GitHub Repository Link

https://github.coventry.ac.uk/5062CEM/STUDENTID_IPA

recogniser.h

```cpp
#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

```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})
```