증강현실
Final Homework

경북대학교 컴퓨터학부
2014105018
김성현

1. 과제 개요

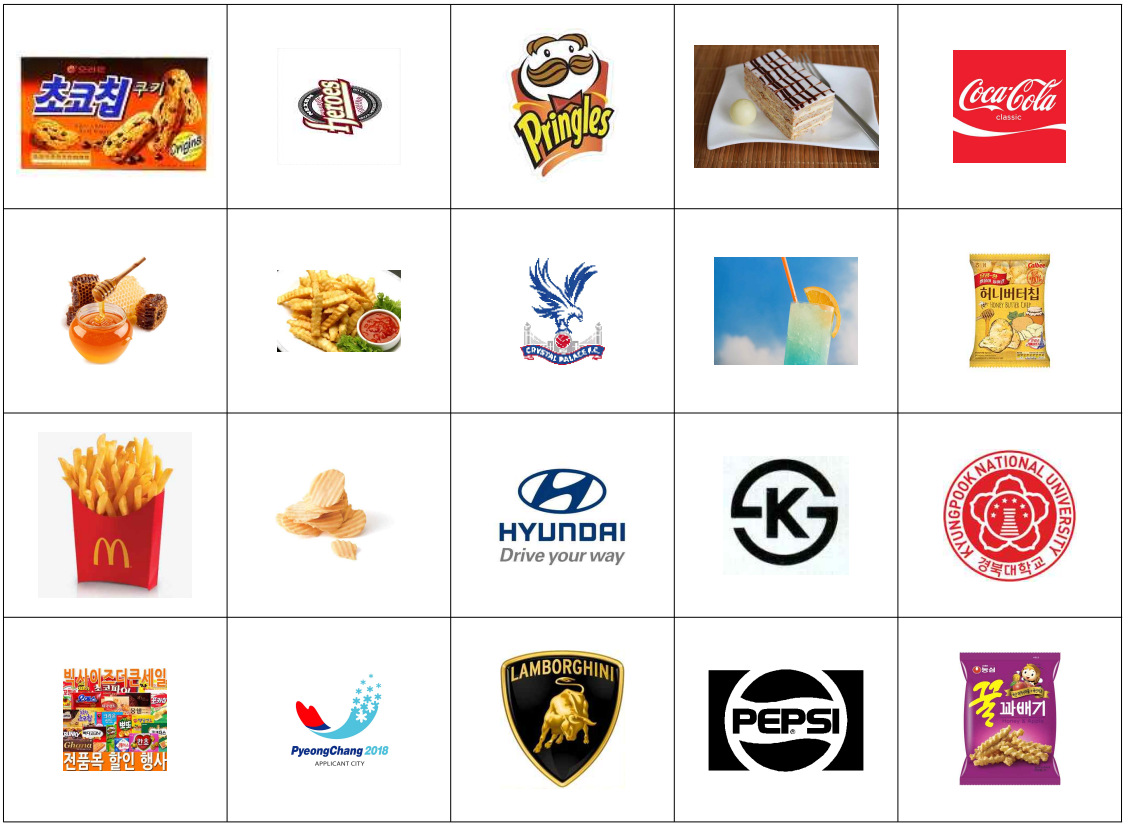
첫 번째 과제는 Multi-marker tracking에 관한 프로젝트이다. Multi-marker tracking에서 2개의 마커를 사용하여 트래킹을 진행하여 이 두 개 마커 사이의 거리를 측정하는 것이 과제의 목표이다. 실제 두 마커 사이의 거리와 영상에서 도출한 두 마커 사이의 거리가 똑같은지 비교해보기로 한다.

두 번째 과제는 가위, 바위, 보로 interaction을 진행하는 프로젝트이다. Multi-marker tracking으로 2개의 마커를 인식하여 한 쪽에는 큐브, 한 쪽에는 주전자 오브젝트를 올린다. 그 후 영상에 바위를 인식시키면 두 오브젝트가 톱니바퀴식으로 회전한다. 가위를 인식시키면 오브젝트들의 색상을 R->G->B순서로 변화시킨다. 보를 인식시키면 기존에 solid형태였던 오브젝트를 wire로 변경시킨다. 이 프로젝트는 간단한 interaction을 해보는 것이 목표이다.

2. 프로젝트 환경

프로젝트는 Microsoft Windows 10 운영체제에서 Microsoft Visual Studio 2013 버전을 이용하여 진행하였으며, OpenCV는 3.0버전을 사용하였다.

3. 사용한 마커



인식이 잘 되는 마커를 찾기 위해 여러 가지의 그림을 뽑아 실험을 해보았다. 그 결과 아래 두 개의 마커가 가자 인식이 잘되어 사용하게 되었다.

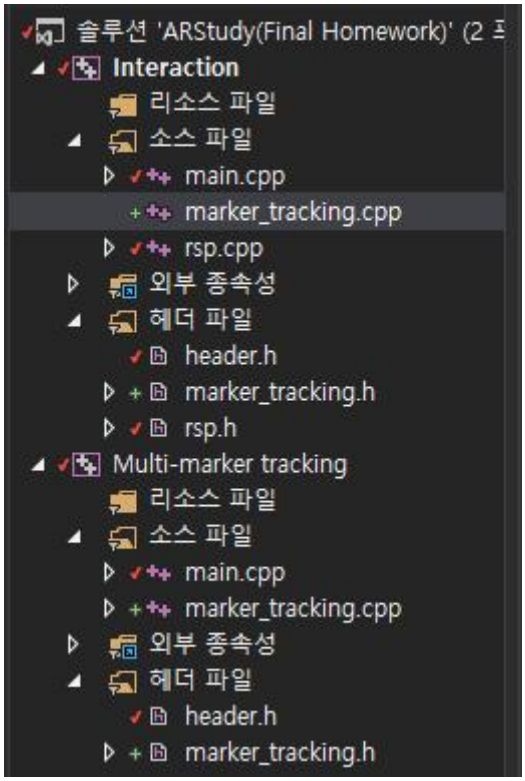
4. 사용한 웹캠

이 프로젝트에 사용된 웹캠은 삼성전자 아티브 NT450R5G-X58L 노트북의 웹캠이다.

5. 프로그램 파일

Multi-marker tracking 프로젝트의 main.cpp에 첫 번째 프로젝트가 구현되었고, Interaction 프로젝트의 main.cpp에 두 번째 프로젝트가 구현되었다. 이 두 프로젝트 모두 이전 과제에서 진행했던 프로젝트를 응용하는 과제이므로 이전 프로젝트를 따로 c와 헤더파일로 만들어 가져와 사용하였다.

marker_tracking.cpp와 marker_tracking.h는 예전에 진행했던 multi-makrer tracking 프로젝트에서 인식하는 주요 부분을 가져온 것이며, rsp.cpp와 rsp.h는 예전에 진행했던 가위, 바위, 보 인식 프로그램의 주요 부분을 가져온 것이다. 이 과제에서는 각 과제에 맞게 어느 정도 고쳐서 사용하였다. 일단 기존에 만든 마커 트래킹과 가위, 바위, 보 인식 프로그램을 먼저 간단히 살펴보고 본 과제로 넘어가겠다.



6. header.h

Multi-marker tracking과 Interaction 과제 모두 동일하게 기본적으로 include 해야하는 것을 담은 헤더파일이다.

```
#pragma once

#include <tchar.h>
#include <math.h>
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <assert.h>
#include <fstream>
#include <Windows.h>
#include <vector>

#pragma comment(lib, "GL/glew32.lib")
#pragma comment(lib, "GL/GLAUX.lib")
#pragma comment(lib, "GL/glut32.lib")
#include "GL/glew.h"
#include "GL/glut.h"
#include "GL/GLAUX.H"

#include "opencv2₩highgui₩highgui.hpp"
#include "opencv2₩opencv.hpp"

#pragma comment(lib, "opencv_world300d.lib")

using namespace std;
using namespace cv;
```

7. marker_tracking.h

이전에 만든 marker_tracking 프로그램의 선언부분을 담은 헤더파일이다.

```
const int MARKERNUM = 2;
```

동시에 인식할 마커의 수

```
//cm단위
const int MARKER_HEIGHT = 8;
const int MARKER_WIDTH = 6;
```

마커의 실제 크기 (cm단위)

```
const int FRAME_WIDTH = 640;
const int FRAME_HEIGHT = 480;
```

웹캠 화면의 크기

```
const int zNear = 1.0;
const int zFar = 1000000;
const double objectSize = 1.5;
const double PI = 3.14159265359;
```

필요한 변수들

(objectSize는 teapot과 cube의 크기 조절을 위한 변수)

```
const std::string nmarker[] = { "./picture/nexen.png", "./picture/pringles.jpg" };
```

인식할 마커 파일들의 경로

```
void init(void);
void convertFromCaemraToOpenGLProjection(double* mGL);
bool calculatePoseFromH(const cv::Mat& H, cv::Mat& R, cv::Mat& T);
void processVideoCapture(void);
void display(void);
void idle(void);
void reshape(int w, int h);

void printSolidTeapot();
void printWireTeapot();
void printSolidCube();
void printWireCube();
```

marker tracking에 필요한 함수들

8. marker_tracking.cpp

1) 선언

우선 marker_traking.h를 선언한 뒤 추가적으로 쓰일 변수들을 선언한다.

```
#include "marker_tracking.h"

cv::Mat gMarkerImg[MARKERNUM];  ///< 마커 이미지
cv::Mat gSceneImg;  ///< 카메라로 캡쳐한 이미지
cv::Mat gOpenGLImg[MARKERNUM];  ///< OpenGL로 렌더링할 이미지
cv::VideoCapture gVideoCapture; ///< 카메라 캡쳐 객체

cv::Ptr<cv::ORB> detector[MARKERNUM];   ///< ORB 특징점 추출기
cv::Ptr<cv::DescriptorMatcher> matcher[MARKERNUM];  ///< ORB 특징정보 매칭 객체

///< 마커 및 카메라로 캡쳐한 이미지의 ORB 특징정보(keypoints)
std::vector<cv::KeyPoint> gvMarkerKeypoints[MARKERNUM], gvSceneKeypoints[MARKERNUM];

///< 마커 및 카메라로 캡쳐한 이미지의 ORB 특징정보(descriprtors)
cv::Mat gMarkerDescriptors[MARKERNUM], gSceneDescriptors[MARKERNUM];

cv::Mat E[MARKERNUM];   ///< 마커 좌표계에서 카메라 좌표계로의 변환 행렬
cv::Mat K;  ///< 카메라 내부 파라메터
```

2) init

marker tracking을 하기 위해 초기화하는 함수

```
///< 마커에서 카메라로의 변환 행렬를 초기화 한다.
for (int i = 0; i < MARKERNUM; i++)
    E[i] = cv::Mat::eye(4, 4, CV_64FC1);
K = cv::Mat::eye(3, 3, CV_64FC1);
```

마커의 개수만큼 행렬을 초기화 시킨다.

```
///< 카메라 내부 파라메터 초기화
K.at<double>(0, 0) = 589.381766;
K.at<double>(1, 1) = 570.846095;
K.at<double>(0, 2) = 306.524379;
K.at<double>(1, 2) = 229.923021;
```

캘리브레이션으로 얻은 카메라 내부 파라메터의 값을 넣어 초기화한다.

```
///< 마커 이미지를 읽는다.
for (int i = 0; i < MARKERNUM; i++)
    gMarkerImg[i] = cv::imread(nmarker[i], 0);

int check = 0;
for (check = 0; check < MARKERNUM; check++){
    if (!gMarkerImg[check].data)
        break;
}
```

미리 선언된 경로를 따라 들어가 마커를 읽어들인다.

이후 마커의 개수에 따라 초기화를 진행한다.

```
///< 특징정보 추출기와 매칭 객체 초기화
for (int i = 0; i < MARKERNUM; i++){
    detector[i] = cv::ORB::create();
    matcher[i] = cv::DescriptorMatcher::create("BruteForce-Hamming");
}

///< 마커 영상의 특징정보 추출
for (int i = 0; i < MARKERNUM; i++){
    detector[i]->detect(gMarkerImg[i], gvMarkerKeypoints[i]);
    detector[i]->compute(gMarkerImg[i], gvMarkerKeypoints[i], gMarkerDescriptors[i]);
}


///< 마커 영상의 실제 크기 측정
for (int k = 0; k < MARKERNUM; k++)
{
    for (int i = 0; i < (int)gvMarkerKeypoints[k].size(); i++) {
        gvMarkerKeypoints[k][i].pt.x /= gMarkerImg[k].cols;
        gvMarkerKeypoints[k][i].pt.y /= gMarkerImg[k].rows;

        gvMarkerKeypoints[k][i].pt.x -= 0.5;
        gvMarkerKeypoints[k][i].pt.y -= 0.5;

        gvMarkerKeypoints[k][i].pt.x *= MARKER_WIDTH;
        gvMarkerKeypoints[k][i].pt.y *= MARKER_HEIGHT;
    }
}

///< OpenGL 초기화
glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
```

3) convertFromCaemraToOpenGLProjection

이 함수는 카메라 내부 파라메터에서 OpenGL 내부 파라메터로 변환하는 함수이다.

```
void    convertFromCaemraToOpenGLProjection(double* mGL)
{
    cv::Mat P = cv::Mat::zeros(4, 4, CV_64FC1);

    P.at<double>(0, 0) = 2 * K.at<double>(0, 0) / FRAME_WIDTH;
    P.at<double>(1, 0) = 0;
    P.at<double>(2, 0) = 0;
    P.at<double>(3, 0) = 0;

    P.at<double>(0, 1) = 0;
    P.at<double>(1, 1) = 2 * K.at<double>(1, 1) / FRAME_HEIGHT;
    P.at<double>(2, 1) = 0;
    P.at<double>(3, 1) = 0;

    P.at<double>(0, 2) = 1 - 2 * K.at<double>(0, 2) / FRAME_WIDTH;
    P.at<double>(1, 2) = -1 + (2 * K.at<double>(1, 2) + 2) / FRAME_HEIGHT;
    P.at<double>(2, 2) = (zNear + zFar) / (zNear - zFar);
    P.at<double>(3, 2) = -1;

    P.at<double>(0, 3) = 0;
    P.at<double>(1, 3) = 0;
    P.at<double>(2, 3) = 2 * zNear*zFar / (zNear - zFar);
    P.at<double>(3, 3) = 0;

    for (int ix = 0; ix < 4; ix++)
    {
        for (int iy = 0; iy < 4; iy++)
        {
            mGL[ix * 4 + iy] = P.at<double>(iy, ix);
        }
    }
```

4) calculatePoseFromH

이 함수는 호모그래피로부터 마커에서 카메라로의 변환 행렬을 추출하는 함수이다.

```
///< 호모그래피로부터 마커에서 카메라로의 변환 행렬 추출
bool    calculatePoseFromH(const cv::Mat& H, cv::Mat& R, cv::Mat& T)
{
    cv::Mat InvK = K.inv();
    cv::Mat InvH = InvK * H;
    cv::Mat h1 = H.col(0);
    cv::Mat h2 = H.col(1);
    cv::Mat h3 = H.col(2);

    double dbNormV1 = cv::norm(InvH.col(0));

    if (dbNormV1 != 0) {
        InvK /= dbNormV1;

        cv::Mat r1 = InvK * h1;
        cv::Mat r2 = InvK * h2;
        cv::Mat r3 = r1.cross(r2);

        T = InvK * h3;
```

5) processVideoCapture

호모그래피와 관련된 함수 연산을 한다.

```
///< 카메라로부터 영상을 읽고, 특징정보 추출한 후, 마커 영상과의 매칭 및 호모그래피를 추정
///< 추정한 호모그래피로부터 마커에서 카메라로의 변환 행렬 추정
void processVideoCapture(void)
{
    cv::Mat grayImg;

    double markerDist[2][3] = { 0 };

    ///< 카메라로부터 영상획득
    gVideoCapture >> gSceneImg;

    ///< 특징정보 추출을 위하여 흑백영상으로 변환
    cv::cvtColor(gSceneImg, grayImg, CV_BGR2GRAY);

    ///< 카메라로부터 획득한 영상의 특징정보 추출
    for (int i = 0; i < MARKERNUM; i++){
        detector[i]->detect(grayImg, gvSceneKeypoints[i]);
        detector[i]->compute(grayImg, gvSceneKeypoints[i], gSceneDescriptors[i]);
    }

    ///< 마커 특징정보와 매칭 수행
    std::vector<std::vector<cv::DMatch>> matches[MARKERNUM];
    for (int i = 0; i < MARKERNUM; i++){
        matcher[i]->knnMatch(gMarkerDescriptors[i], gSceneDescriptors[i], matches[i], 2);
    }

    std::vector<cv::DMatch> good_matches[MARKERNUM];
```

6) display

실제로 계산한 값들을 이용해 마커 위에 가상의 객체를 출력하는 함수이다.
마커가 여러 개 쓰이므로 for문으로 거의 함수전체를 마커 개수만큼 돌며 마커를 하나씩 출력한다.

```
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    for (int k = 0; k < MARKERNUM; k++){
        ///< 배경 영상 렌더링
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();

        glDrawPixels(gOpenGLImg[k].cols, gOpenGLImg[k].rows, GL_BGR_EXT, GL_UNSIGNED_BYTE, (void *)gOpenGLImg[k].data);

        ///< 마커로부터 카메라로의 변환행렬을 통해 마커좌표계로의 변환
        glMatrixMode(GL_MODELVIEW);

        cv::Mat E1;
        E1 = E[k].t();

        double * a = (double *)E1.data;

        glMultMatrixd((double *)E1.data);
        glLineWidth(2.0f);
        glBegin(GL_LINES);
        glColor3f(1.0f, 0.0f, 0.0f); glVertex3d(0.0, 0.0, 0.0); glVertex3d(10.0, 0.0, 0.0);
        glColor3f(0.0f, 1.0f, 0.0f); glVertex3d(0.0, 0.0, 0.0); glVertex3d(0.0, 10.0, 0.0);
        glColor3f(0.0f, 0.0f, 1.0f); glVertex3d(0.0, 0.0, 0.0); glVertex3d(0.0, 0.0, 10.0);
        glEnd();

        if (k == 0)
            glRotated(0.0, 1.0, 0.0, 0.0);
        else
```

9. rsp.h

```
#include "header.h"

// 주먹 범위
const int rock_low = 14;
const int rock_high = 20;

// 가위 범위
const int scissors_low = 21;
const int scissors_high = 29;

//보 범위
const int paper_low = 31;
//const int paper_high = 41;
const int paper_high = 70;

enum rsp { rock, scissors, paper, none };

int checkNoise(int * noise, int noise_index, int pixel);
int checkPixel(Mat pic);
rsp check_rsp(int pixel, int noise);
void fillMat(Mat * mat, int n);
rsp image_processing(Mat camFrame, Rect box);
```

가위, 바위, 보 프로그램은 손을 인식하는 영역에 손이 얼마나 넓은 면적을 차지하는가에 따라 가위, 바위, 보를 판단한다. 위의 주먹 범위, 가위 범위, 보 범위는 미리 실험을 통해 계산해둔 가위, 바위, 보에 해당하는 픽셀의 차지 영역 값들이다.

enum rsp는 프로그램의 가독성을 위해 선언을 하였다.

그 외에는 프로그램의 함수를 선언하였다.

10. rsp.cpp

1) checkNoise

```
// 살색 픽셀 수 체크
int checkNoise(int * noise, int noise_index, int pixel)
{
    // 프로그램 시작 후 10 번만 실행한다.
    if (noise_index > 10){
        return noise_index;
    }
    // 10번 실행 후 살색 픽셀 수의 평균값을 계산하여 살색 픽셀수를 계산해 놓는다.
    else if (noise_index == 10){
        *noise = *noise / noise_index;
        return noise_index + 1;
    }

    *noise += pixel;
    return noise_index + 1;
}
```

프로그램의 정확도를 위해 따로 만든 함수이다.
프로그램을 실행하면 분명 화면에 내 손이 없음에도 불구하고 픽셀수가 0이 아닌 경우가 있다. 손의 색과 비슷한 색을 가진 물체들이 원인인데, 이것을 미리 계산하여 가위, 바위, 보 판단에 사용하기 위해 함수를 구현하였다. 프로그램을 시작한 후 10번 0이 아닌 픽셀의 퍼센트를 계산하였고, 그 평균을 노이즈 픽셀 퍼센트로 하였다.

2) checkPixel

```
// 0이 아닌 픽셀 수 체크
int checkPixel(Mat pic)
{
    int ret = 0;
    // 0값을 넣은 픽셀 하나를 선언한다.
    Mat zpic = Mat::zeros(Size(1, 1), pic.type());

    // 0 픽셀과 비교하여 아니면 덧셈하여 정보가 있는 픽셀 수를 계산한다.
    for (int i = 0; i < pic.size().width; i++){
        for (int j = 0; j < pic.size().height; j++){
            if (pic.at<Vec3b>(j, i) != zpic.at<Vec3b>(0, 0)){
                ret++;
            }
        }
    }

    return (double)ret / (double)(pic.size().width * pic.size().height) * 100;
}
```

이 프로그램은 손의 색과 비슷한 색을 가진 픽셀의 데이터만을 남겨놓고 아니면 전부 0으로 만들어버린다. 그러므로 결과 데이터에서 0이지 않은 픽셀 수를 세면 손의 면적이 나오게 되는데, 그러기 위해 0인 픽셀 하나를 선언하여 이 픽셀과 다른 픽셀 수를 세어 비율을 계산하였다.

3) check_rsp

```
// 가위 바위 보 판단
rsp check_rsp(int pixel, int noise){
    // 기본적으로 계산해 놓은 데이터들에 의해 가위 바위 보를 판단한다.(픽셀수에 의해 결정된다)
    // 프로그램 시작시 계산한 노이즈 수를 더함으로써 노이즈에 의한 에러를 줄인다.
    if (noise + rock_low <= pixel && pixel <= noise + rock_high){
        return rock;
    }
    else if (noise + scissors_low <= pixel && pixel <= noise + scissors_high){
        return scissors;
    }
    else if (noise + paper_low <= pixel && pixel <= noise + paper_high){
        return paper;
    }

    return none;
}
```

여러 번의 시행착오로 가위, 바위, 보일 때의 화면 상의 픽셀 퍼센트를 계산한 값을 rsp.h에 이미 선언을 해두었는데, 이것을 이용하여 현재 화면에 가위, 바위, 보 중 어느 것이 있는지 판단하는 함수이다. 이 함수에는 선언한 데이터 외에 노이즈 데이터도 들어가는데 현재 픽셀에는 프로그램 시작 시 처음부터 보이던 손의 색 픽셀도 들어있다 라는 가정 하에 선언한 데이터에 이 값을 더해 가위, 바위, 보 판단의 정확도를 높였다.

반환은 선언해둔 enum형식을 사용하였으며 코드의 가독성을 높였다.

4) fillMat

```
// 화면 색 채우기
void fillMat(Mat * mat, int n)
{
    // 필요로 하는 곳의 데이터를 255로 만들어 빨간, 파란, 초록 색 중 하나로 변환시킨다.
    Mat pic = Mat::zeros((*mat).size(), (*mat).type());
    for (int i = 0; i < (*mat).size().width; i++){
        for (int k = 0; k < (*mat).size().height; k++){
            pic.at<Vec3b>(k, i)[n] = 255;
        }
    }

    // 색을 입힌 이미지와 원래 mat이미지를 bitwise하여 기본 이미지의 색을 바꾼다.
    bitwise_and(*mat, pic, *mat);
}
```

가위, 바위, 보에 따라 손에 색을 입히는 작업을 하는 함수이다.
pic이라는 임의의 Mat 변수의 배열에 255를 입힘으로서 색을 설정할 수 있다. [0]는 빨간색, [1]은 초록색, [2]는 빨간색을 가리키며 한 곳이 255고 다른 곳이 0 이면 셋 중 한 색을 가리키게 된다.

이렇게 생성한 색 영상을 bitwise_and로 기존 이미지에 덮어씌우면 기존 이미지의 0이 아닌 부분은 해당 색으로 변하게 된다.

5) image_processing

이 함수는 원래 가위, 바위, 보 인식 프로그램이 하나의 프로그램이였을 때의 메인 부분을 담당한다. 그러므로 만약 다른 프로그램에서 이 가위, 바위, 보를 판단하기 위해서는 이 함수만을 호출하면 되게 하였다. 이 함수의 진행과정을 간단히 설명하면 이렇다.

일단 손을 인식할 범위인 Box와 실제 웹캠의 영상을 함수 파라메터로 받아온다. 일단 이 웹캠에서 박스부분을 따로 떼어내는 작업을 하고 이 부분의 영상을 YCrCb 형식으로 바꾼 뒤, split함수로 Y, Cr, Cb 세 가지로 나눈다. 이 중, Cr, Cb에서 손의 색 부분만을 추출한 뒤, 두 영상을 bitwise_and함수를 이용하여 합친다. 이 영상을 다시 BGR 형식으로 바꾼 뒤 원래 영상과 합치면 일단 손만을 추출해낸 영상이 나온다. 그 후 영상의 픽셀 수를 계산한 뒤, 처음 10번 동안은 영상의 손의 색 픽셀을 계산하게 된다. 이후 계산한 픽셀 수를 기반으로 가위, 바위, 보를 판단하여 색을 입힌 뒤 이것을 원본 영상에 씌운 후 영상을 출력하게 된다. 이 과정에 모두 끝나면 가위, 바위, 보의 결과값을 다시 리턴해주게 된다.

```
Mat image_processing(Mat camFrame, Rect box)
{
    static int noise = 0;

    Mat frame;
    frame = camFrame(box); // 웹캠 정보에서 손을 인식할 부분만 따로 떼어낸다.

    // 기존 이미지를 YCrCb 3개의 정보를 저장한 형태로 변환
    Mat rgbMat, yCbCrMat;
    cvtColor(frame, yCbCrMat, CV_RGB2YCrCb);

    // split으로 3개가 합쳐진 정보를 떼어낸다.
    // splitMatV[0] : Y, splitMatV[1] : Cr, splitMatV[2] : Cb
    vector <Mat> splitMatV;
    split(yCbCrMat, splitMatV);

    // 각각 떼어낸 이미지에서 손의 색과 비슷한 부분만을 추출한다.
    Mat crMat, cbMat;
    inRange(splitMatV[1], Scalar(77), Scalar(150), crMat);
    inRange(splitMatV[2], Scalar(133), Scalar(173), cbMat);

    // bitwise_and를 이용하여 Cr과 Cb 정보중 남아 있는 것을 and 연산하여 손의 이미지를 확정짓는다.
    Mat grayMat;
    bitwise_and(crMat, cbMat, grayMat);

    // 이미지를 BGR형태로 변환
    Mat bgrMat;
    cvtColor(grayMat, bgrMat, CV_GRAY2BGR);
```

11. Multi-marker tracking (과제 1번)

1) 마커 사이의 거리

왼쪽 마커와 오른쪽 마커 사이의 거리를 재보면 약 5.3cm이다.

2) 영상에서의 마커 사이의 거리

이 과제에서 영상 속에서 마커 사이의 거리를 구하기 위해서 2가지 방법을 썼다. 우선 첫 번째는 marker_tracking 프로그램의 E행렬을 사용한 방법이다. E행렬은 마커 좌표계에서 카메라 좌표계로의 변환 행렬이다.

Rotation and translation

$$\begin{bmatrix} X_C \\ Y_C \\ Z_C \\ 1 \end{bmatrix} = \begin{bmatrix} R_{11} & R_{12} & R_{13} & T_1 \\ R_{21} & R_{22} & R_{23} & T_2 \\ R_{31} & R_{32} & R_{33} & T_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_M \\ Y_M \\ Z_M \\ 1 \end{bmatrix}$$

$$= \mathbf{T_{CM}} \begin{bmatrix} X_M \\ Y_M \\ Z_M \\ 1 \end{bmatrix}$$

출처 : 강의자료

E행렬은 이 사진에서의 4x4 행렬을 나타내며, 이중 T1, T2, T3는 평행 이동을 나타내게 된다. 그리하여 일단 이 3개의 값을 빼내어 거리를 측정하였다.
코드는 이러하다.

```
            markerDist[k][0] = T.at<double>(0);
            markerDist[k][1] = T.at<double>(1);
            markerDist[k][1] = T.at<double>(2);
        }
    }
}

double dist = sqrt(pow(markerDist[0][0] - markerDist[1][0], 2) + pow(markerDist[0][1] - markerDist[1][1], 2) + pow(markerDist[0][2] - markerD
cout << dist << endl;

if (gSceneImg.data)
    cv::flip(gSceneImg, gOpenGLImg[0], 0);

glutPostRedisplay();
```

이 코드는 processVideoCapture 함수의 일부분이며, 원래 프로그램에서 추가한 부분이다. markerDist에 위의 T1, T2, T3부분을 저장하여 거리를 측정하였다.

또 다른 방법은 norm을 사용한 방법이다. norm은 선형대수학 및 함수해석학에서 벡터 공간의 원소들에 일종의 '길이' 또는 '크기'를 부여하는 함수이며, opencv에서 지원하는 함수이다.

```
cout << "n : " << norm(E[0], E[1]) << endl;
```

그리하여 norm에 E 두 행렬을 넣어 거리를 측정하는 방법을 사용했으며, 이 코드는 display함수에서 사용하였다.

3) 메인

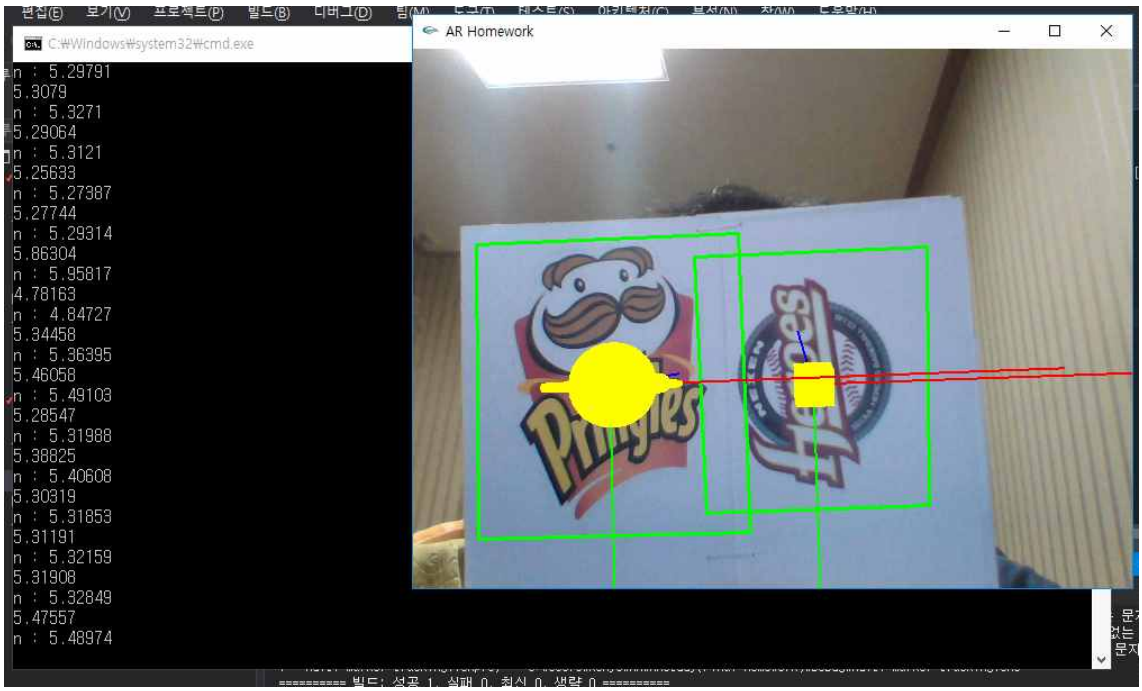
```
#include "marker_tracking.h"

int main(int argc, char** argv)
{
    glutInit(&argc, argv);

    glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
    glutInitWindowSize(FRAME_WIDTH, FRAME_HEIGHT);
    glutCreateWindow("AR Homework");

    init();

    glutDisplayFunc(display);
    glutIdleFunc(idle);
    glutReshapeFunc(reshape);
    glutMainLoop();

    return 0;
}
```

Marker tracking 프로그램의 메인과 똑같다.

4) 결과 화면

화면에 두 개의 마커가 인식이 되었고, 콘솔창에는 두 마커 사이의 거리를 계속 갱신해주고 있는 모습이다. 그냥 숫자는 직접 계산한 거리이며, 'n :' 뒤에 오는

숫자는 norm을 이용해 구한 거리이다. 두 거리는 크게 차이나지 않는 것을 확인할 수 있으며, 실제 거리인 5.3cm와 유사한 거리가 출력되는 것을 볼 수 있다.



5) 분석

E행렬, 즉 카메라 외부 파라미터는 위의 사진을 참고하였을 때, T부분은 두 좌표계 사이의 평행이동을 나타낸다. 두 마커모두 좌표계의 이동전엔 똑같은 위치의 좌표계를 가지고 있으므로, 좌표계의 평행이동 사이의 거리를 구하면 두 마커 사이의 원점이 나온다. 하지만 오차가 다소 나는 것은 마커의 실제 크기를 입력한 값에 어느 정도 오차가 있거나, 실제 마커로 쓰이는 파일 자체의 가로 세로 크기도 두 개가 같지 않다는 것이 영향을 미치는 것 같다.

여기서 E에 대한 R부분이 좌표계 사이의 회전을 나타내는데, 좌표계사이의 평행이동 외에도 회전에 대해서도 거리가 차이가 날 것 같다고 생각하여

```
double dot_list[2][4];
for (int i = 0; i < 2; i++){
    for (int j = 0; j < 4; j++){
        dot_list[i][j] = 0;
        for (int k = 0; k < 4; k++){
            dot_list[i][j] = 1 * E[i].at<double>(j, k);
        }
    }
}

double dist = sqrt(pow(dot_list[0][0] - dot_list[1][0], 2) + pow(dot_list[0][1] - dot_list[1][1], 2) + pow(dot_list[0][2] - dot_list[1][2], 2
cout << "d : " << dist << endl;
```

[1, 1, 1, 1] 행렬을 만들어 E행렬을 직접 곱해 두 결과 행렬 사이의 거리를 측정해보았다. 결과는 norm 연산과 기존에 계산한 것과 크게 차이가 나지 않는 것을 확인할 수 있었다.

6) 실행 영상

주소 : https://youtu.be/ONGYnDMUfsU

12. Interaction (과제 2번)

1) 메인

```
#include "rsp.h"
#include "marker_tracking.h"

int main(int argc, char** argv)
{
    glutInit(&argc, argv);

    glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
    glutInitWindowSize(FRAME_WIDTH, FRAME_HEIGHT);
    glutCreateWindow("AR Homework");

    init();

    glutDisplayFunc(display);
    glutIdleFunc(idle);
    glutReshapeFunc(reshape);
    glutMainLoop();

    return 0;
}
```

Marker tracking 프로그램의 메인과 똑같다.

2) 초기화

```
#include "marker_tracking.h"
#include "rsp.h"
```

marker_tracking.cpp에 rsp.h를 추가하여 marker_tracking 코드에서 가위바위보 판단하는 함수를 사용할 수 있게 한다.

```
rsp rspdata;
Rect box;
int objColor;
int degree;

void init(void)
{
    rspdata = none;
    objColor = 0;
    degree = 0;

    // 실제로 사람의 손을 인식할 사각형 범위를 지정할 사각형
    box.x = 0;
    box.y = 0;
    box.height = 300;
    box.width = 300;
```

가위바위보 결과를 저장할 변수, 손을 인식할 사각형 범위를 지정할 사각형,

오브젝트의 색을 결정할 변수, 오브젝트의 위치를 정할 변수 등을 선언하고 init함수에서 초기화한다.

3) 가위, 바위, 보 판단 함수

processVideoCapture 함수에는 카메라로부터 영상을 획득하는 부분이 있다.

```
///< 카메라로부터 영상획득
gVideoCapture >> gSceneImg;
```

그러므로 이 함수에서 영상을 획득하여 처리하는 일이 모두 끝난 후에 이 영상으로 가위, 바위, 보를 판단하였다.

```
timer++;
if (timer == 11){
    rspdata = image_processing(gSceneImg, box);

    if (rspdata == scissors){
        objColor += 1;
        objColor %= 3;
    }
    timer = 0;
}
else{
    image_processing(gSceneImg, box);
}

if (gSceneImg.data)
    cv::flip(gSceneImg, gOpenGLImg[0], 0);

glutPostRedisplay( );
```

이때 timer는 적당한 수치를 주어, 가위 바위 보에 해당하는 행동을 어느 정도 지속하게 하였다.
rspdata에 가위, 바위, 보의 결과가 들어가게 되며, 그에 해당하는 행동을 하게 하였다.

4) 가위

```
void printSolidTeapot()
{
    if (objColor == 0)
        glColor3f(0.0, 0.0, 1.0f);
    if (objColor == 1)
        glColor3f(1.0f, 0.0, 0.0);
    if (objColor == 2)
        glColor3f(0.0, 1.0f, 0.0);
    glutSolidTeapot(objectSize);
}

void printWireTeapot()
{
    if (objColor == 0)
        glColor3f(0.0, 0.0, 1.0f);
    if (objColor == 1)
        glColor3f(1.0f, 0.0, 0.0);
    if (objColor == 2)
        glColor3f(0.0, 1.0f, 0.0);
    glutWireTeapot(objectSize);
}
```

위의 코드에서 rspdata가 scissors이면 objColor에 1을 더하고 3으로 나누라고 되어있다. 이 의미는 오브젝트 출력함수를 보면 알 수 있다.

objColor에 따라 색이 바뀌며 그 색은 빨강, 초록, 파랑 이 세 가지 색이다. 이 세 가지 색이 번갈아가면서 출력되기 위해 1을 더해 색을 바꾸고 혹시 3을 넘어가면 3으로 나눈 나머지를 저장함으로써 색을 계속 변화시킨다.

5) 바위

```
if (rspdata == rock)
    degree = (degree + 1);
```

display 함수에서 가장 상단에 위치하는 코드이다. rspdata가 rock이면 degree를 1 증가시켜준다. 즉 각도를 일정 수치 더하여 회전을 시킨다.

```
if (k == 0){
    glRotated(degree * 30, 0.0, 0.0, 1.0);
    glTranslated(3 * sin(degree), 3 * cos(degree), 0.0);
}
if (k == 1){
    glRotated(-degree * 30, 0.0, 0.0, 1.0);
    glTranslated(3 * sin(-degree), 3 * cos(-degree), 0.0);
}
```

k가 0일 때, 즉 첫 번째 마커라는 이야기이며, 이 마커는 degree에 대해 회전을 하며 원형으로 이동한다.

k가 1일 때, 즉 두 번째 마커는 -degree에 대해 회전을 하며 원형으로 이동한다. 즉 두 마커는 반대의 방향으로 회전을 하게 되며 톱니바퀴식 회전이란 조건을 만족하게 된다.

회전 시 30을 곱한 것은 회전이 좀 더 잘 보이게 하기 위해서이며, 원형 이동시 3을 곱하는 것은 좌표를 기준으로 3만큼의 반경으로 돌게 하기 위해서이다.

6) 보

```
///< 마커 좌표계의 중심에서 객체 렌더링
if (rspdata == paper){
    if (k == 0){
        printWireCube();
    }
    else{
        printWireTeapot();
    }
}
else{
    if (k == 0){
        printSolidCube();
    }
    else{
        printSolidTeapot();
    }
}
```

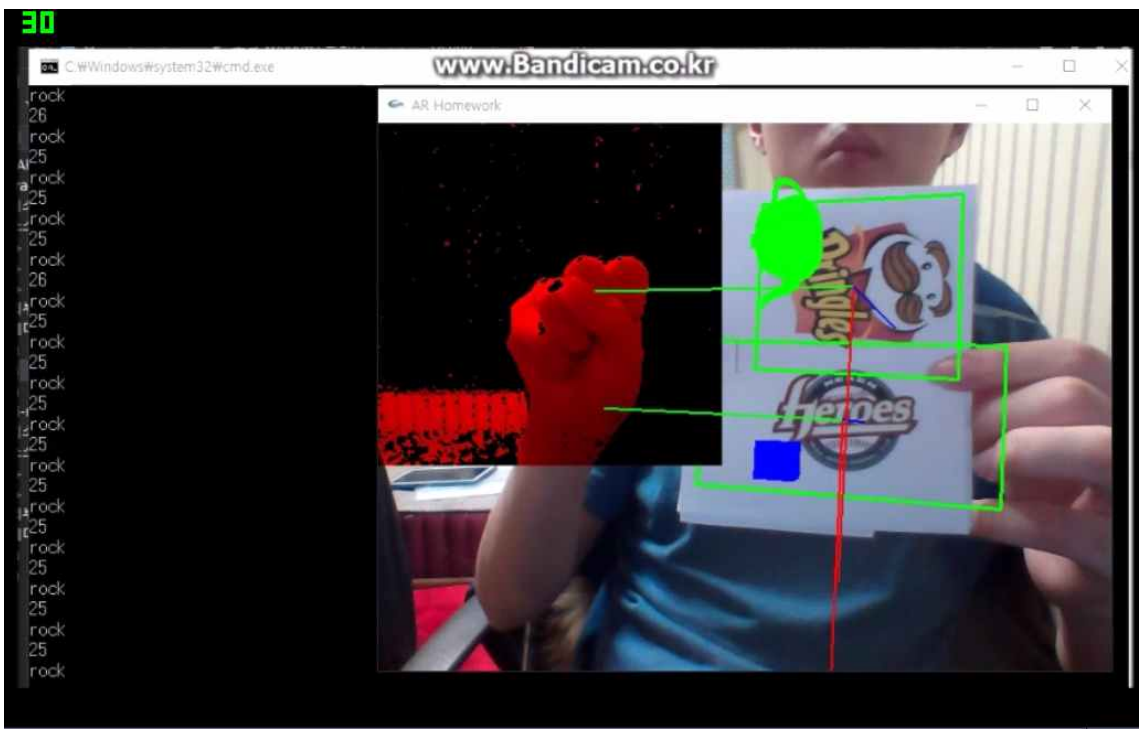
rspdata가 paper이면 printWireCube, printWireTeapot을 실행하게 되고 기존에는 printSolidCube, printSolidTeapot을 실행하게 된다.

```
void printSolidTeapot()
{
    if (objColor == 0)
        glColor3f(0.0, 0.0, 1.0f);
    if (objColor == 1)
        glColor3f(1.0f, 0.0, 0.0);
    if (objColor == 2)
        glColor3f(0.0, 1.0f, 0.0);
    glutSolidTeapot(objectSize);
}

void printWireTeapot()
{
    if (objColor == 0)
        glColor3f(0.0, 0.0, 1.0f);
    if (objColor == 1)
        glColor3f(1.0f, 0.0, 0.0);
    if (objColor == 2)
        glColor3f(0.0, 1.0f, 0.0);
    glutWireTeapot(objectSize);
}
```

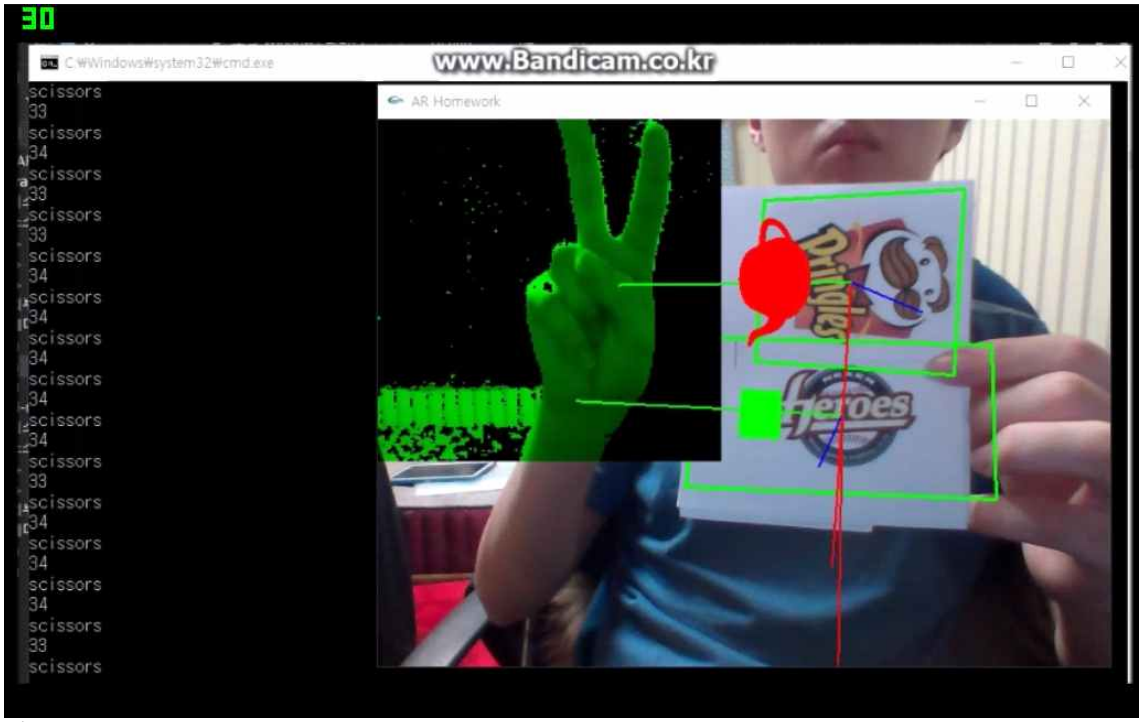
함수를 보면 각 프린트 함수에는 rgb를 색을 설정하는 함수가 먼저 실행되며, 함수에 따라 wire나 solid로 나뉘어 실행이 된다.

7) 실행 화면

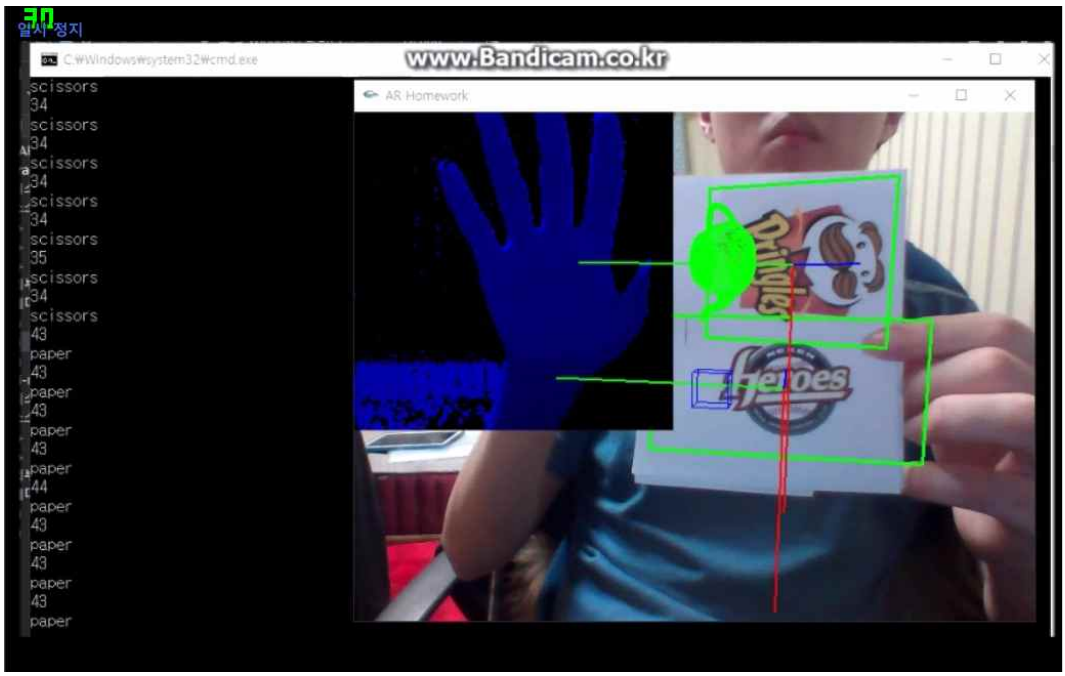
바위



가위



보



8) 실행 영상

주소 : https://youtu.be/BXjLwScrbcc