멀티미디어프로그래밍

과제 5 보고서

제출일	2024. 06. 03	전공	소프트웨어학과
과목	멀티미디어프로그래밍	학번	21011746
담당교수	박상일 교수님	이름	양현석

1. Image Homography

a) 문제 분석

주어진 코드 안에 homography를 이용해서 이미지를 변형하는 문제입니다. 3차원 육면체를 그리는 코드는 주어졌고, 각 면의 네 꼭지점의 2차원 위치(dst)에 입력받은 이미지의 네 꼭지점을 맞추는 방식으로 구현해야 합니다. 점이 4개, 즉 8개의 변수(H33 제외)와 이에 관련한 8개의 방정식을 풀면 되는 문제입니다. 이는 역행렬 연산을 이용해 간단히 구현할 수 있습니다.

b) 문제 풀이

가장 먼저 해야 할 것은 입력 받을 이미지의 4개의 기준점과 위치시킬 4개의 점을 찾아 저장하는 것입니다. 육면체는 항상 최대 3개의 면까지 볼 수 있습니다. 다음과 같이 코드를 작성했습니다.

```
In "function drawImage()"
//        dst의 움직이는 점
CvPoint Q[4];

//        0,0에서 반시계 방향으로
CvPoint SRC[4] = { cvPoint(0,0), cvPoint(0,src->height - 1),
        cvPoint(src->width - 1,src->height - 1), cvPoint(src->width - 1,0)
};
for (int j = 0; j < 4; j++)
                {
                        vec3 p1 = cube[i].pos[j];
                        vec3 p2 = cube[i].pos[(j + 1) % 4];

                        //        배열 Q에 각 지점의 좌표를 저장
                        Q[j].x = p1.x;
                        Q[j].y = p1.y;
...(후략)...
}
```

그 다음은 8*8의 행렬을 구해야 합니다. 하지만 우리가 변형에 직접적으로 사용하는 행렬은 3*3이므로 estimateTransform() 함수의 인자로 3*3 행렬을 넘겨줍니다. estimateTransform() 함수는 우리가 찾아야 하는 변수를 가진 행렬을 찾기 위해 다음과 같이 작동합니다.

(1) 배열 A[8][8], B[1][8] 생성 (사실 B는 1차원 배열이거나, B[8][1]와 같은 형태여도 상관없다)

-> A에 다음과 같이 값을 집어넣습니다.

$(-x_1, -y_1, -1, 0, 0, 0, x_2'x_1, x_2'y_1, x_2')$ -> 인덱스 : 짝수(0 포함)

$(0, 0, 0, -x_1, -y_1, -1, y_2'x_1, y_2'y_1, y_2')$ -> 인덱스 : 홀수

(x1, y1)은 위 코드의 배열 SRC에 해당하는 값들, (x2', y2')은 위 코드의 배열 Q에 해당하는 값들

이다.

-> B에 다음과 같이 값을 집어넣습니다.

-x2' -> 인덱스 : 짝수(0 포함)

-y2' -> 인덱스 : 홀수

그 후에 구하고자 하는 1*8 배열을 임시로 temp라 하겠습니다. 원래 행렬의 연산 식은 다음과 같습니다.

A * temp = B (*은 내적 연산을 의미)

-> temp = A^(-1) * B (^(-1)은 역행렬을 의미)

우리는 A, B의 모든 값을 알고 있기 때문에(H33은 1로 설정), A의 역행렬만 구하면 임시 배열 temp를 바로 구할 수 있습니다.

여기서 주의할 점은, H33을 1로 두었기 때문에 A' * temp' = 0 (temp'는 1*9 행렬)의 상수항이 되는 부분을 이항시켜서 위의 꼴이 아닌 A * temp = B 의 꼴로 만든 것이다.

(2) A의 역행렬 invA를 구합니다.

기본 코드에 제공되어 있는 "MatrixInverse.h"를 include하여 사용합니다.

(3) invA와 B를 내적 연산하여 temp를 구합니다.

(4) temp는 1*8 행렬이므로 3*3 행렬로 변환시킵니다. (convertMatrix() 사용)

다음 코드는 이 과정의 구현입니다.

```
//		Homography를 하기 위해서 8개의 변수(H33 제외)를 찾는다.
void estimateTransform(float M[][3], CvPoint P[], CvPoint Q[]) {

	float A[8][8] = { 0 };
	float B[1][8] = { 0 };


	for (int i = 0; i < 8; i++) {
		if (i % 2 == 0) {
			A[i][0] = -P[i / 2].x;
			A[i][1] = -P[i / 2].y;
			A[i][2] = -1;
			A[i][6] = Q[i / 2].x * P[i / 2].x;
			A[i][7] = Q[i / 2].x * P[i / 2].y;

			B[0][i] = -Q[i / 2].x;
		}
		else {
			A[i][3] = -P[i / 2].x;
			A[i][4] = -P[i / 2].y;
			A[i][5] = -1;
			A[i][6] = Q[i / 2].y * P[i / 2].x;
```

```
                A[i][7] = Q[i / 2].y * P[i / 2].y;

                B[0][i] = -Q[i / 2].y;
            }
        }

        float invA[8][8];
        float temp[1][8];
        InverseMatrixGJ8(A, invA);
        multiplyMatrix(temp, invA, B);

        convertMatrix(temp, M);
}
```

multiplyMatrix() 함수에 대해서 조금 더 자세히 설명하겠습니다. 이전까지 3*3 행렬의 내적과는 다르게 이번에는 8*8 행렬과 1*8 행렬의 내적을 계산해야 하기 때문입니다. 변형된 코드는 다음과 같습니다.

```
void multiplyMatrix(float M[][8], float A[][8], float B[][8]) {
        for (int i = 0; i < 8; i++) {
            M[0][i] = 0.0f;
            for (int j = 0; j < 8; j++) {
                M[0][i] += A[i][j] * B[0][j];
            }
        }
}
```

사용된 함수인 convertMatrix()의 구현에 대해서는 코드만 첨부하겠습니다. 1*8 행렬을 3*3 행렬로 변환시키는 과정입니다. (H33을 1로 고정했기에 가능)

```
void convertMatrix(float M[][8],float dstM[][3]) {
        float Mat[3][3];

        int idx = 0;
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                if (idx == 8) {      //         H33
                    Mat[i][j] = 1;
                    break;
                }
                Mat[i][j] = M[0][idx++];
            }
```

```
        }

        InverseMatrixGJ3(Mat, dstM);
}
```

참고로 역변형을 사용하기 위해서 생성한 행렬 Mat를 역행렬로 다시 계산해줍니다.

마지막으로 앞의 과정들을 통해 구한 행렬을 통해서 dst에 그려줍니다. 이 알고리즘은 기존 역변형의 코드와 동일합니다.

```
//      역변형으로 출력
void applyInverseTransform(IplImage* src, IplImage* dst, float M[][3]) {

        for (int y2 = 0; y2 < dst->height; y2++) {
                for (int x2 = 0; x2 < dst->width; x2++) {
                        float w2 = 1;

                        float x1 = M[0][0] * x2 + M[0][1] * y2 + M[0][2] * w2;
                        float y1 = M[1][0] * x2 + M[1][1] * y2 + M[1][2] * w2;
                        float w1 = M[2][0] * x2 + M[2][1] * y2 + M[2][2] * w2;
                        x1 /= w1;
                        y1 /= w1;

                        if (x1<0 || x1>src->width - 1) continue;
                        if (y1<0 || y1>src->height - 1) continue;
                        CvScalar f = cvGet2D(src, y1, x1);
                        cvSet2D(dst, y2, x2, f);

                }
        }
}
```

c) 시행 착오

Homography의 개념을 이해하는데 어려웠을 뿐, 이를 구현하는 과정은 어렵지 않았습니다. 하지만 크게 실수하거나 아쉬웠던 부분들을 되짚어보겠습니다.

(1) 변수 입력에 주의하자.

변수를 잘못 쓰는 경우는 사실 그리 많지 않습니다. 하지만 i,j 가 있는 반복문을 사용할 때 종종 일어납니다. multiplyMatrix()를 구현할 때 이 실수를 해서 행렬 연산이 올바르게 되지 않았었습니다. 다음은 그 설명을 돕기 위한 코드입니다.

```
void multiplyMatrix(float M[][8], float A[][8], float B[][8]) {
        for (int i = 0; i < 8; i++) {
                M[0][i] = 0.0f;
                for (int j = 0; j < 8; j++) {
                        M[0][i] += A[i][j] * B[0][j]; -> j를 i라고 작성했었음.
                }
        }
}
```

사실 이런 단순한 실수를 찾는 것이 아주 어렵다고 생각합니다. 이런 작은 실수를 줄이는 코드가 좋은 코드라고 생각하게 되기도 했습니다.

(2) 속도적인 측면

사실 이 부분을 고쳐보려고 여러가지 시도를 해보았지만, 제가 보기엔 역행렬을 구하는 함수가 4중 반복문으로 되어있기에 프로그램이 느려지는 것이라고 생각했습니다. 이 부분은 그래서 고칠 수 없다고 판단했고, 만일 기회가 된다면 교수님께 여쭈어 보는 것이 좋을 것 같다고 생각했습니다.

이상입니다.