```
#include <stdlib.h>
#include <stdio.h>
#include "opencv2/opencv.hpp"
#include <vector>
#include <cmath>
#include <pthread.h>
#include <unistd.h>

#include "LineDesc.h"
#include "PotentialBug.h"
#include "tracker.h"
#include "analyzeFrame.h"

using namespace cv;
using namespace std;

#define sqr(a)  ((a) * (a))

int movingAverage;
Mat *im_out2;
double fps;
bool vflag = false;
char baseFileName[250];

char *ret_base_fname(char *arg, char * baseFileName) {
   int back, j1;

   back = (int) strlen(arg);

   for (j1 = back - 1; j1 >= 0; j1--) {
      if (arg[j1] == '.') {
         strncpy(baseFileName, arg, j1);
         baseFileName[j1] = 0;
         return baseFileName;
      }
   }
   return strcpy(baseFileName, arg);
}

void MouseCallBackFunc(int event, int x, int y, int flags, void* ptr) {
   char *trackname = NULL;
   if (event == EVENT_LBUTTONDOWN) {
      unsigned int trackNum;
      char stringBuffer[100];
      char fileName[250];
      tracker *p = (tracker *) ptr;
      Point2f point;
```

```
        if (vflag) {
            point.x = x;
            point.y = y;
        } else {
            point.x = x;
            point.y = y;
        }
        trackNum = p->findNearestTrackToPoint(point);
        sprintf(fileName, "%s-trk-%d.txt", baseFileName, trackNum);
        p->writeATrack(trackNum, movingAverage, 30, fileName, NULL);
        sprintf(stringBuffer, " Writing track %d", trackNum);
        putText(*im_out2, stringBuffer, Point2f(x, y), CV_FONT_HERSHEY_SIMPLEX, 0.5, cv::Scalar(255), 1,
8, false);

    } else if (event == EVENT_RBUTTONDOWN) {
    } else if (event == EVENT_MBUTTONDOWN) {
    } else if (event == EVENT_MOUSEMOVE) {
    }
}

bool pointRectangleTest(Rect interiorLimit, Point2f aPoint) {
    if (aPoint.x >= interiorLimit.x && aPoint.x <= (interiorLimit.x + interiorLimit.width)) {
        if (aPoint.y >= interiorLimit.y && aPoint.y <= (interiorLimit.y + interiorLimit.height)) {
            return false;
        }
    }
    return false;

}

int findLargestKeypoint(const vector<KeyPoint>& keypoints) {
    float maxSize = 0.0;
    int indx = 0;
    for (int i = 0; i < keypoints.size(); i++) {
        if (keypoints[i].size > maxSize) {
            maxSize = keypoints[i].size;
            indx = i;
        }
    }
    return indx;
}

int findLargestContour(const vector<std::vector<cv::Point> > contours) {
    //double maxContour = 200000.0;
    //double minContour = 25.0;
    int largest_area = 0;
    int largest_contour_index = -1;
    for (int i = 0; i < contours.size(); i++) // iterate through each contour.
```

```
    {
        double a = contourArea(contours[i], false); //  Find the area of contour
        //if (a < minContour || a > maxContour)
        //    continue;
        if (a > largest_area) {
            largest_area = a;
            largest_contour_index = i; //Store the index of largest contour
        }
    }
    return largest_contour_index;
}

float euclidDistance(Point2f p1, Point2f p2) {
    float sum;
    sum = (p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) *(p1.y - p2.y);
    return (sqrt(sum));
}

Point2f averageVelocity(Point2f p1) {
    int buffSize = 100;
    static Point2f *Carray = new Point2f[buffSize];
    static int current = 0;
    static int total = 0;
    Point2f sum0, sum1, newVec;
    static Point2f prevPoint;

    if ((total > 0) && euclidDistance(prevPoint, p1) > 10) {
        prevPoint = p1; //allow us to slide over a glitch
        return p1;
    }
    if (total < buffSize)
        total++;
    Carray[current] = p1;
    current++;
    if (current >= buffSize)
        current -= buffSize;
    if (total < 20) { //warming up buffer
        return p1;
    }
    for (int j = current - total; j < current - total / 2 - 1; j++) {
        int indx = j;
        while (indx >= buffSize)
            indx -= buffSize;
        while (indx < 0)
            indx += buffSize;
        sum0.x += Carray[indx].x;
        sum0.y += Carray[indx].y;
    }
```

```
    sum0.x /= (total - total / 2 - 1);
    sum0.y /= (total - total / 2 - 1);

    for (int j = current - total / 2; j < current; j++) {
        int indx = j;
        while (indx >= buffSize)
            indx -= buffSize;
        while (indx < 0)
            indx += buffSize;
        sum1.x += Carray[indx].x;
        sum1.y += Carray[indx].y;
    }
    sum1.x /= (total / 2);
    sum1.y /= (total / 2);

    newVec.y = p1.y + (sum1.y - sum0.y) * 5;
    newVec.x = p1.x + (sum1.x - sum0.x) * 5;

    return newVec;
}

int randomInt(int min, int max) {
    return (min + (rand() % (int) (max - min + 1)));
}

int main(int argc, char *argv[]) {
    Mat img, img2, imgBackgroundModel, imgUnmapped, im_out, subtracted_frame_shrunk;
    Mat fore;
    Mat back;
    BackgroundSubtractorMOG2 bgMain;
    Mat foreMain, backMain;
    int skipFrames = 1;
    int initialSkip = 1;
    unsigned int frameCount = 0;
    int nCores = 1;
    double velocityVectorMult = 10.0;
    movingAverage = 60;
    Point2f prevCenter;
    vector<KeyPoint> keypointsBlob;
    char jpegFileName[250];
    char outputFileName[250];
    char subFileName[250];
    char reFileName[250];
    bool reWrite = false;
    int64 e1, e2;
    double execTime;
    bool saveBackgroundModel = false;
```

```
char *fname = NULL;
int index;
int c;
opterr = 0;
while ((c = getopt(argc, argv, "svrj:f:i:m:")) != -1)
   switch (c) {
      case 'r':
         reWrite = true;
         break;
      case 'v':
         vflag = true;
         break;
      case 'j':
         nCores = atoi(optarg);
         break;
      case 'i':
         initialSkip = atoi(optarg);
         break;
      case 'm':
         movingAverage = atoi(optarg);
         break;
      case 'f':
         fname = optarg;
         break;
      case 's':
         saveBackgroundModel = true;
         break;
      case '?':
         if (optopt == 'c')
            fprintf(stderr, "Option -%c requires an argument.\n", optopt);
         else if (isprint(optopt))
            fprintf(stderr, "Unknown option `-%c'.\n", optopt);
         else
            fprintf(stderr,
               "Unknown option character `\\x%x'.\n",
               optopt);
         return 1;
      default:
         abort();
   }
Mat intrinsic_matrix = Mat::eye(3, 3, CV_64F);
Mat distortion_coeffs = Mat::zeros(8, 1, CV_64F);
FileStorage fs;
fs.open("JVCDistortion.xml", FileStorage::READ);
if (!fs.isOpened()) {
   cerr << "Failed to open " << "JVCDistortion.xml" << endl;
   return 1;
```

```
}
fs["Intrinsics"] >> intrinsic_matrix;
fs["DistortionCoefs"] >> distortion_coeffs;
setNumThreads(nCores);
analyzeFrame frameThreads[nCores];
ret_base_fname(fname, baseFileName);

VideoCapture capture(fname);
if (!capture.isOpened()) // if not success, exit program
{
   puts("Cannot open the video file");
   return -1;
}

bool bSuccess = capture.read(img); // read a new frame from video
if (!bSuccess) //if not success, break loop
{
   puts("Cannot read the frame from video file");
   exit(-1);
}

Mat view, rview, mapx, mapy;
initUndistortRectifyMap(intrinsic_matrix, distortion_coeffs, Mat(),
      getOptimalNewCameraMatrix(intrinsic_matrix, distortion_coeffs, img.size(), 1, img.size(), 0),
      img.size(), CV_16SC2, mapx, mapy);

// create a window
namedWindow("mainWin2", CV_WINDOW_AUTOSIZE);
moveWindow("mainWin2", 950, 50);

namedWindow("CurrentView", CV_WINDOW_NORMAL | CV_WINDOW_KEEPRATIO);
moveWindow("CurrentView", 50, 50);

namedWindow("BlobView", CV_WINDOW_AUTOSIZE);
moveWindow("BlobView", 50, 500);


e1 = getTickCount();
frameCount = 0;
for (;;) {
   bSuccess = capture.read(imgUnmapped);
   if (!bSuccess) break;
   frameCount++;
   if (frameCount > 3) break;
}
Mat imgblur(imgUnmapped.size(), CV_8UC3);

if (REMAPPING)
```

```
    remap(imgUnmapped, img, mapx, mapy, INTER_LINEAR);
  else
    img = imgUnmapped;

  if (GAUSSIAN) {
    GaussianBlur(img, imgblur, Size(GAUSSIANBLUR, GAUSSIANBLUR), 0, 0); //note: remember to match
this with call in analyzeframe
    img = imgblur.clone();
  }


  resizeWindow("CurrentView", img.cols / 2, img.rows / 2);


  for (;;) {
    bSuccess = capture.read(imgBackgroundModel);
    if (!bSuccess) break;
    frameCount++;
    if (frameCount > initialSkip) break;
  }

  if (JORGE)
    img = imgBackgroundModel;

  if (HYBRIDADAPTIVEBACKGROUNDMODELING) {
    //This are explained in analyzeframe.  Note that changes there are not reflected here.
    bgMain.set("nmixtures", 3);
    bgMain.set("detectShadows", true);
    bgMain.set("backgroundRatio", 0.99);
    bgMain.set("fCT", 0.3); //was 0.05
    bgMain.set("history", 200);
    sprintf(jpegFileName, "%sBackgroundModel.jpg", baseFileName);
    VideoCapture capture2(jpegFileName);
    if (capture2.isOpened()) // if not success, exit program
    {
      bSuccess = capture2.read(img);

    } else bSuccess = false;
    if (!bSuccess) {
      puts("Building background model");
      for (int j = 0; j < HYBRIDMAXINITIALRUN; j++) {
        bSuccess = capture.read(imgBackgroundModel);
        if (!bSuccess) break;
        //tmp = imgBackgroundModel.clone();
        bgMain.operator()(imgBackgroundModel, foreMain); //0.005 seemed good
        bgMain.getBackgroundImage(backMain);
        resize(backMain, subtracted_frame_shrunk, Size(img.cols / 2, img.rows / 2), 0, 0, INTER_AREA);
        imshow("mainWin2", subtracted_frame_shrunk);
```

```
            resize(imgBackgroundModel, im_out, Size(imgBackgroundModel.cols / 2,
imgBackgroundModel.rows / 2), 0, 0, INTER_AREA);
            imshow("CurrentView", im_out);
            if (cvWaitKey(1) >= 0) break;
          }
          bgMain.getBackgroundImage(img);
          if (GAUSSIAN) {
            GaussianBlur(img, imgblur, Size(GAUSSIANBLUR, GAUSSIANBLUR), 0, 0); //note: remember to
match this with call in analyzeframe
            img = imgblur.clone();
          }
          sprintf(jpegFileName, "%sBackgroundModel.jpg", baseFileName);
          imwrite(jpegFileName, img);
        }
    }


    capture.set(CV_CAP_PROP_POS_FRAMES, initialSkip); //Rewind the video
    frameCount = initialSkip;



    Mat img3(img.size(), CV_8UC3);
    Mat img4(img.size(), CV_8UC3);
    img4.setTo(Scalar(0, 0, 0));
    tracker trackerList(&img4);

    sprintf(outputFileName, "%s-Analysis.avi", baseFileName);
    sprintf(subFileName, "%s-Subtracted.avi", baseFileName);
    sprintf(reFileName, "re-%s.avi", baseFileName);

    fps = capture.get(CV_CAP_PROP_FPS);
    VideoWriter writer(outputFileName, CV_FOURCC('M', 'P', '4', '2'), capture.get(CV_CAP_PROP_FPS),
img.size(), true); //initialize the VideoWriter object
    VideoWriter subWriter(subFileName, CV_FOURCC('M', 'P', '4', '2'), capture.get(CV_CAP_PROP_FPS),
img.size(), false); //initialize the VideoWriter object
    VideoWriter reWriter(reFileName, CV_FOURCC('M', 'P', '4', '2'), capture.get(CV_CAP_PROP_FPS),
img.size(), true); //initialize the VideoWriter object

    if (ADAPTIVEBACKGROUNDMODELING) //We need to process images 1 by 1
      nCores = 1;

    for (int j = 0; j < nCores; j++) {
      bSuccess = capture.read(img2);
      if (!bSuccess) break;
      frameCount++;
      if ((frameCount % skipFrames) != 0) //Skips skipFrames frame
        continue;
```

```
      frameThreads[j].initializeMats(img.size());
      frameThreads[j].setImg(img);
      frameThreads[j].setImg2(img2);
      frameThreads[j].setParameters();
      frameThreads[j].setFramecount(frameCount);
      frameThreads[j].setMapx(mapx);
      frameThreads[j].setMapy(mapy);
      frameThreads[j].setFore(fore);
      frameThreads[j].setBack(back);
      frameThreads[j].StartInternalThread();
   }

   int roundRobinIndex = 0;
   Mat im_with_keypoints;
   for (;;) {
      frameThreads[roundRobinIndex].WaitForInternalThreadToExit();

      keypointsBlob = frameThreads[roundRobinIndex].GetCurrentBlobs();

      // Draw detected blobs as red circles.
      // DrawMatchesFlags::DRAW_RICH_KEYPOINTS flag ensures the size of the circle corresponds to the
size of blob
      drawKeypoints(img, keypointsBlob, im_with_keypoints, Scalar(0, 0, 255),
DrawMatchesFlags::DRAW_RICH_KEYPOINTS);
      resize(im_with_keypoints, im_out, Size(imgBackgroundModel.cols / 2, imgBackgroundModel.rows /
2), 0, 0, INTER_AREA);
      imshow("BlobView", im_out);


      bSuccess = capture.read(img2);
      if (!bSuccess) break;
      frameCount++;
      if ((frameCount % skipFrames) != 0) //Skip skipFrames
         continue;
      if ((frameCount % 500) == 0) {
         printf(".");
         fflush(stdout);
      }
      img3 = frameThreads[roundRobinIndex].getImg2Mapped().clone();

      char stringBuffer[100];
      sprintf(stringBuffer, " Frame %u", frameCount);
      putText(img3, stringBuffer, Point2f(100, 100), CV_FONT_HERSHEY_SIMPLEX, 1.0, cv::Scalar(0, 0,
255), 1, 8, false);

      Point2f center;
      std::list<PotentialBug>::iterator it;
```

```
    if (keypointsBlob.size() > 0) {
       for (int j1 = 0; j1 < keypointsBlob.size(); j1++) {
          center.x = keypointsBlob[j1].pt.x;
          center.y = keypointsBlob[j1].pt.y;
          circle(img3, center, 3, Scalar(0, 0, 255), 2, 8, 0);
          PotentialBug *pb = new PotentialBug(center, frameCount);
          trackerList.evaluatePoint(pb, frameCount);
       }
       trackerList.cleanupPotentialBugs(frameCount);
       prevCenter = center;
    }


    if (vflag || (frameCount % 1 == 0)) {
       //Now we overlay non-black point from img4 over img3 (the color image we see)
       uint8_t* pixelPtr = (uint8_t*) img3.data;
       uint8_t* img4Ptr = (uint8_t*) img4.data;
       int cn = img3.channels();
       unsigned int tmp;
       Scalar_<uint8_t> bgrPixel, img4Pixel;
       for (unsigned int i = 0; i < img4.cols; i++) {
          for (unsigned int u = 0; u < img4.rows; u++) {
             tmp = u * img4.cols * cn + i * cn;
             img4Pixel.val[0] = img4Ptr[tmp + 0]; // B
             img4Pixel.val[1] = img4Ptr[tmp + 1]; // G
             img4Pixel.val[2] = img4Ptr[tmp + 2]; // R
             if ((img4Pixel.val[0] != 0) || (img4Pixel.val[1] != 0) || (img4Pixel.val[2] != 0)) {
                tmp = u * img3.cols * cn + i * cn;
                pixelPtr[tmp + 0] = img4Pixel.val[0];
                pixelPtr[tmp + 1] = img4Pixel.val[1];
                pixelPtr[tmp + 2] = img4Pixel.val[2];
             }
          }
       }
    }

    if (vflag) {
       resize(frameThreads[roundRobinIndex].getSubtracted_frame(), subtracted_frame_shrunk,
Size(img.cols / 2, img.rows / 2), 0, 0, INTER_AREA);
       imshow("mainWin2", subtracted_frame_shrunk);
       resize(img3, im_out, Size(img4.cols / 2, img4.rows / 2), 0, 0, INTER_AREA);
       imshow("CurrentView", im_out);
    }

    if (frameCount % 1 == 0) {
       writer.write(img3);
       subWriter.write(frameThreads[roundRobinIndex].getSubtracted_frame());
       if (reWrite)
```

```
            reWriter.write(img2);
        }

        if (cvWaitKey(1 * SLOWMOTIONFACTOR) >= 0) break;

        frameThreads[roundRobinIndex].setImg2(img2);
        frameThreads[roundRobinIndex].setFramecount(frameCount);
        frameThreads[roundRobinIndex].StartInternalThread();
        roundRobinIndex++;
        roundRobinIndex %= nCores;
    }

    printf("\nNumber tracks %u\n", trackerList.getNumTracks());
    for (int j = 0; j < trackerList.getNumTracks(); j++) {
        printf("track # %4u, size %4u , initial %5u, final %5u, distance %7.4f, degrees %7.4f\n", j,
trackerList.getTrackSize(j), trackerList.getInitialTrackFrame(j), trackerList.getFinalTrackFrame(j),
            trackerList.getTrackAverageDistance(j, movingAverage),
trackerList.getTrackAverageDegreesTurned(j, movingAverage));
    }

    if (!vflag) { //We didn't do this without video flag set
        //Now we overlay non-black point from img4 over img3 (the color image we see)
        uint8_t* pixelPtr = (uint8_t*) img3.data;
        uint8_t* img4Ptr = (uint8_t*) img4.data;
        int cn = img3.channels();
        unsigned int tmp;
        Scalar_<uint8_t> bgrPixel, img4Pixel;
        for (unsigned int i = 0; i < img4.cols; i++) {
            for (unsigned int u = 0; u < img4.rows; u++) {
                tmp = u * img4.cols * cn + i * cn;
                img4Pixel.val[0] = img4Ptr[tmp + 0]; // B
                img4Pixel.val[1] = img4Ptr[tmp + 1]; // G
                img4Pixel.val[2] = img4Ptr[tmp + 2]; // R
                if ((img4Pixel.val[0] != 0) || (img4Pixel.val[1] != 0) || (img4Pixel.val[2] != 0)) {
                    tmp = u * img3.cols * cn + i * cn;
                    pixelPtr[tmp + 0] = img4Pixel.val[0];
                    pixelPtr[tmp + 1] = img4Pixel.val[1];
                    pixelPtr[tmp + 2] = img4Pixel.val[2];
                }
            }
        }
    }

    writer.release(); //Doing this here allows us to access these to help analyse the tracks
    subWriter.release();
    reWriter.release();
    e2 = getTickCount();
    execTime = (e2 - e1) / getTickFrequency();
```

```
    printf("Execution time %7.3f \n", execTime);

    //set the callback function for any mouse event
    setMouseCallback("CurrentView", MouseCallBackFunc, &trackerList);

    for (;;) {
        if (vflag)
            im_out2 = &img3;
        else
            im_out2 = &img3;
        imshow("CurrentView", *im_out2);
        if (waitKey(1) == 27) break; //This allows us to identify and write tracks
    }


    sprintf(jpegFileName, "%s.jpg", baseFileName);
    imwrite(jpegFileName, img3);
    destroyAllWindows();
    return 0;
}
```