# GitHub Repository Link

https://github.coventry.ac.uk/5062CEM/STUDENTID\_IPA

# recogniser.h

#ifndef OPENCV\_OBJECTDETECTION\_RECOGNISER\_H
#define OPENCV\_OBJECTDETECTION\_RECOGNISER\_H

#include <iostream>
#include <time.h>
#include "opencv2/opencv.hpp"
#include "opencv2/core/core.hpp"
#include "opencv2/xfeatures2d/nonfree.hpp"
#include "opencv2/features2d/features2d.hpp"

using namespace std;
using namespace cv;
using namespace xfeatures2d;

class Detector {
public:

 string get\_type(int type) {
 string r;

 uchar depth = type & CV\_MAT\_DEPTH\_MASK;
 uchar chans = 1 + (type >> CV\_CN\_SHIFT);

 switch ( depth ) {
 case CV\_8U: r = "8U"; break;
 case CV\_8S: r = "8S"; break;
 case CV\_16U: r = "16U"; break;
 case CV\_16S: r = "16S"; break;
 case CV\_32S: r = "32S"; break;
 case CV\_32F: r = "32F"; break;
 case CV\_64F: r = "64F"; break;
 default: r = "User"; break;
 }

 r += "C";
 r += (chans+'0');

 return r;
 }

 /\*
 \* This method is used to set the feature detector of for the object detection class. There
 \* are two types of detectors that we use in this class: BRISK and SURF.
 \*/
 void set\_detector(const string& detector) {
 if(detector == "BRISK") {
 feature\_detector = BRISK::create(30, 3, 1.0f);
 }
 else if(detector == "SURF") {
 feature\_detector = SURF::create(100, 4, 3, false, false);
 }
 }

 /\*
 \* This function is used to detect the key areas of interest, otherwise known as keypoints, form an
 \* image or a frame grabbed from the camera or video.
 \*/
 vector<KeyPoint> get\_keypoints(const Mat& img) {
 vector<KeyPoint> keypoints;
 feature\_detector->detect(img, keypoints);
 return keypoints;
 }

 /\*
 \* This function is used to extract the descriptor from an image, or frame grabbed from the camera
 \* or video. It will use the key areas of interest, otherwise known as keypoints, to form this
 \* descriptor.
 \*/
 Mat get\_descriptor(const Mat& img, vector<KeyPoint> keypoints) {
 Mat descriptor;
 feature\_detector->compute(img, keypoints, descriptor);
 return descriptor;
 }

 /\*
 \* This function will match the descriptors that were extracted from the image, an a frame grabbed
 \* from the camera or video. Depending upon the type of feature detector used, there may be two types
 \* of descriptors extracted. This means that their may be two types of 'matchers', BruteForce for those
 \* that are an 8-bit integer, or Flann for those that are 32-bit floats.
 \* The matching process uses the k-nearest-neighbour algorithm, with k set as 2. These initial matches
 \* are then refined using a ratio test (set to 0.7) to filter out the 'good matches'. These matches are
 \* then used to determine whether the object has been found in the image or frame grabbed from the camera
 \* or video.
 \*/
 vector<DMatch> match(const Mat& d1, const Mat& d2) {
 std::vector< std::vector<DMatch> > knn\_matches;

 if(get\_type(d1.type()) == "32FC1") {
 matcher = DescriptorMatcher::create(DescriptorMatcher::FLANNBASED);
 }
 else if(get\_type(d1.type()) == "8UC1") {
 matcher = DescriptorMatcher::create(DescriptorMatcher::BRUTEFORCE);
 }

 matcher->knnMatch(d1, d2, knn\_matches, 2);
 const float ratio\_thresh = 0.7f;
 std::vector<DMatch> good\_matches;
 for (size\_t i = 0; i < knn\_matches.size(); i++)
 {
 if (knn\_matches[i][0].distance < ratio\_thresh \* knn\_matches[i][1].distance)
 {
 good\_matches.push\_back(knn\_matches[i][0]);
 }
 }
 return good\_matches;
 }

 /\*
 \* This function will detect the object from the frame grabbed from the camera or the video.
 \*/
 vector<Point2f> detect\_object(const Mat& \_training\_image, vector<KeyPoint> \_training\_keypoints,
 vector<KeyPoint> \_frame\_keypoints, const vector<DMatch>& \_filtered\_matches) {

 vector<Point2f> tmp\_object;
 vector<Point2f> tmp\_frame;

 for(auto m : \_filtered\_matches) {
 tmp\_object.push\_back(\_training\_keypoints[m.queryIdx].pt);
 tmp\_frame.push\_back(\_frame\_keypoints[m.trainIdx].pt);
 }

 Mat h = findHomography(tmp\_object, tmp\_frame, RANSAC);

 if(h.empty()) {
 CV\_Assert("H Is empty");
 return {Point2f(0, 0), Point2f(0, 0), Point2f(0, 0), Point2f(0, 0)};
 }

 int height = \_training\_image.rows;
 int width = \_training\_image.cols;

 vector<Point2f> object\_corners(4);
 vector<Point2f> frame\_corners(4);

 object\_corners[0] = Point2f(0, 0);
 object\_corners[1] = Point2f(width, 0);
 object\_corners[2] = Point2f(width, height);
 object\_corners[3] = Point2f(0, height);

 perspectiveTransform(object\_corners, frame\_corners, h);

 return frame\_corners;
 }

 /\*
 \* This function will calculate the FPS in which the camera, or video file is running at. This is purely
 \* for debugging purposes only; and ensures that I can see my object detector is running in a real-time
 \* constraint. For example, if my camera feed is 30FPS, then I expect it to continue running at 30FPS.
 \*/
 int calculate\_fps(int \_frame\_number, time\_t \_time) {
 return (int) \_frame\_number / difftime(time(NULL), \_time);
 }

private:
 Ptr<FeatureDetector> feature\_detector;
 Ptr<BRISK> brisk = BRISK::create(30, 3, 1.0f);
 Ptr<DescriptorMatcher> matcher;
 Ptr<SURF> surf = SURF::create(100, 4, 3, false, false);
};

#endif //OPENCV\_OBJECTDETECTION\_RECOGNISER\_H

# main.cpp

#include "recogniser.h"

int main() {
 Detector object\_detector;
 const char \*pipeline = "autovideosrc ! videoconvert ! video/x-raw,width=640,height=480,framerate=30/1 ! queue ! appsink";
 VideoCapture cap(pipeline, CAP\_GSTREAMER);

// VideoCapture cap(0);

 // Create an instance of the class using the SURF feature detector
 object\_detector.set\_detector("BRISK");

 // Load the training image that we want to use to detect
 Mat training\_img = imread("img.png", IMREAD\_COLOR);
 // Grab some key areas of interest from the training image
 vector<KeyPoint> training\_keypoints = object\_detector.get\_keypoints(training\_img);
 // Extract a descriptor from the training image using the key areas of interest
 Mat training\_descriptor = object\_detector.get\_descriptor(training\_img, training\_keypoints);

 // Set the video capture method to use our in-built webcam and check whether we have opened the camera or not.
 if(!cap.isOpened()) {
 CV\_Assert("Opening the Camera Failed");
 return 0;
 }

 // Set a frame number
 int frame\_number = 0;

 // Get the start time
 time\_t start\_time;
 time(&start\_time);

 // Use an infinite loop to grab frames from the webcam, we can break this later on
 for(;;) {
 // Read a frame from the camera, and a return value on whether it is grabbing a frame
 Mat frame;
 cap >> frame;
 // If the frame is none (i.e. empty) then we can throw an error
 if(frame.empty()) {
 CV\_Assert("Error reading a frame");
 break;
 }

 // Find key areas of interest from the frame of the camera
 vector<KeyPoint> keypoints = object\_detector.get\_keypoints(frame);
 // Generate a descriptor from the key areas of interest
 Mat descriptor = object\_detector.get\_descriptor(frame, keypoints);

 // Perform a match between the descriptor of the training image and the frame to determine if
 // the object can be found.
 vector<DMatch> matches = object\_detector.match(training\_descriptor, descriptor);

 vector<Point> points;
 try {
 vector<Point2f> boundaries = object\_detector.detect\_object(training\_img, training\_keypoints, keypoints, matches);
 for(auto & b : boundaries) {
 points.push\_back(b);
 }
 if(!points.empty()) {
 polylines(frame, points, true, Scalar(0, 255, 0), 2, LINE\_AA, 0);
 }
 } catch (Exception e) {
// cout << e.what() << endl;
 }

 // Increments the frame number
 frame\_number += 1;

 // Put the FPS in the top-left corner of the image
 putText(frame, to\_string(object\_detector.calculate\_fps(frame\_number, start\_time)), Point(0, 15),
 FONT\_HERSHEY\_COMPLEX, 0.5, Scalar(255, 255, 255), 1, LINE\_AA);

 // Displays just the camera, with a bounding box around the detected image.
 imshow("Window", frame);

 // Sets a wait key for one second, and listens for ESC key to break the loop
 if(waitKey(1) == 27) {
 break;
 }
 }

 // Releases the camera when the while loop has ended
 cap.release();
 // Destroys any windows that were created
 destroyAllWindows();

 return 0;
}

# CMakeLists.txt

cmake\_minimum\_required(VERSION 3.25)
project(OpenCV\_ObjectDetection)

set(CMAKE\_CXX\_STANDARD 17)

# Add OpenCV
set(OpenCV\_DIR "/opt/opencv/build")

find\_package(OpenCV REQUIRED)
include\_directories(${OpenCV\_INCLUDE\_DIRS})

add\_executable(OpenCV\_ObjectDetection main.cpp recogniser.h)

# Add OpenCV Libraries
set(OpenCV\_LIBS opencv\_core opencv\_imgproc opencv\_highgui opencv\_imgcodecs opencv\_xfeatures2d opencv\_features2d opencv\_calib3d)
# Link OpenCV Libraries
target\_link\_libraries(OpenCV\_ObjectDetection ${OpenCV\_LIBS})