멀티미디어 프로그래밍 과제 5 보고서

소프트웨어학과
20003325 이태혁

아이디어 및 코딩과정

Affine변환을 수행하는 행렬을 생성하는 함수들을 만들어두고, 행렬의 곱을 이용하여 Affine변환 행렬들을 곱하여 최종적인 변환행렬을 src에 적용시킨다. 이 때 회전각도, 크기배율, 이동거리는 모두 단순한 float변수(아래 사진의 4개 변수)

```
float theta = 0.0f; // 회전각도 (degree)
float scale = 1.0f; // 확대축소 비율
float tx = 0.f; // 이미지 이동 정도
float ty = 0.f;
```

가 기억한다는 점이 중요하다. 이 정보들이 변환행렬이나 dst이미지가 기억한다고 생각하여서 발생한 문제들 때문에 굉장히 많이 고생했다.

int draging = 0 이라는 변수를 생성하여 일반 L버튼 클릭시에는 draging = 1, shift + L 버튼 클릭시에는 draging = 2 로 바뀌도록 해서 일반드래그와 shift드래그를 분리했다.

```
int draging = 0; // 그냥 드래그 하면 1, shift + 클릭 후 드래그 하면 2로 설정
    if (event == CV_EVENT_LBUTTONDOWN) {
        draging = 1;
    }
    if (event == CV_EVENT_LBUTTONDOWN && (flags & CV_EVENT_FLAG_SHIFTKEY) == CV_EVENT_FLAG_SHIFTKEY) {
        draging = 2;
    }
```

(draging 변수 설정 파트)

마우스 드래그 시 일어나는 과정

```
float RM[3][3]; // 회전행렬
float SM[3][3]; // 확대축소행렬
float TM[3][3]; // 이동행렬
float TM1[3][3]; // 피봇 ->원점 이동행렬
float TM2[3][3]; // 원점 -> 피봇 이동행렬
```

(변환행렬 정의 이미지)

1. 피봇좌표가 원점으로 이동하고 그에 따라 이미지도 이동 (TM1)
2. shift + 드래그 시에 이미지와 피봇좌표 이동 (TM)
3. 이미지 확대축소 (SM)
4. 이미지 회전 (RM)
5. 피봇좌표와 이미지를 원점에서 원래위치로 이동 (TM2)

1~5번 까지의 행렬들을 곱으로 연결하는데, 이때 역변형을 고려하여 순서를 반대로 하여 곱해준다.

```
// 행렬 곱하는 순서
// M = TM2 RM SM TM TM1 rem 정방향
// IM = rem TM1 TM SM RM TM2 역방향
```

기본적으로 모든 변환행렬은 기본행렬로 초기화 되어있으므로 shift + 드래그나 일반드래그중 하나만 수행되어서 반대쪽의 행렬이 정상적으로 생성되지 않으면 어떡하지? 같은 고민은 할 필요가 없다. 어처피 기본행렬은 어떤행렬을 곱해도 변화가 없기 때문이다.

또한 3번과 4번의 순서가 바뀌면 안된다. 회전한 후 이미지를 늘리거나 줄이면 이미지의 모양이 틀어지기 때문이다.

수업시간에는 원점을 중심으로 회전하거나 크기변환 했지만 여기서는 계속 움직여야 하는 피봇좌표를 중심으로 이동해야 하기 때문에 피봇좌표를 P로 설정하여 피봇좌표와 현재 클릭좌표와의 거리차이를 반영하여 코드를 작성하였다.

```
// 회전
float theta1 = atan2(pt1.y - P.y, pt1.x - P.x);
float theta2 = atan2(pt2.y - P.y, pt2.x - P.x);
```

```
// 확대축소
float r1 = sqrt((pt1.x - P.x) * (pt1.x - P.x) + (pt1.y - P.y) * (pt1.y - P.y));
float r2 = sqrt((pt2.x - P.x) * (pt2.x - P.x) + (pt2.y - P.y) * (pt2.y - P.y));
```

(회전, 확대축소 코드 이미지)

R버튼 클릭 시에 클릭한 곳으로 피봇좌표가 이동하도록 하고, 바뀌기 이전 피봇에서 일어났던 변환들을 모두 rem행렬에 저장하도록 하였다. (이렇게 하는 이유는 시행착오 파트에 기술하였다) rem행렬에 저장할 때는 copyMatrix라는 단순한 행렬 복사함수를 만들어서 사용하였다.

```
if (event == CV_EVENT_RBUTTONDOWN) {
    P = cvPoint(x, y); // 오른쪽버튼 클릭 시 피봇좌표를 저장
    copyMatrix(IM, rem); // IM을 rem에 저장
```

(R버튼 클릭 시 코드 이미지)

```
// 행렬 복사 함수
void copyMatrix(float M[][3], float copy_M[][3]) {
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            copy_M[i][j] = M[i][j];
}
```

(copyMatrix 함수 이미지)

시행착오

1. Shift + Lbutton 클릭 시에 일반 L버튼까지 작동하는 문제

Shift + Lbutton문에서 마지막에 return을 넣어주고 코드의 위치를 일반 L버튼 클릭보다 위쪽으로 올려주어서 shift클릭 시에 딱 이 절만 작동하고 함수가 종료되도록 하여 해결해였다.

```
if (event == CV_EVENT_LBUTTONDOWN && (flags & CV_EVENT_FLAG_SHIFTKEY) == CV_EVENT_FLAG_SHIFTKEY) {
    pt1 = cvPoint(x, y); // 클릭한 좌표를 저장
    draging = 2;

    return; // 아래에 그냥 L버튼을 클릭한 경우도 동작하는 것을 막기 위해 리턴으로 함수를 종료해 주
}
```

2. 모든 변환행렬이 한번에 src에 적용되는 문제

이부분이 이번 과제에서 가장 시간을 오래 잡아먹었다. 결과 이미지의 변환이 예측할 수 없는 방향으로 계속 움직이고 오류가 뜨는것도 아니어서 코드를 처음부터 차근차근 모두 읽어보며 모든 코드를 완벽하게 이해하지 않고서는 잡아낼 수가 없었다.

오른쪽 버튼 클릭시에 피봇좌표가 변경되는데, 그 시점부터는 회전과 확대축소가 새로운 피봇을 기준으로 이루어져야 한다. 그리고 당연하게도 그 전까지의 변환은 바뀌기 전의 피봇을 기준으로 변환되어야 한다.

하지만 나의 이전 코드는 모든 변환정보를 theta, scale, tx, ty에 저장한 후 한번에 현재 피봇을 기준으로 변환행렬을 생성하였다. 따라서 마지막 피봇좌표를 기준으로 한번에 모든 변환이 이루어져서 이전의 피봇을 기준으로 했던 변환들이 무시되어 결과이미지가 이상하게 출력되었던 것이다.

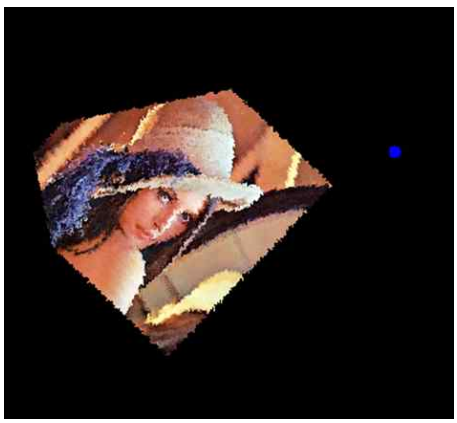
이 문제를 해결할 때 처음에는 새로운 피봇이 생성될 때 src를 현재 dst로 갱신을 하고, 갱신된 src를 변형하는 하는 방향으로 코드를 구현했었다. 그러자 변형을 하면 할수록 이미지의 픽셀이 무너졌고, 이미지가 화면 밖으로 한번 나가면 나간부분이 소멸되는 현상이 발생했다.

```
if (event == CV_EVENT_RBUTTONDOWN) {
    P = cvPoint(x, y);

    // 변환값 모두 초기화
    theta = 0.0f;
    scale = 1.0f;
    tx = 0.f;
    ty = 0.f;

    cvCopy(dst, buf);
    cvCopy(dst, src); // src 갱신
```

(dst로 src를 갱신해주는 코드)



(실행결과)

따라서 dst이미지를 기억하는 것이 아닌, 이전까지의 변환행렬을 기억하는 방향으로 코드를 수정하였다. R버튼이 클릭되어 피봇좌표가 변화할 때 IM에 저장되어있는 이전피봇에서의 변환을 copyMatrix 함수를 이용하여 rem이라는 행렬에 저장하고, 새로운 피봇에서의

변환행렬 생성 시에 rem행렬을 가장 먼저 곱해주어서 이전의 변환들이 반영될 수 있도록 하였다.

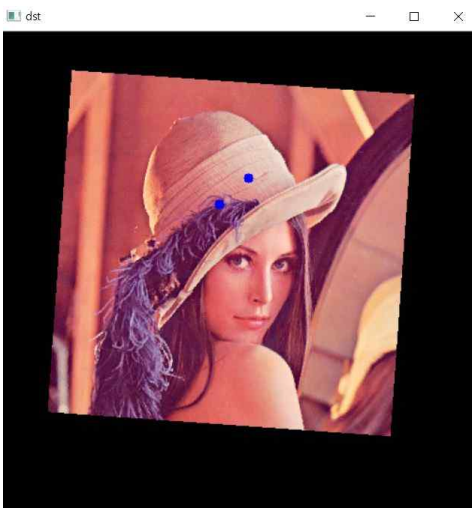
```
if (event == CV_EVENT_RBUTTONDOWN) {
    P = cvPoint(x, y); // 오른쪽버튼 클릭 시 피봇좌표를 저장
    copyMatrix(IM, rem); // IM을 rem에 저장
```

(rem행렬에 이전까지의 변환 저장)

결론은 같은 종류의 변환행렬이라도 한번에 적용하면 안되고 반드시 순서대로 적용하고 넘어가야 한다는 것이다.

그리고 R버튼 클릭 시에 모든 변수들을 다시 초기화해줌으로써 다음 피봇에서의 변환행렬이 기본행렬로 세팅되도록 해야 한다. 그래야 쓰레기값이 들어가지 않아 필요한 변환만 정상적으로 수행할 수 있다.

3. R클릭으로 피봇좌표 변경 시 피봇 표시가 두개가 되는 현상



dst이미지 위에 현재의 피봇원이 표시되고 새로운 피봇을 생성하면 그 좌표까지 원이 하나

더 찍히게 된다. 이 문제를 해결하기 위해 dst를 복사한 이미지인 buf이미지를 생성하고 buf위에 피봇을 찍어준 후 출력한다. 이렇게 하면 dst이미지는 피봇표시가 없는 깨끗한 이미지이기 때문에 buf위에 점을 찍으면 하나씩만 찍히도록 출력할 수 있다.

4. setMultiplyMatrix 함수 사용시 주의할 점

setMultiplyMatrix(M1, M1, M2) 이런식으로 같은 행렬을 두번 넣어주면 안된다.
왜냐하면 인자로 배열의 주소값이 들어가기 때문에 +=로 중첩되는 과정에서 증가한 배열이 다시 함수에 들어가기 때문에 정상적으로 연산이 되지 않고 값이 중첩되면서 점점 더 커지게 된다. 따라서 M1에 M2를 곱한값을 다시 M1에 주고싶다면 M1을 복사한 행렬을 하나 더 만들어서 함수의 인자에 서로 다른 3개의 행렬을 넣어주어야 한다.