멀티미디어프로그래밍

과제 2 보고서

제출일	2024. 04. 05	전공	소프트웨어학과
과목	멀티미디어프로그래밍	학번	21011746
담당교수	박상일 교수님	이름	양현석

1. Prokudin-Gorskii가 찍은 사진들의 칼라를 복원하기

a) 문제 분석

한 장의 이미지에는 각각 3장의 이미지가 포함되어 있다. 이 원본 이미지는 위에서부터 차례로 B, G, R의 색상의 Element를 갖고 있다. 이를 이용해서 각각 B, G, R를 값을 뽑아내서 합쳐야 한다. 하지만 3장의 이미지가 다른 구도에서 사진을 찍었기에, 그대로 합치게 되면 제대로 된 사진의 형태가 나오지 않는다. 이 문제점을 해결하기 위해 자동으로 사진을 맞춰주는 프로그램까지 완성하면 된다.

이 프로그램의 주안점은 크게 2가지로 주어졌다. 이미지가 정돈되었는지와 실행시간이다. 하지만 실행시간이 짧아질수록 이미지 정돈의 정확도가 떨어질 수 있기 때문에 이에 유의하며 코드를 짜야 한다.

b) 문제 풀이

첫 세팅은 다음과 같다. 원본 이미지 주소를 받고 불러온 후, 위에서부터 각각 이미지들을 불러와 IplImage 형 배열 div를 선언해서 3개의 이미지를 먼저 담아준다.

```
char name[1000];
printf("Test CV₩nInput File Name : ");
scanf("%s", name);
```

```
IplImage* div[3], * dst = cvCreateImage(cvSize(w, h / 3), 8, 3);
for (int i = 0; i < 3; i++)
        div[i] = cvCreateImage(cvSize(w, h / 3), 8, 3);

for (int i = 0; i < 3; i++)
        for (int y = i * (h / 3); y < (i + 1) * (h / 3); y++)
                for (int x = 0; x < w; x++) {
                        CvScalar temp = cvGet2D(src, y, x);
                        cvSet2D(div[i], y - i * (h / 3), x, temp);
                }
```

이제 div의 3장의 사진을 각각 비교할 준비를 해볼 것이다. 여기서 여러가지 시도를 해볼 수 있는데, 예를 들어 (div[0], div[1])와 (div[0], div[2]), (div[0], div[1])와 (div[1], div[2]), (div[0], div[2])와 (div[1], div[2]) 와 같이 방법이 한 가지가 아닐 것이다. 사실 큰 차이는 없겠지만, 이미지의 밝기를 생각해봤을 때 div[1]를 기준으로 하는 (div[0], div[1])와 (div[1], div[2]) 방법이 제일 좋을 것 같아

서 이를 채택해서 코드를 작성했다. (실제로도 결과가 가장 좋게 나왔다. 나름 이유를 추론해보자면, 이 알고리즘에서는 사진의 일정 영역이 비슷하다고 판단할 때 평균적인 색의 차이라 할 수 있는 값을 비교한다. 우리는 여기서 서로 강하지 않는 색을 이용해서 비교한다. 이 알고리즘에서 정의하는 서로 강하지 않은 색이란, div[0]과 div[1]을 비교한다고 했을 때 ~.val[2]를 지칭한다. <그래야 가장 공평하게 색을 비교할 수 있는 방법이라고 생각했다.> 사진을 육안으로만 보았을 때 보통 B->G->R로 갈수록 사진의 밝기가 밝아지기에 val[1]를 기준으로 잡는 이유가 될 것이다.)

이제 속도를 줄이는 알고리즘을 생각해볼 것이다. 사진을 비교하는데 쓰이는 코드의 기본형은 다음과 같다.

```
for (int v = (-h / 3) / 3; v <= (h / 3) / 3; v += 1)
                for (int u = -w / 3; u <= w / 3; u += 1) {
                        float err1 = 0.0f;
                        float err2 = 0.0f;
                        int ct = 0;
                        for (int y = h / 3; y < h / 3; y+=1) {
                                for (int x = w; x < w; x+=1) {      . . .
```

하지만 이 방법을 그대로 쓰면 매우 시간이 오래 걸리기 때문에, 정확도에는 영향을 최대한 주지 않고 좀 더 효율적으로 빠르게 비교할 수 있는 방법을 생각해보았다. 일단, 2개의 사진을 비교하는 경우에 전체를 다 비교하는 것이 아니라 특징이 뚜렷하다고 할 수 있는 부분을 정해서 그 부분만을 이용해서 비교했다. 가장 효율적인 방법은 사진을 그리드 형으로 나누고, 가운데 부분만을 비교할 것이다. 여기서 더 빠르게 실행할 수 있기 위해, u와 v의 값의 범위를 줄이고 x와 y를 확인하는 빈도를 줄이는 방법을 택했다. 각 변수는 정확도를 떨어뜨리지 않는 가장 최선의 숫자를 임의로 기입했다. (각 변수는 임의로 변경 가능하다.)

+ 속도는 N=11, plus=3일 때가 가장 빠르고 정확도가 괜찮은 코드였지만, 실행시간이 1초에서 2초로 늘어남에 따라 더 안정적으로 실행될 수 있도록 N=5, plus=5로 변경하였다.

```
        int N = 5;
        int n = N / 2;
        int m = N / 2 + 1;

        int M = 25;

        int plus = 5;

        for (int v = (-h / 3) / M; v <= (h / 3) / M; v += 1)
                for (int u = -w / M; u <= w / M; u += 1)

                        for (int y = (h / 3) / N * n; y < (h / 3) / N * m; y+=plus)
```

```
for (int x = w / N * n; x < w / N * m; x+=plus) { . . .
```

N : 그리드 열의 개수, n, m : 가운데 조각의 범위, M : u, v 값의 범위를 줄임, plus : x, y를 건너뛰는 수

그 다음으로 해야 하는 작업은 각각을 비교하는 작업이다. float 형 변수 err1, err2에 각각 비교한 차이 값을 누적하고, int 형 변수인 ct를 반복문이 돈 횟수를 저장하여 평균을 내주었다. 이때 차이 값의 평균이 가장 작은 지점이 유사도가 가장 높다고 볼 수 있기에, 이 값과 좌표를 저장했다. 주의할 점은 반복문을 돌면서 비교당하는 사진 외에 비교하는 사진 2장에서 x, y값이 사진의 size를 넘어갈 수 있기 때문에 이를 방지해야 했다.

+ err1과 err2의 값을 계산하는 방법 외에 왜 이렇게 계산 했는지의 이유는 위를 참고 바랍니다.

이렇게 얻어진 좌표들로 사진을 평행이동 시켜서 IplImage형 변수 dst에 저장하기 위해, div[0~2]의 사진들의 각각 가지고 있는 가장 강한 B, G, R element만을 Set한다.

```
CvScalar g;
g.val[1] = f1.val[1];
g.val[0] = f2.val[0];
g.val[2] = f3.val[2];
cvSet2D(dst, y, x, g);
```

이렇게 하면 자동으로 사진이 정렬되며, 칼라로 나오는 모습을 볼 수 있다.

c) 시행 착오

이 문제를 풀면서 정말 많은 시행 착오가 있었지만, 대표적인 것들로 간추려서 작성하겠다.

1) 반복문 깨기 - 1

이 생각을 하게 된 것은 min_err~를 계산할 때, 쓸모 없는 연산을 많이 하게 된다는 것이었다. 실행시간을 줄이기 위해선 반복문을 도는 것을 최대한 줄여야 했기 때문이다. 그래서 생각한 아이디어는 u, v가 양수 일 때, 둘 중 하나만 양수일 때, 둘 다 음수 일 때로 나누어서 해보았다. 차이 값 평균을 계산하고 이것의 최솟값이 최신화 될 때, u, v의 범위가 정해지는 줄 알았다. 하지만 그것은 특정 사진에만 해당되는 내용이었고, Normalize할 수 없는 것이었다. 다음은 이에 대한 코드 일부분이다.

```
int flag = 0;          //          u, v     0 : - - , 1 : + - ,2: - + ,3: + +
. . .
if (flag == 0)                    //          - -
          if (!(u <= 0 && v <= 0)) break;

else if (flag == 1)               //          + -
          if (!(u >= 0 && v <= 0)) break;

else if (flag == 2)  //          - +
          if (!(u <= 0 && v >= 0)) break;

else if (flag == 3)  //          + +
          if (!(u >= 0 && v >= 0)) break;
. . .
int findFlag(int u, int v) {

          if (u <= 0 && v <= 0) {
                    return 0;
          }
          else if (u >= 0 && v <= 0) {
                    return 1;
          }
          else if (u <= 0 && v >= 0) {
                    return 2;
          }
          else {
                    return 3;
          }

}
```

2) 반복문 깨기 – 2

이 방법을 생각하게 된 것도 위의 이유와 같다. 하지만 이 방법은 int 형 변수 cnt를 통해 차이값이 최신화 되지 않는 횟수를 세고 일정 수준을 넘어서면 반복문을 break하는 아이디어였다. 사실 좀 깊게 생각해보면 당연히 되지 않을 방법인데, 시간에 대한 지나친 집착으로 인해 나오게 된 잘못된 생각이었다. 당연히 일정 수준(일정 범위)를 정하지 못해 파기했다.

3) 시간 줄이기

시간을 줄이기 위해 코드의 전반적인 부분을 손봤을 정도로 고쳐보지 않은 곳이 없다. 그 중에

서 가장 긴가민가했던 부분은 바로 u, v에 관한 for구문이었다. 당연한 이야기이겠지만, u, v를 많이 더할수록 코드의 시간은 당연히 줄어들게 된다. 하지만 이는 정확히 사진을 정렬하는 데에는 최악의 선택이다. x, y값은 사진의 특정점을 골라서 비교하면 되지만, u, v는 이동하는 좌표이기 때문에 1픽셀이라도 어긋나면 결과물에서 바로 사진이 티가 나게 된다. 처음에는 너무 여러가지 코드를 작성한 나머지 이 부분을 망각하고, 어찌해야 사진이 제대로 정렬되는 것인지에 대해 매우 긴 고민을 하였다.

이상입니다.