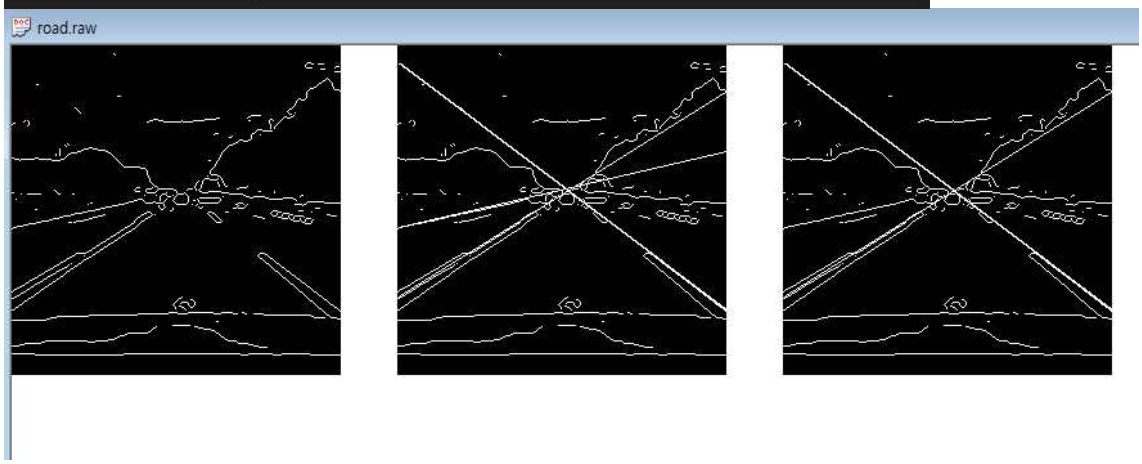
```
// For display
for (d = 4; d ≤ 360; d++)
{
    for (k = 0; k < 360; k++)
    {
        if (H[k][d] > thres)
        {
            i = j = 2;

            for (j = 2; j < height; j++) // vertical pixel
            {
                i = (int)((d - j * LUT_SIN[k]) / LUT_COS[k]);
                if (i < height && i > 0) outImg[i * width + j] = 255;
            }

            for (i = 2; i < width; i++) // horizontal pixel
            {
                j = (int)((d - i * LUT_COS[k]) / LUT_SIN[k]);
                if (j < height && j > 0) outImg[i * width + j] = 255;
            }
        }
    }
}
```

0~180사이 값중에서 젤 큰 값

```
// For display
// 우선 0~180사이
int max_k = -1;
int dd = -1;
for (d = 4; d ≤ 360; d++)
{
    for (k = 0; k < 180; k++)
    {
        if (H[k][d] > thres)
        {

            if (k > max_k)
            {
                max_k = k;
                dd = d;
            }


        }
    }
}

i = j = 2;

for (j = 2; j < height; j++) // vertical pixel
{
    i = (int)((dd - j * LUT_SIN[max_k]) / LUT_COS[max_k]);
    if (i < height && i > 0)
        outImg[i * width + j] = 255;
}

for (i = 2; i < width; i++) // horizontal pixel
{
    j = (int)((dd - i * LUT_COS[max_k]) / LUT_SIN[max_k]);
    if (j < height && j > 0)
        outImg[i * width + j] = 255;
}
```

180~360 사이에서 젤 작은값

```
//다음 180~360사이
int max_kk = 1000;
int ddd = -1;
for (d = 4; d ≤ 360; d++)
{
    for (k = 180; k < 360; k++)
    {
        if (H[k][d] > thres)
        {

            if (k < max_kk)
            {
                max_kk = k;
                ddd = d;
            }
        }
    }
}

i = j = 2;

for (j = 2; j < height; j++) // vertical pixel
{
    i = (int)((ddd - j * LUT_SIN[max_kk]) / LUT_COS[max_kk]);
    if (i < height && i > 0)
        outImg[i * width + j] = 255;
}

for (i = 2; i < width; i++) // horizontal pixel
{
    j = (int)((ddd - i * LUT_COS[max_kk]) / LUT_SIN[max_kk]);
    if (j < height && j > 0)
        outImg[i * width + j] = 255;
}

delete[] LUT_COS;
delete[] LUT_SIN;
```

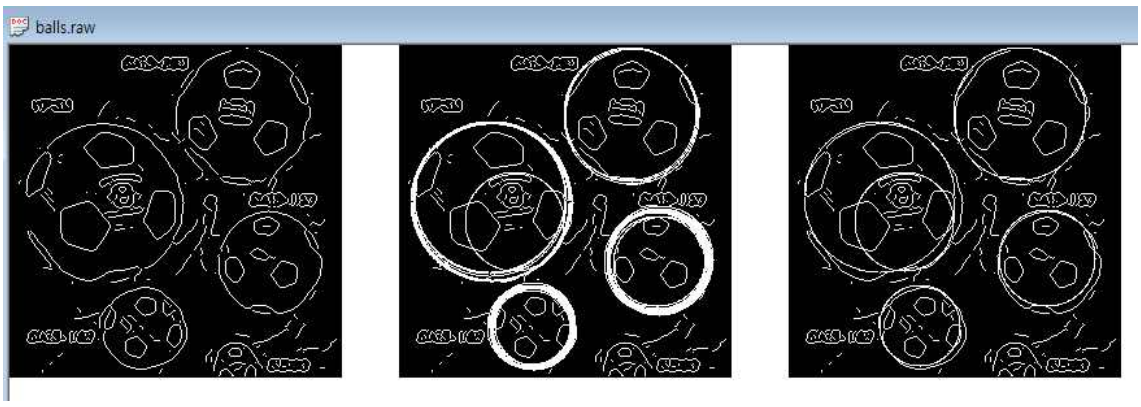


원 추출에서 노이즈 없애기

우선 직선과 마찬가지로

원 중심의 x,y좌표 와 반지름 r 에 관해서 일정범위 안에를 대푯값으로 통일 시키기

```
//합치기
for (i = 0; i < height; i++)
{
    for (j = 0; j < width; j++)
    {
        for (k = r_min; k < r_max; k++)
        {
            if (H[i][j][k] > thres) //th값(90)이상인것 들만 실제로 구현하자는것임
            {
                for (int i2 = i - 10; i2 < i + 10; i2++)
                {
                    if (i2 ≥ 0 && i2 ≤ height)
                    {
                        for (int j2 = j - 10; j2 < j + 10; j2++)
                        {
                            if (j2 ≥ 0 && j2 ≤ width)
                            {
                                for (int k2 = k - 10; k2 < k + 10; k2++)
                                {
                                    if (k2 ≥ r_min && k2 ≤ r_max)
                                    {
                                        if (H[i][j][k] > thres && !(i == i2 && j == j2 && k == k2))
                                        {
                                            H[i][j][k] += H[i2][j2][k2];
                                            H[i2][j2][k2] = 0;
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

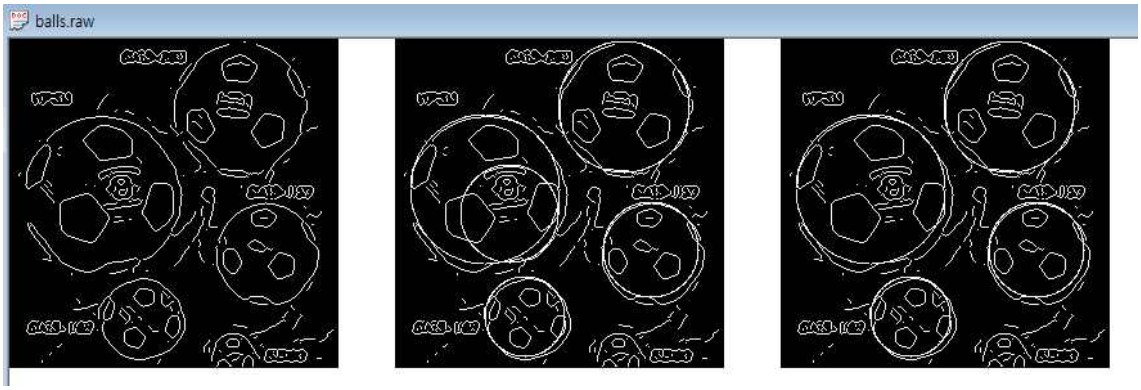


- 임계점 이상의 원의 중심좌표와 반지름을 구조체에 저장
- 원의 중심과의 거리와 반지름을 비교하여 큰원 안에 작은 원이 있는지 확인
- 작은원을 제거

```
#include <vector>
#include <cmath>
struct Circle //구조체 선언
{
        int i; // center y-coordinate
        int j; // center x-coordinate
        int k; // radius
};
```

std::vector<Circle> circles; //벡터 선언

```
// 임계점 이상의 원을 저장해라
for (i = 0; i < height; i++)
{
    for (j = 0; j < width; j++)
    {
        for (k = r_min; k < r_max; k++)
        {
            if (H[i][j][k] > thres) //th값(90)이상인것 들만 실제로 구현하자는것임
            {
                Circle circle = { i, j, k };
                circles.push_back(circle); //만들어진 객체를 저장하는 함수
            }
        }
    }
}

// 큰원안에 작은원 있으면 제거할거임
for (size_t m = 0; m < circles.size(); m++) //m은 비교의 기준이 되는 원이고 , n은 비교대상인 원
{
    for (size_t n = 0; n < circles.size(); n++)
    {
        if (m != n)// 자기자신과 비교하지 않도록
        {
            // 두 원의 중심 사이의 거리의 제곱
            int dist_sq = (circles[m].i - circles[n].i) * (circles[m].i - circles[n].i) +
                          (circles[m].j - circles[n].j) * (circles[m].j - circles[n].j);
            // 작은 원이 큰 원 안에 있는지 확인
            if (dist_sq < circles[m].k * circles[m].k && circles[m].k > circles[n].k)
            {
                // 작은 원을 벡터에서 제거
                circles.erase(circles.begin() + n);
                n--;// 벡터에서 원소를 제거했으므로 인덱스를 하나 줄임
            }
        }
    }
}

// For display
for (const auto& circle : circles)
{
    for (ang = 0; ang < 360; ang++)
    {
        rr = (int)(circle.i + circle.k * LUT_COS[ang]);
        cc = (int)(circle.j + circle.k * LUT_SIN[ang]);

        if (rr > 0 && rr < height && cc > 0 && cc < width)
            outImg[rr * width + cc] = 255;
    }
}
```


balls.raw