**T.C.**

**SELÇUK ÜNİVERSİTESİ**

**MÜHENDİSLİK FAKÜLTESİ**

**BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ**

**Bilgisayarla Görmeye Giriş**

**Final Projesi**

**Hazırlayanlar**

**121213085 , Rıza YALÇIN**

**121213008 , Osman Çağrı GENÇ**

**Ders Sorumlusu**

**Yrd. Doç. Dr. Ömer K. BAYKAN**

Final projesinin amacı kalman filtresini kullanarak nesnenin tespit edilip gideceği yerin tahminin yapılmasıdır.

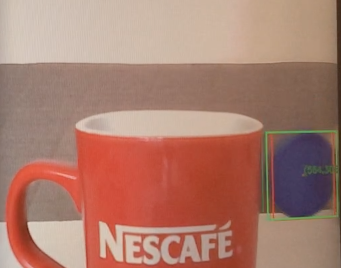
Görüntü üzerinde çalışma yapabilmek için aşağıdaki adımlar gerçekleştirildi.

1. Görüntü frame olarak okundu,
2. GaussianBlur filtresi uygulandı,
3. Görüntünün renk uzayı BGR renk uzayından HSV renk uzayına çevrildi,
4. Görüntüye uygun Hsv renk uzayı değerleri aracılığıyla thresholding uygulandı,
5. Morfolojik işlemler uygulandı ve sonuç olarak nesne tespit edildi.
6. Nesne tespit edildikten sonra nesne 2 adet alan içerisine alınır. İlk alan ( Yeşil alan ) nesnenin tespit edilmiş hali , ikinci alan( Kırmızı alan ) nesnenin hareket yönünün alanıdır.
7. Nesnenin önüne bir engel geldiğinde oluşturulan ikinci alan nesnenin hareket yönünü tahmin etmek için kullanılacaktır.



Resim 1. Nesnenin tespit edilmesi

Yukarıdaki Resim 1. nesne tespit edilme işlemi yapılmıştır. Bu işlem uygulanan algoritmanın ilk 5 adımı ile yapılmıştır.



Resim 2. Nesnenin Alanlarının Belirlenmesi

Yukarıdaki Resim 2. nesne tespit edilme işlemi yapılmıştır. Bu işlem uygulanan algoritmanın 6. adımı ile yapılmıştır.



Resim 3. Nesnenin Önüne Engel Gelme Durumu

Yukarıdaki Resim 3. nesne tespit edildikten sonra önüne bir engel gelmiştir ve hareket yönünün tahmini işlemi başlamıştır. Bu işlem uygulanan algoritmanın 7. adımı ile yapılmıştır. Bu durumdan itibaren nesne için oluşturulan ikinci alan ile kalman filtresi aracılığıyla nesnenin ilk görüntüsünden itibaren oluşturduğu hareket yönüyle hareketinin devam edeceği yönün tahmini yapılacaktır.



Resim 4. Nesnenin Hareket Yönünün Tahmin Edilmesi



Resim 5. Nesnenin Hareket Yönünün Tahmin Edilmesi

Yukarıdaki Resim 3 de nesne tespit edildikten sonra önüne bir engel gelmiştir ve hareket yönünün tahmini işlemi başlamıştır. Resim 3-4-5 de hareket yönünün tahmini yapılmıştır. Nesnenin hareket yönü framenin sonuna kadar devem edecektir ve frameden çıkış yaptığı da varsayıldığında tekrar bir nesne arayışı devam edecektir.

Proje kodu aşağıda yer almaktadır.

// Module "core"

#include <opencv2/core/core.hpp>

// Module "highgui"

#include <opencv2/highgui/highgui.hpp>

// Module "imgproc"

#include <opencv2/imgproc/imgproc.hpp>

// Module "video"

#include <opencv2/video/video.hpp>

// Output

#include <iostream>

// Vector

#include <vector>

using namespace std;

// >>>>> Color to be tracked

#define MIN\_H\_BLUE 200

#define MAX\_H\_BLUE 300

// <<<<< Color to be tracked

int main()

{

// Camera frame

cv::Mat frame;

// >>>> Kalman Filter

int stateSize = 6;

int measSize = 4;

int contrSize = 0;

unsigned int type = CV\_32F;

cv::KalmanFilter kf(stateSize, measSize, contrSize, type);

cv::Mat state(stateSize, 1, type); // [x,y,v\_x,v\_y,w,h]

cv::Mat meas(measSize, 1, type); // [z\_x,z\_y,z\_w,z\_h]

//cv::Mat procNoise(stateSize, 1, type)

// [E\_x,E\_y,E\_v\_x,E\_v\_y,E\_w,E\_h]

// Transition State Matrix A

// Note: set dT at each processing step!

// [ 1 0 dT 0 0 0 ]

// [ 0 1 0 dT 0 0 ]

// [ 0 0 1 0 0 0 ]

// [ 0 0 0 1 0 0 ]

// [ 0 0 0 0 1 0 ]

// [ 0 0 0 0 0 1 ]

cv::setIdentity(kf.transitionMatrix);

// Measure Matrix H

// [ 1 0 0 0 0 0 ]

// [ 0 1 0 0 0 0 ]

// [ 0 0 0 0 1 0 ]

// [ 0 0 0 0 0 1 ]

kf.measurementMatrix = cv::Mat::zeros(measSize, stateSize, type);

kf.measurementMatrix.at<float>(0) = 1.0f;

kf.measurementMatrix.at<float>(7) = 1.0f;

kf.measurementMatrix.at<float>(16) = 1.0f;

kf.measurementMatrix.at<float>(23) = 1.0f;

// Process Noise Covariance Matrix Q

// [ Ex 0 0 0 0 0 ]

// [ 0 Ey 0 0 0 0 ]

// [ 0 0 Ev\_x 0 0 0 ]

// [ 0 0 0 Ev\_y 0 0 ]

// [ 0 0 0 0 Ew 0 ]

// [ 0 0 0 0 0 Eh ]

//cv::setIdentity(kf.processNoiseCov, cv::Scalar(1e-2));

kf.processNoiseCov.at<float>(0) = 1e-2;

kf.processNoiseCov.at<float>(7) = 1e-2;

kf.processNoiseCov.at<float>(14) = 5.0f;

kf.processNoiseCov.at<float>(21) = 5.0f;

kf.processNoiseCov.at<float>(28) = 1e-2;

kf.processNoiseCov.at<float>(35) = 1e-2;

// Measures Noise Covariance Matrix R

cv::setIdentity(kf.measurementNoiseCov, cv::Scalar(1e-1));

// <<<< Kalman Filter

// Camera Index

int idx = 0;

// Camera Capture

cv::VideoCapture cap;

// >>>>> Camera Settings

if (!cap.open(idx))

{

cout << "Webcam not connected.\n" << "Please verify\n";

return EXIT\_FAILURE;

}

cap.set(CV\_CAP\_PROP\_FRAME\_WIDTH, 1024);

cap.set(CV\_CAP\_PROP\_FRAME\_HEIGHT, 768);

// <<<<< Camera Settings

cout << "\nHit 'q' to exit...\n";

char ch = 0;

double ticks = 0;

bool found = false;

int notFoundCount = 0;

// >>>>> Main loop

while (ch != 'q' && ch != 'Q')

{

double precTick = ticks;

ticks = (double) cv::getTickCount();

double dT = (ticks - precTick) / cv::getTickFrequency(); //seconds

// Frame acquisition

cap >> frame;

cv::Mat res;

frame.copyTo( res );

if (found)

{

// >>>> Matrix A

kf.transitionMatrix.at<float>(2) = dT;

kf.transitionMatrix.at<float>(9) = dT;

// <<<< Matrix A

cout << "dT:" << endl << dT << endl;

state = kf.predict();

cout << "State post:" << endl << state << endl;

cv::Rect predRect;

predRect.width = state.at<float>(4);

predRect.height = state.at<float>(5);

predRect.x = state.at<float>(0) - predRect.width / 2;

predRect.y = state.at<float>(1) - predRect.height / 2;

cv::Point center;

center.x = state.at<float>(0);

center.y = state.at<float>(1);

cv::circle(res, center, 2, CV\_RGB(255,0,0), -1);

cv::rectangle(res, predRect, CV\_RGB(255,0,0), 2);

}

// >>>>> Noise smoothing

cv::Mat blur;

cv::GaussianBlur(frame, blur, cv::Size(5, 5), 3.0, 3.0);

// <<<<< Noise smoothing

// >>>>> HSV conversion

cv::Mat frmHsv;

cv::cvtColor(blur, frmHsv, CV\_BGR2HSV);

// <<<<< HSV conversion

// >>>>> Color Thresholding

// Note: change parameters for different colors

cv::Mat rangeRes = cv::Mat::zeros(frame.size(), CV\_8UC1);

cv::inRange(frmHsv, cv::Scalar(MIN\_H\_BLUE / 2, 100, 80),

cv::Scalar(MAX\_H\_BLUE / 2, 255, 255), rangeRes);

// <<<<< Color Thresholding

// >>>>> Improving the result

cv::erode(rangeRes, rangeRes, cv::Mat(), cv::Point(-1, -1), 2);

cv::dilate(rangeRes, rangeRes, cv::Mat(), cv::Point(-1, -1), 2);

// <<<<< Improving the result

// Thresholding viewing

cv::imshow("Threshold", rangeRes);

// >>>>> Contours detection

vector<vector<cv::Point> > contours;

cv::findContours(rangeRes, contours, CV\_RETR\_EXTERNAL,

CV\_CHAIN\_APPROX\_NONE);

// <<<<< Contours detection

// >>>>> Filtering

vector<vector<cv::Point> > balls;

vector<cv::Rect> ballsBox;

for (size\_t i = 0; i < contours.size(); i++)

{

cv::Rect bBox;

bBox = cv::boundingRect(contours[i]);

float ratio = (float) bBox.width / (float) bBox.height;

if (ratio > 1.0f)

ratio = 1.0f / ratio;

// Searching for a bBox almost square

if (ratio > 0.75 && bBox.area() >= 400)

{

balls.push\_back(contours[i]);

ballsBox.push\_back(bBox);

}

}

// <<<<< Filtering

cout << "Balls found:" << ballsBox.size() << endl;

// >>>>> Detection result

for (size\_t i = 0; i < balls.size(); i++)

{

cv::drawContours(res, balls, i, CV\_RGB(20,150,20), 1);

cv::rectangle(res, ballsBox[i], CV\_RGB(0,255,0), 2);

cv::Point center;

center.x = ballsBox[i].x + ballsBox[i].width / 2;

center.y = ballsBox[i].y + ballsBox[i].height / 2;

cv::circle(res, center, 2, CV\_RGB(20,150,20), -1);

stringstream sstr;

sstr << "(" << center.x << "," << center.y << ")";

cv::putText(res, sstr.str(),

cv::Point(center.x + 3, center.y - 3),

cv::FONT\_HERSHEY\_SIMPLEX, 0.5, CV\_RGB(20,150,20), 2);

}

// <<<<< Detection result

// >>>>> Kalman Update

if (balls.size() == 0)

{

notFoundCount++;

cout << "notFoundCount:" << notFoundCount << endl;

if( notFoundCount >= 100 )

{

found = false;

}

/\*else

kf.statePost = state;\*/

}

else

{

notFoundCount = 0;

meas.at<float>(0) = ballsBox[0].x + ballsBox[0].width / 2;

meas.at<float>(1) = ballsBox[0].y + ballsBox[0].height / 2;

meas.at<float>(2) = (float)ballsBox[0].width;

meas.at<float>(3) = (float)ballsBox[0].height;

if (!found) // First detection!

{

// >>>> Initialization

kf.errorCovPre.at<float>(0) = 1; // px

kf.errorCovPre.at<float>(7) = 1; // px

kf.errorCovPre.at<float>(14) = 1;

kf.errorCovPre.at<float>(21) = 1;

kf.errorCovPre.at<float>(28) = 1; // px

kf.errorCovPre.at<float>(35) = 1; // px

state.at<float>(0) = meas.at<float>(0);

state.at<float>(1) = meas.at<float>(1);

state.at<float>(2) = 0;

state.at<float>(3) = 0;

state.at<float>(4) = meas.at<float>(2);

state.at<float>(5) = meas.at<float>(3);

// <<<< Initialization

kf.statePost = state;

found = true;

}

else

kf.correct(meas); // Kalman Correction

cout << "Measure matrix:" << endl << meas << endl;

}

// <<<<< Kalman Update

// Final result

cv::imshow("Tracking", res);

// User key

ch = cv::waitKey(1);

}

// <<<<< Main loop

return EXIT\_SUCCESS;

}