실습 및 과제

12211728 강민영

▪ 9x9 Gaussian filter를 구현하고 결과를 확인할 것

```cpp
#include <iostream>
#include <vector>
#include <string>
#include "opencv2/core/core.hpp"
#include <opencv2/opencv.hpp>
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"

using namespace cv;
using std::string;
using namespace std;

Mat MyCopy(Mat srcImg)
{
    int width = srcImg.cols;
    int height = srcImg.rows;
    Mat dstImg(srcImg.size(), CV_8UC1);
    uchar* srcData = srcImg.data;
    uchar* dstData = dstImg.data;
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            dstData[y * width + x] = srcData[y * width + x];// dst data에 src data를 복사

        }
    }
    return dstImg;
}
```

먼저 myCopy함수를 통해 기존의 이미지인 srcImg의 데이터를 dstImg에 전송해서 이미지를 복사하는 함수이다.

```cpp
int myKernelConv9x9(uchar* arr, int kernel[][9], int x, int y, int width, int height)
{
    int sum = 0;
    int sumKernel = 0;

    for (int j = -1; j <= 1; j++) {
        for (int i = -1; i <= 1; i++)
        {
            if ((y + j) >= 0 && (y + j) < height && (x + j) >= 0 && (x + i) < width) {
                sum += arr[(y + j) * width + (x + i)] * kernel[i + 1][j + 1];
                sumKernel += kernel[i + 1][j + 1];
            }
        }
    }
    if (sumKernel != 0)
    {
        return sum / sumKernel;
    }
    else
        return sum;
}
```

주어진 이미지에 9x9 커널을 사용하여 컨볼루션 연산을 수행한다. 이 함수는 가중치가 적용된 평

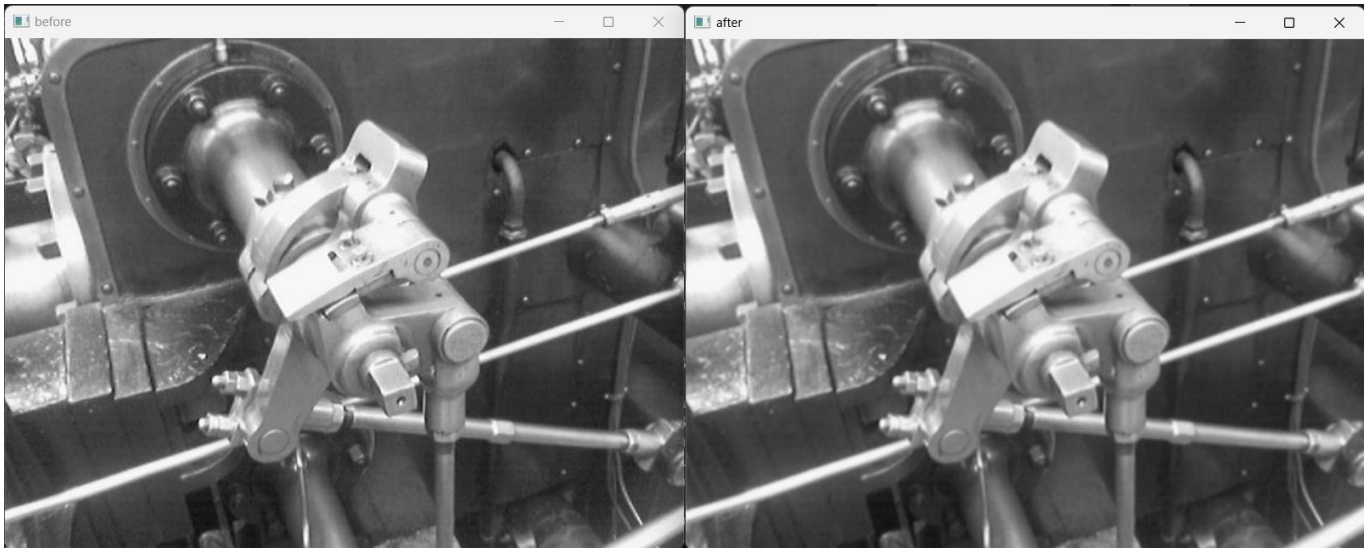
균을 계산하고 입력 이미지 경계를 벗어나는 경우에는 해당 픽셀을 무시한다

```
Mat myGaussianFilter(Mat srcImg) {
    int width = srcImg.cols;
    int height = srcImg.rows;
    int kernel[9][9] = {
        { 1,2,3,4,5,4,3,2,1 },
        { 2,3,4,5,6,5,4,3,2 },
        { 3,4,5,6,7,6,5,4,3 },
        { 4,5,6,7,8,7,6,5,4 },
        { 5,6,7,8,9,8,7,6,5 },
        { 4,5,6,7,8,7,6,5,4 },
        { 3,4,5,6,7,6,5,4,3 },
        { 2,3,4,5,6,5,4,3,2 },
        { 1,2,3,4,5,4,3,2,1 }
    };
    Mat dstImg(srcImg.size(), CV_8UC1);
    uchar* srcData = srcImg.data;
    uchar* dstData = dstImg.data;

    for (int y = 0; y < height; y++)
    {
        for (int x = 0; x < width; x++)
        {
            dstData[y * width + x] = myKernelConv9x9(srcData, kernel, x, y, width, height);
        }
    }
    return dstImg;
}
```

커널이 9x9인 행렬로 설정한 가우시안 필터이다. 가우시안 필터는 회전해도 값이 바뀌지 않기 때문에 대각선 기준으로 값이 대칭이게 한다. Dstdata에 값을 복사해주고 이때도 컨볼루션 함수를 사용하여 계산을 한다.

```
int main()
{
    Mat result;
    Mat temp = imread("gear.jpg", 0);
    Mat src_img = imread("gear.jpg", 0);

    result = myGaussianFilter(temp);//가우시안 필터 실행

    imshow("before", src_img);
    imshow("after", result);
    waitKey(0);
    destroyAllWindows();
}
```

Gear의 이미지를 불러와서 main함수를 실행하면 after사진이 before사진보다 블러 처리가 된 것을 확인할 수 있다.

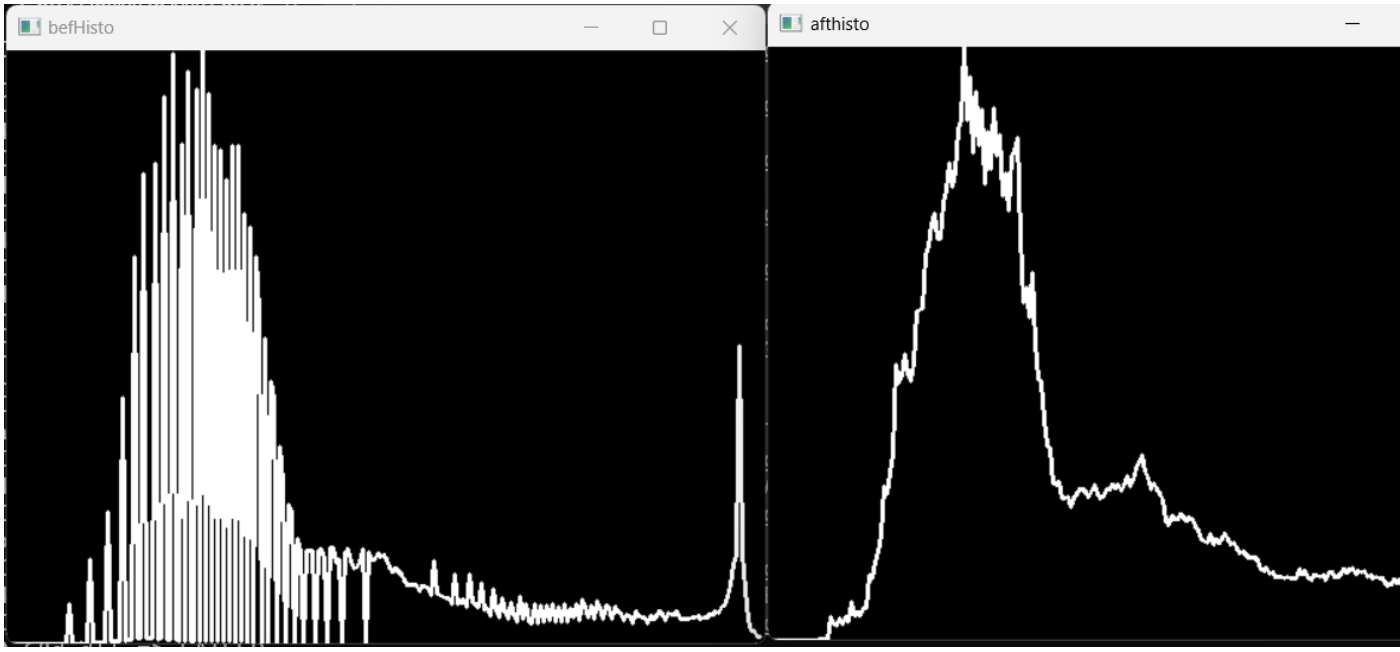


- **9x9 Gaussian filter를 적용했을 때 히스토그램이 어떻게 변하는지 확인할 것**

```
Mat GetHistogram(Mat src) {
    Mat histogram;
    const int* channel_numbers = { 0 };
    float channel_range[] = { 0.0, 255.0 };
    const float* channel_ranges = channel_range;
    int number_bins = 255;
    //히스토그램 계산
    calcHist(&src, 1, channel_numbers, Mat(), histogram, 1, &number_bins, &channel_ranges);

    //히스토그램 plot
    int hist_w = 512;
    int hist_h = 400;
    int bin_w = cvRound((double)hist_w / number_bins);

    Mat histImage(hist_h, hist_w, CV_8UC1, Scalar(0, 0, 0));

    //정규화 과정을 진행
    normalize(histogram, histogram, 0, histImage.rows, NORM_MINMAX, -1, Mat());

    for (int i = 1; i < number_bins; i++) {
        line(histImage, Point(bin_w * (i - 1), hist_h - cvRound(histogram.at<float>(i - 1))),
            Point(bin_w * (i), hist_h - cvRound(histogram.at<float>(i))),
            Scalar(255, 0, 0), 2, 8, 0);
    }

    return histImage;
}
```

히스토그램 함수를 불러와서 1번 문제에서의 코드에 추가해준다.

```
int main()
{
    Mat result;
    Mat temp = imread("gear.jpg", 0);
    Mat src_img = imread("gear.jpg", 0);

    Mat befhisto = GetHistogram(src_img);
    result = myGaussianFilter(temp);
    Mat afterhisto = GetHistogram(result);
    imshow("befHisto", befhisto);
    imshow("afthisto", afterhisto);

    waitKey(0);
    destroyAllWindows();
}
```



히스토그램의 결과를 확인하면 afterhito가 befhisto에 비해 노이즈가 없어 그래프가 깔끔하게 나오는 것을 확인할 수 있다.

가우시안 필터를 통해 노이즈들이 블러링 되면서 필터링이 된 것을 히스토그램을 통해 확인할 수 있다.

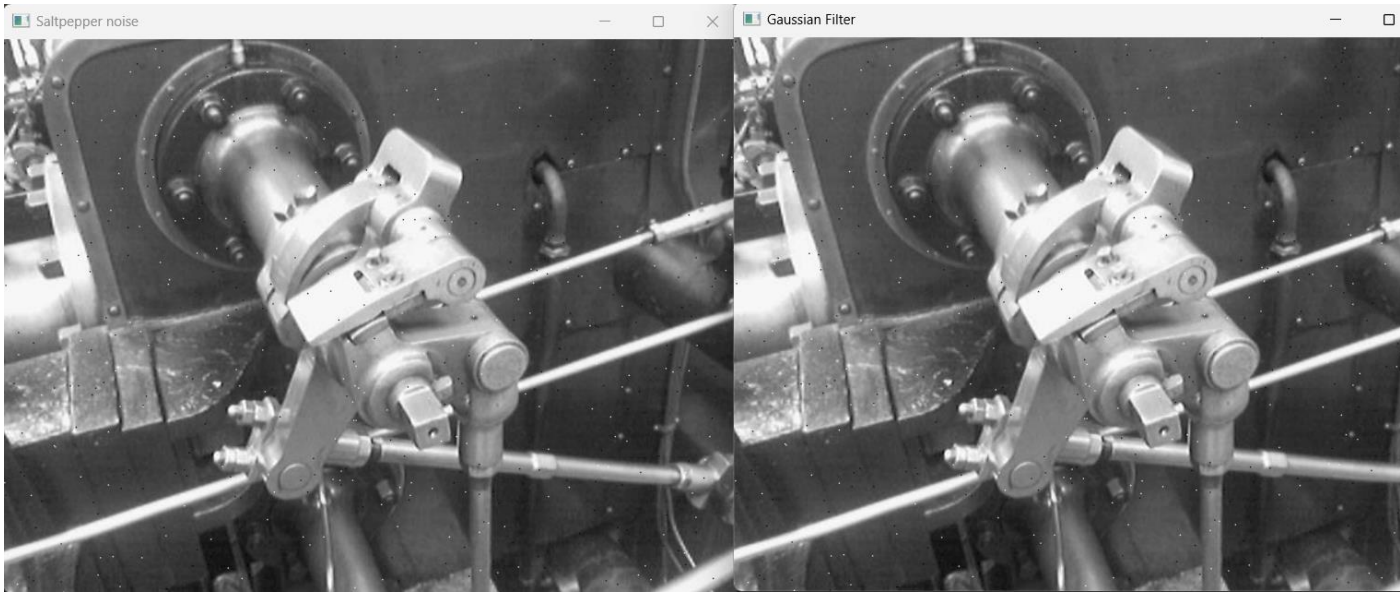
▪ 영상에 Salt and pepper noise를 주고, 구현한 9x9 Gaussian filter를 적용해볼 것

```
Mat saltAndPepper(Mat img, int num) {
    //찍을 점의 개수를 num으로 설정
    for (int n = 0; n < num; n++) {
        int x = rand() % img.cols; //이미지의 폭 정보저장
        int y = rand() % img.rows; //이미지의 높이 정보저장


        if (img.channels() == 1) {
            //이미지 채널수 반환
            if (n % 2 == 0) {
                img.at<char>(y, x) = 255; //단일 채널 접근
            }
            else {
                img.at<char>(y, x) = 0;
            }
        }

    }

    return img;
}
```

랜덤으로 RGB 점을 생성했던 함수를 참고하여 흑과 백의 점들을 만들어주는 saltandpepper함수를 만들었다. Uuchar을 통해 흰색은 255 검은색은 0으로 값을 설정하여 랜덤으로 이미지에 뿌려주는 함수를 만든다.

```
int main()
{
    Mat src_img = imread("gear.jpg", 0);

    Mat img = saltAndPepper(src_img, 500);
    //saltpepper함수 적용한 이미지 불러오기
    imshow("Saltpepper noise", img);

    Mat gaussianImg = myGaussianFilter(img);
    imshow("Gaussian Filter", img);
    //가우시안 필터 씌운 이미지 불러오기
    waitKey(0);
    destroyAllWindows();


    return 0;
}
```

이렇게 실행하게 되면



생성된 노이즈인 saltandpepper가 필터에 의해 조금 블러처리된 것을 알 수 있다.

▪ 45도와 135도의 대각 에지를 검출하는 Sobel filter를 구현하고 결과를 확인할 것

```
Mat mySobelFilter(Mat srcImg) {
    int kernel45[3][3] = { -2,-1,0,
                           -1,0,1,
                           0,1,2 };
    // 45도 대각 에지 검출
    int kernel135[3][3] = { 0,1,2,
                           -1,0,1,
                           -2,-1,0 };
    // 135도 대각 에지 검출

    Mat dstImg(srcImg.size(), CV_8UC1);
    uchar* srcData = srcImg.data;
    uchar* dstData = dstImg.data;
    int width = srcImg.cols;
    int height = srcImg.rows;

    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            dstData[y * width + x] = (abs(myKernerConv3x3(srcData, kernel45, x, y, width, height))
                + abs(myKernerConv3x3(srcData, kernel135, x, y, width, height))) / 2;
            // 45도와 135도의 대각 에지를 검출하기 위해 두 에지 결과의 절댓값 합 형태로 해서 최종결과를 도출
        }
    }

    return dstImg;

}
```

```
int main()
{
    Mat src_img = imread("gear.jpg", 0);

    Mat img = mySobelFilter(src_img);
    imshow("SobelImg", img);
    waitKey(0);
    destroyAllWindows();
}
```

기존의 sobelfilter은 수직이나 수평에서의 에지를 검출했다면 이번에는 45도와 135도의 에지를 검출해야 하기 때문에 45도와 135도 대각 부분에 0을 주어 급격한 변화를 줌으로써 에지를 검출하게 한다. 이때 둘의 절댓값의 합을 취해서 둘의 합으로 에지가 검출된 이미지를 얻는다.

▪ 컬러영상에 대한 Gaussian pyramid를 구축하고 결과를 확인할 것

```cpp
#include <iostream>
#include <vector>
#include <string>
#include "opencv2/core/core.hpp"
#include <opencv2/opencv.hpp>
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"

using namespace cv;
using std::string;
using namespace std;


//3x3 커널의 컨볼루션 연산 함수
int myKernerConv3x3(uchar* arr, int kernel[][3], int x, int y, int width, int height) {
    int sum = 0;
    int sumKernel = 0;

    for (int j = -1; j <= 1; j++) {
        for (int i = -1; i <= 1; i++) {
            if ((y + j) >= 0 && (y + j) < height && (x + i) >= 0 && (x + i) < width) {
                sum += arr[(y + j) * width + (x + i)] * kernel[i + 1][j + 1];
                sumKernel += kernel[i + 1][j + 1];
            }
        }
    }

    if (sumKernel != 0) { return sum / sumKernel; }
    else return sum;
}
```

컬러 영상에 대한 가우시안 피라미드를 구축하여야하기 때문에 기존의 가우시안 필터링 함수와 컨볼루션 함수를 수정하여 r,g,b를 설정하는 커널로 한다. 기존 컨볼루션 연산 함수에 커널에 색을 위해 chnnel array를 추가하였고 rgb 컬러 영상을 보여주기 위해 행과 열에 3을 곱하여 coloring을 가지는 convolution연산이 가능하게 한다.

```cpp
//컬러 영상에서 Gaussian filter에서 활용하는 마스크 배열 함수
int myKernerConv3x3(uchar* arr, int kernel[][3], int col, int row, int k, int width, int height)
{
    int sum = 0;
    int sumKernel = 0;

    for (int j = -1; j <= 1; j++)
    {
        for (int i = -1; i <= 1; i++)
        {
            if ((row + j) >= 0 && (row + j) < height && (col + i) >= 0 && (col + i) < width)
            {
                //RGB 컬러 영상에서 row, col 항에 3을 곱한 값 사용
                int color = arr[(row + j) * 3 * width + (col + i) * 3 + k];
                sum += color * kernel[i + 1][j + 1];
                sumKernel += kernel[i + 1][j + 1];
            }
        }
    }

    return sum / sumKernel;
}
```

```
Mat myGaussianFilter(Mat srcImg) {
    int width = srcImg.cols;
    int height = srcImg.rows;
    int kernel[3][3] = { 1,2,1,
                         2,4,2,
                         1,2,1 };

    Mat dstImg(srcImg.size(), CV_8UC3); //컬러 영상에는 CV_8UC3를 사용
    uchar* srcData = srcImg.data;
    uchar* dstData = dstImg.data;

    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            int index = y * width * 3 + x * 3; //RGB 컬러 영상이므로 data 값에 3을 곱하기
            int index2 = (y * 2) * (width * 2) * 3 + (x * 2) * 3;

            for (int k = 0; k < 3; k++)
            {
                dstData[index + k] = myKernerConv3x3(srcData, kernel, x, y, k, width, height);
            }
        }
    }

    return dstImg;
}
```

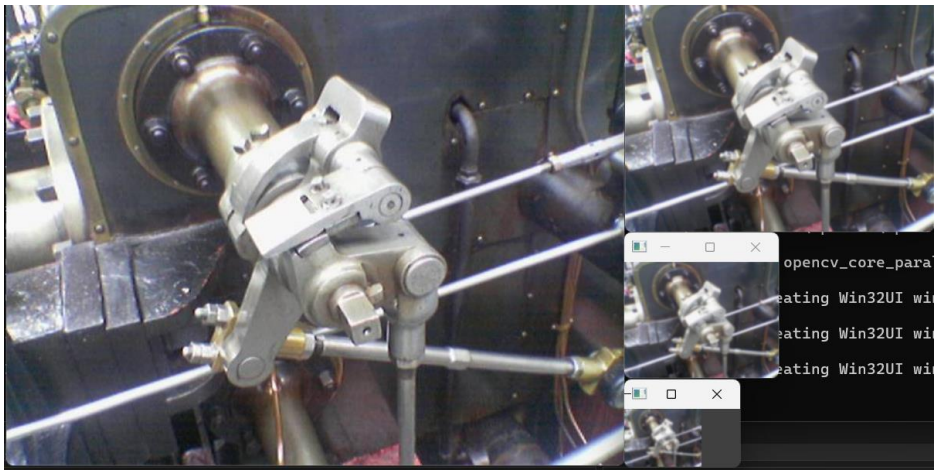
필터를 통과할 커널을 설정하고 rgb컬러 영상이므로 data값에 3을 곱하여 color의 데이터 값을 만든다.

```
//영상을 다운 샘플링하는 함수
Mat mySampling(Mat srcImg) {
    int width = srcImg.cols / 2;
    int height = srcImg.rows / 2;
    Mat dstImg(height, width, CV_8UC3);
    //컬러 영상이므로 CV_8UC3를 사용
    uchar* srcData = srcImg.data;
    uchar* dstData = dstImg.data;

    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            //컬러 영상이므로 data 값에 3을 곱한거사용
            int i1 = y * width * 3 + x * 3;
            int i2 = (y * 2) * (width * 2) * 3 + (x * 2) * 3;

            dstData[i1 + 0] = srcData[i2 + 0];
            dstData[i1 + 1] = srcData[i2 + 1];
            dstData[i1 + 2] = srcData[i2 + 2];
        }
    }

    return dstImg;
}
```

mySampling 함수는 입력 이미지를 절반으로 다운 샘플링하여 새로운 이미지를 생성하는데 RGB 세 개의 채널인 컬러 영상이므로 이를 고려하여 픽셀 데이터에 접근한다. 이때 가로와 세로 각각 절반 크기의 새로운 이미지를 만들기 위해 두 번째 루프에서는 가로와 세로를 각각 절반으로 설정한다. 이렇게 해당 채널들을 고려하여 이미지를 처리한다.

```cpp
//Gaussian 피라미드 생성 함수
vector<Mat> myGaussianPyramid(Mat src_img) {
    vector<Mat> Vec;

    Vec.push_back(src_img);
    for (int i = 0; i < 4; i++) {
#if USE_OPENCV
        pyrDown(src_img, src_img, Size(src_img.cols / 2, src_img.rows / 2));
#else
        src_img = mySampling(src_img);
        src_img = myGaussianFilter(src_img);//가우시안 필터링
#endif
        Vec.push_back(src_img);
    }

    return Vec;
}


int main()
{
    Mat src_img = imread("gear.jpg", 1);

    vector<Mat> pyramid = myGaussianPyramid(src_img);

    imshow("1", pyramid[0]);
    imshow("2", pyramid[1]);
    imshow("3", pyramid[2]);
    imshow("4", pyramid[3]);

    waitKey(0);
    destroyAllWindows();

    return 0;
```

가우시안 피라미드 함수는 src_img를 샘플링 함수와 필터링 함수를 거치고 나서 피라미드에 push_back된다. main 함수에서는 "gear.jpg" 파일을 컬러로 읽고 이를 가우시안 피라미드로 변환한다. 이렇게 변환된 피라미드 이미지들을 차례대로 화면에 표시한다.



결과를 보면 각 가로 세로의 절반 크기로 작아지면서 필터링이 되는 것을 볼 수 있다.

▪ 컬러영상에 대한 Laplacian pyramid를 구축하고 복원을 수행한 결과를 확인할 것

기존의 함수는 유지한 채 라플라시안 피라미드 함수만 추가한다.

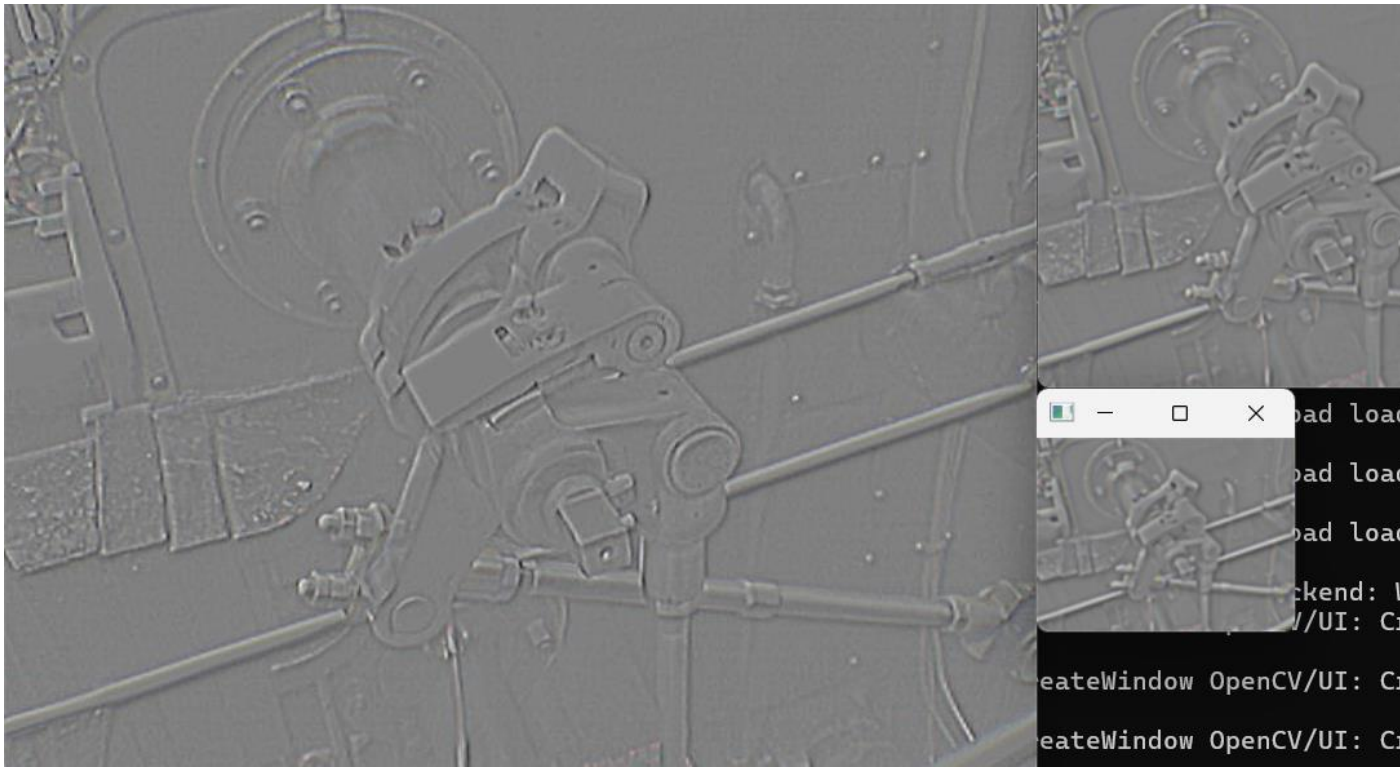
```
vector<Mat> myLaplacianPyramid(Mat srcImg) {
    vector<Mat> Vec;

    for (int i = 0; i < 4; i++) {
        if (i != 3) {
            Mat highImg = srcImg;

            srcImg = mySampling(srcImg);
            srcImg = myGaussianFilter(srcImg);

            Mat lowImg = srcImg;
            resize(lowImg, lowImg, highImg.size());
            Vec.push_back(highImg - lowImg + 128);

        }
        else {
            Vec.push_back(srcImg);
        }
    }

    return Vec;
}
```

라플라시안 피라미드 함수는 가우시안 피라미드 함수와 같이 샘플링을 하고 가우시안 필터링을 진행한다. Resize를 통해 자가진 영상을 백업한 영상의 크기로 확대한다. 차의 연산을 벡터 배열에 삽입할 때 0~255의 범위를 벗어나는 것을 방지하기 위해 128을 더한다.

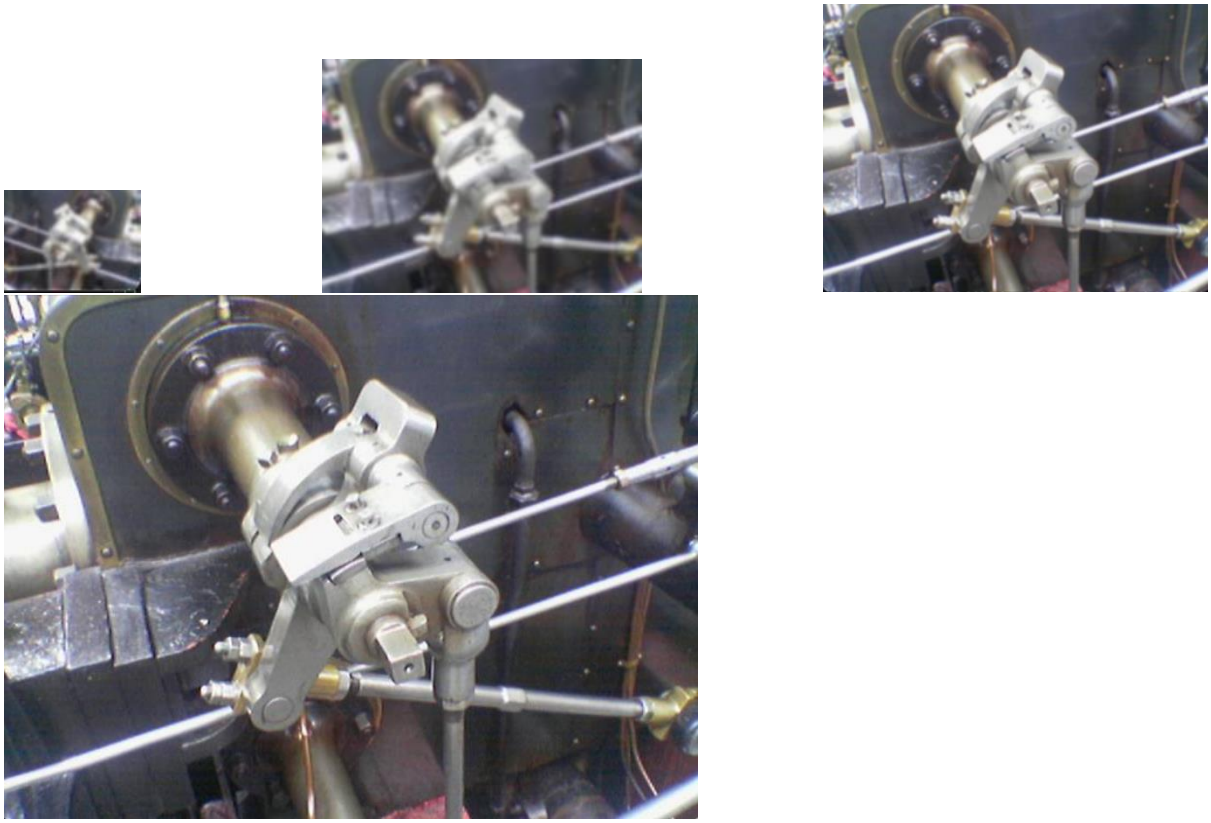
```cpp
int main()
{
    Mat src_img = imread("gear.jpg", 1);
    Mat dst_img;

    vector<Mat> Vecpyra = myLaplacianPyramid(src_img);

    imshow("1", Vecpyra[0]);
    imshow("2", Vecpyra[1]);
    imshow("3", Vecpyra[2]);


    waitKey(0);
    destroyAllWindows();

    reverse(Vecpyra.begin(), Vecpyra.end()); //작은 영상순서로 처리하기 위해 벡터 순서를 반대로

    for (int i = 0; i < Vecpyra.size(); i++) {
        if (i == 0) { // 가장 작은 영상 바로 출력
            dst_img = Vecpyra[i];

            imshow("Test1", dst_img);
            waitKey(0);
            destroyWindow("Test1");
        }
        else {
            resize(dst_img, dst_img, Vecpyra[i].size()); //작은 영상 확대
            dst_img = dst_img + Vecpyra[i] - 128;//큰 영상으로 복원
            //더했던 128을 다시 뺌

            imshow("Test2", dst_img);
            waitKey(0);
            destroyWindow("Test2");
        }
    }
    waitKey(0);
    destroyAllWindows();

    return 0;
}
```

똑같이 피라미드로 출력하지만 라플라시안피라미드 함수를 적용하여 main 함수를 실행한다.

이를 출력하고 reverse함수를 통해 작은 영상부터 순서가 처리되도록 하고 resize함수를 통해 작은 영상을 확대한다. 또한 오버플로우를 방지하기 위해 아까 더했던 128을 다시 빼도록 한다.



영상을 보면 함수가 잘 적용되어 출력됨을 확인할 수 있다.



출력결과를 보면 영상의 크기가 점점 커지면서 색깔 또한 점점 드러나는 것을 확인할 수 있다.