KWANGWOON UNIVERSITY

디 지 털 영 상 처 리

HW #3

담당 교수님 : 이윤구 교수님

담당 조교님 : 윤여경 조교님

제 출 날 짜 : 2018. 11. 25.

학 과 : 컴퓨터소프트웨어학과

학 번 : 2014707040

이 름 : 유 진 혁

HW #3 – 1

■ Color Slicing

- 소스 코드 및 구현 방법

```
const int HEIGHT = 256;      // 이미지 세로 길이
const int WIDTH = 256;       // 이미지 가로 길이

// 대상 이미지
unsigned char imageR[HEIGHT][WIDTH] = { 0 };
unsigned char imageG[HEIGHT][WIDTH] = { 0 };
unsigned char imageB[HEIGHT][WIDTH] = { 0 };

bool extract[HEIGHT][WIDTH] = { false };    // Color Slicing 추출 영역

// 얼굴 영역 Smoothing 영상
unsigned char maskR[HEIGHT][WIDTH] = { 0 };
unsigned char maskG[HEIGHT][WIDTH] = { 0 };
unsigned char maskB[HEIGHT][WIDTH] = { 0 };
```

전역으로 선언된 변수들이다. Color slicing을 할 대상 이미지의 R, G, B 값이 imageR, imageG, imageB 세 배열로 나눠 저장된다. bool형 extract 배열은 추출한 영역을 나타내는 배열이다. 인덱스의 좌표가 대상 이미지에서 추출할 영역에 속하면 true가, 그렇지 않으면 false가 저장된다. extract 배열을 참고하여 얼굴 영역에 smoothing filter를 적용한 이미지는 R, G, B로 나뉘어 maskR, maskG, maskB에 저장된다.

```
int main()
{
    // 파일 입력
    ifstream fin;
    fin.open("input.raw", ios::binary);
    for (int i = 0; i < HEIGHT; i++)
        for (int j = 0; j < WIDTH; j++)
        {
            imageR[i][j] = fin.get();
            imageG[i][j] = fin.get();
            imageB[i][j] = fin.get();
        }
    fin.close();

    ColorSlicing(215, 168, 134);    // Color Slicing 중심 픽셀 RGB = (215, 168, 134)
    SmoothingFilter();
    Overwrite();
```

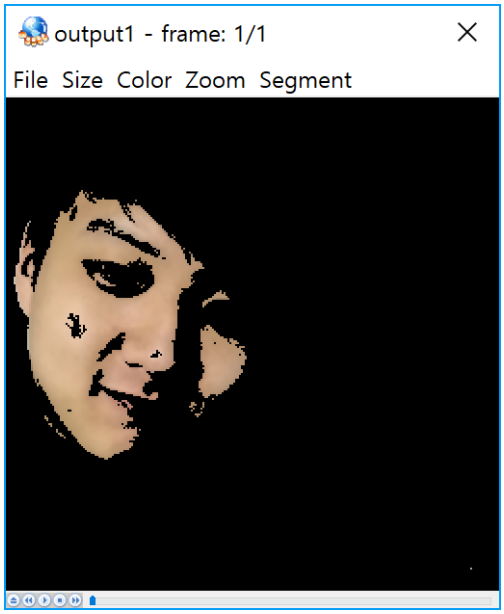
프로그램이 시작될 때, 파일명이 input.raw인 얼굴 사진을 읽는다. RAW 파일에는 픽셀 당 R, G, B의 값이 차례대로 저장되어 있으므로 한 픽셀을 읽을 때 제일 먼저 R값, 그 다음에 G값, 그 다음에 B값을 읽었다. 이를 각각 배열 imageR, imageG, imageB에 저장했다. 그리고 ColorSlicing, SmoothingFilter, Overwrite 함수를 순서대로 호출했다. ColorSlicing 함수에 들어가는 세 매개변수는 Color Slicing을 할 때 기준이 되는 중심 픽셀의 RGB 값이다. 여기서는 얼굴 피부색 중 밝지도, 어둡지도 않은 중간 영역의 픽셀을 사용했다.

```
// Sphere로 영역 추출
void ColorSlicing(unsigned char centerR, unsigned char centerG, unsigned char centerB)
{
    for (int i = 0; i < HEIGHT; i++)
        for (int j = 0; j < WIDTH; j++)
        {
            if(((imageR[i][j] - centerR)*(imageR[i][j] - centerR) + (imageG[i][j] - centerG)*(imageG[i][j] - centerG)
                                                                  + (imageB[i][j] - centerB)*(imageB[i][j] - centerB)) > 51 * 51)
                extract[i][j] = false;
            else
                extract[i][j] = true;
        }
}
```

Color Slicing은 얼굴 영역의 색을 구의 형태로 감싸는 방법을 사용했다. 매개변수로 들어온 RGB 값을 중심으로 하고, 반지름의 길이는 51로 지정하였다. 이 영역을 벗어나는 픽셀의 좌표에 대해 extract 배열의 값을 false로 하였고, 그렇지 않으면 true로 하였다.

```
// Color Slicing 추출 영역에 대해서 smoothing filter 적용하여 mask 생성
void SmoothingFilter()
{
    const double FILTER[3][3] = {{1.0 / 16.0, 2.0 / 16.0, 1.0 / 16.0},
                                 {2.0 / 16.0, 4.0 / 16.0, 2.0 / 16.0},
                                 {1.0 / 16.0, 2.0 / 16.0, 1.0 / 16.0}};

    // Spatial filtering
    for (int i = 1; i < HEIGHT - 1; i++)
        for (int j = 1; j < WIDTH - 1; j++)
        {
            if (extract[i][j])
            {
                double sum = 0.0;
                for (int s = -1; s <= 1; s++)
                    for (int t = -1; t <= 1; t++)
                        sum += (FILTER[s + 1][t + 1] * imageR[i + s][j + t]);
                maskR[i][j] = (unsigned char)sum;
                sum = 0.0;
                for (int s = -1; s <= 1; s++)
                    for (int t = -1; t <= 1; t++)
                        sum += (FILTER[s + 1][t + 1] * imageG[i + s][j + t]);
                maskG[i][j] = (unsigned char)sum;
                sum = 0.0;
                for (int s = -1; s <= 1; s++)
                    for (int t = -1; t <= 1; t++)
                        sum += (FILTER[s + 1][t + 1] * imageB[i + s][j + t]);
                maskB[i][j] = (unsigned char)sum;
            }
        }
}
```

Smoothing 과정에선 과제 2에서 주어졌던 smooth filter를 사용했다. extract 배열의 값을 보고 extract 값이 true라면, 즉, 얼굴 영역의 픽셀이라면 대상 이미지를 spatial filtering 하여 그 결과를 maskR, maskG, maskB에 저장했다.



maskR, maskG, maskB에 저장된 값을 출력하면 위와 같은 이미지를 얻을 수 있다. 대상 이미지의 얼굴 영역만 smoothing되어 저장된 것을 알 수 있다.

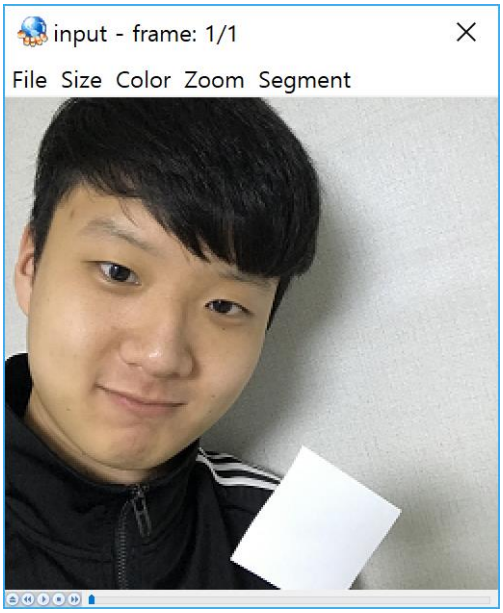
```
// 원본 영상에 mask overwrite
void Overwrite()
{
    for (int i = 0; i < HEIGHT; i++)
        for (int j = 0; j < WIDTH; j++)
        {
            if (extract[i][j])
            {
                imageR[i][j] = maskR[i][j];
                imageG[i][j] = maskG[i][j];
                imageB[i][j] = maskB[i][j];
            }
        }
}
```

이렇게 얻은 maskR, maskG, maskB의 픽셀 값을 얼굴 영역에서만 원본 대상 이미지에 대체시켰다.

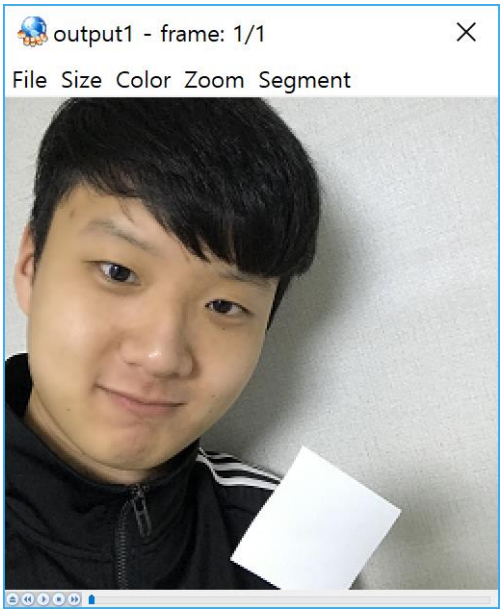
```
// 파일 출력
ofstream fout;
fout.open("output1.raw", ios::binary);
for (int i = 0; i < HEIGHT; i++)
    for (int j = 0; j < WIDTH; j++)
    {
        fout << imageR[i][j];
        fout << imageG[i][j];
        fout << imageB[i][j];
    }
fout.close();
```

마지막으로 imageR, imageG, imageB를 차례로 출력하여 결과 영상 output1.raw를 만들었다.

- 결과 분석



입력 영상



결과 영상

위 영상에서 얼굴의 볼 부분과, 눈썹, 코를 보면, 원본 영상에 있는 잡티와 모공이 결과 영상에선 사라진 것을 볼 수 있다. 얼굴 영역에 대한 smoothing filter의 효과로 피부가 더 뽀얗게 되었다. 그 외 부분에서는 변화가 없는 것을 볼 수 있다.

- 고찰

RAW 파일은 한 픽셀의 RGB 값을 차례로 저장한다는 것을 알아 내고, 먼저 전체 컬러 이미지를 smoothing 하는 것부터 차근차근 시작하였다. 컬러 입력 이미지를 R, G, B의 세 이미지로 나누고, 흑백 사진을 smoothing 할 때와 마찬가지로 각각을 spatial filtering 한 뒤 다시 합치면 되겠다고 생각하고 프로그램을 작성하였다. spatial filtering을 하면 테두리의 픽셀이 사라지는 것을 감안하여 결과 영상이 테두리 부분 없이 출력되도록 출력 시작 위치를 조정하였는데, 그 결과 정상적인 색의 이미지가 아닌, 잘못된 색의 이미지가 출력되었다. 디버깅을 해보니 컬러 이미지에서 한 픽셀을 건너뛰기 위해선 3bytes를 건너뛰어야 하는데 흑백 이미지에서 했던 대로 1byte만 건너 뛰어 생긴 문제였다. 그래서 G값으로 출력되는 부분엔 R값이, B값으로 출력되는 부분엔 G값이 들어가 다른 색의 이미지가 출력되었다. 이를 염두에 두고 이제 얼굴 영역에서만 smoothing을 적용하기 위해 적당한 픽셀을 잡아 얼굴 영역을 추출하였다. 추출해낸 얼굴을 smoothing하고 이를 원본 영상에 덮어씌웠더니 덮어씌운 부분의 경계가 검게 출력되는 문제가 발생하였다. 이는 얼굴 영역을 추출할 때, 얼굴이 아닌 부분의 RGB 값을 0으로 하였는데 spatial filtering을 할 때 이 영역의 값을 사용하여 발생한 문제였다. RGB가 0인 픽셀이 filtering에 개입되면서 추출 영상 경계의 픽셀이 부자연스러운 어두운 색의 픽셀로 대체되었다. 이 문제는 추출한 영상을 smoothing 하지 않고 단순히 참고만 하여 해당 부분의 원본 영상을 smoothing 하는 것으로 해결할 수 있었다.

HW #3 – 2

■ White Balancing

- 소스 코드 및 구현 방법

```
const int HEIGHT = 256;      // 이미지 세로 길이
const int WIDTH = 256;       // 이미지 가로 길이

// White Balancing 적용 전 이미지
unsigned char inputR[HEIGHT][WIDTH] = { 0 };
unsigned char inputG[HEIGHT][WIDTH] = { 0 };
unsigned char inputB[HEIGHT][WIDTH] = { 0 };

// White Balancing 적용 전 이미지
unsigned char outputR[HEIGHT][WIDTH] = { 0 };
unsigned char outputG[HEIGHT][WIDTH] = { 0 };
unsigned char outputB[HEIGHT][WIDTH] = { 0 };
```

White balancing이 적용되지 않은 이미지의 RGB 값을 저장할 배열 inputR, inputG, inputB를 선언하였다. 그리고 white balancing을 적용한 이미지의 RGB 값이 저장되는 배열 outputR, outputG, outputB도 선언하였다.

```
int main()
{
    // 파일 입력
    ifstream fin;
    fin.open("input.raw", ios::binary);
    for (int i = 0; i < HEIGHT; i++)
        for (int j = 0; j < WIDTH; j++)
        {
            inputR[i][j] = fin.get();
            inputG[i][j] = fin.get();
            inputB[i][j] = fin.get();
        }
    fin.close();

    WhiteBalancing(217, 222, 226);  // 흰색 종이 영역 중 가장 어두운 픽셀 값 = (217, 222, 226)

    // 파일 출력
    ofstream fout;
    fout.open("output2.raw", ios::binary);
    for (int i = 0; i < HEIGHT; i++)
        for (int j = 0; j < WIDTH; j++)
        {
            fout << outputR[i][j];
            fout << outputG[i][j];
            fout << outputB[i][j];
        }
    fout.close();
```

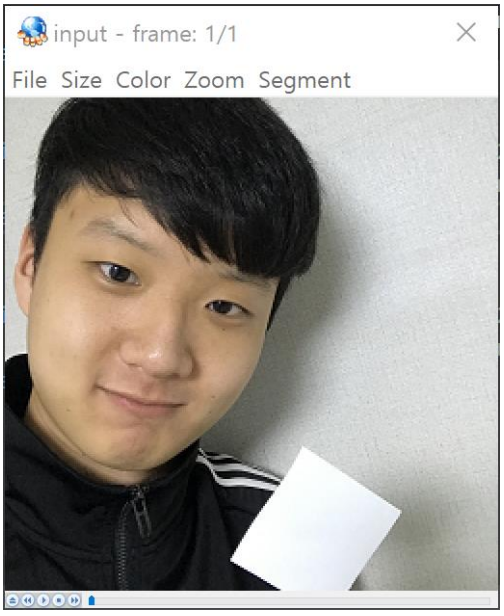
입력 영상인 input.raw 파일을 열고 이 영상의 R, G, B 영상을 inputR, inputG, inputB에 각각 저장하였다. 그리고 WhiteBalancing 함수를 호출한 뒤, 결과 영상이 저장된 배열 outputR, outputG,

outputB를 output2.raw 파일로 출력하였다. WhiteBalancing 함수의 매개변수는 값을 (255, 255, 255)로 맞출 픽셀의 RGB 값이다. 여기서는 흰색 종이의 영역 중에서 가장 어두운 픽셀의 값을 매개변수로 넣어줬다.

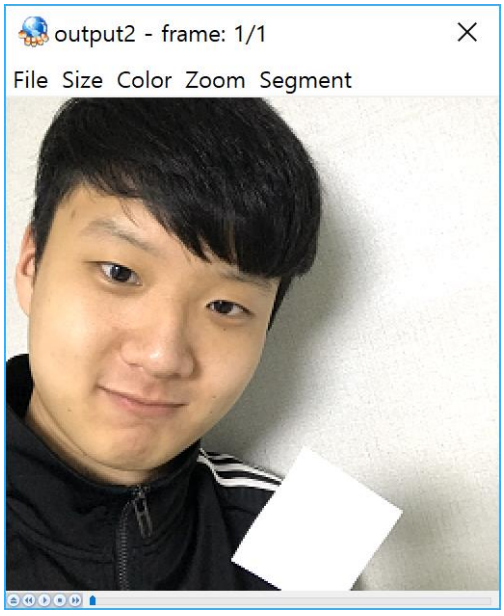
```
void WhiteBalancing(double whiteR, double whiteG, double whiteB)
{
    for (int i = 0; i < HEIGHT; i++)
        for (int j = 0; j < WIDTH; j++)
        {
            if ((inputR[i][j] / whiteR * 255) <= 255)
                outputR[i][j] = (unsigned char)(inputR[i][j] / whiteR * 255);
            else
                outputR[i][j] = 255;

            if ((inputG[i][j] / whiteG * 255) <= 255)
                outputG[i][j] = (unsigned char)(inputG[i][j] / whiteG * 255);
            else
                outputG[i][j] = 255;

            if ((inputB[i][j] / whiteB * 255) <= 255)
                outputB[i][j] = (unsigned char)(inputB[i][j] / whiteB * 255);
            else
                outputB[i][j] = 255;
        }
}
```

WhiteBalancing 함수 안에는 흰색 종이 영역의 픽셀 값이 (255, 255, 255)가 되도록 하는 color transformation 함수가 정의되어 있다. 매개변수로 들어온 픽셀 값을 (255, 255, 255)로 맞추기 위해, 현재 픽셀 값을 매개변수 픽셀 값으로 나누고 이 값에 255을 곱해주는 연산을 모든 픽셀에 수행하게 하였다. 그리고 그 결과 값을 배열 outputR, outputG, outputB에 저장하였다.

- 결과 분석



입력 영상



결과 영상

(255, 255, 255)보다 조금 어두운 흰색 종이 영역의 픽셀 값을 (255, 255, 255)로 맞추면서 흰색 종이가 어두운 부분없이 완전히 흰색으로 변한 것을 볼 수 있다. 이러한 변환이 입력 영상 모든 픽셀에 적용되면서 흰색 종이 영역 외 다른 부분의 색도 같이 밝아졌다.

- 고찰

선택 영역의 값이 (255, 255, 255)가 되도록 하는 연산을 모든 픽셀에 대해 적용해주면 되므로 구현은 비교적 간단했다. 다만, 곱하고 나누는 연산 결과의 자료형이 unsigned char이므로 계산 중간 자료형 변환에 주의가 필요했다. 단순히 unsigned char형으로만 계산 시, 나눗셈 부분에서 결과가 0 또는 1만 반환되므로 double형으로의 확장이 필요하다. 그래서 WhiteBalancing 함수의 인자의 자료형을 double로 하여 함수 계산이 진행될 때 자동으로 자료형 변환이 일어나도록 유도하였다. double형 연산 결과가 unsigned char형 배열에 저장되면서 소수점 이하 부분은 절사되어 결과 픽셀 값으로 저장된다.