Report on Camera Calibration

Elim Yi Lam Kwan (ylk25)

1 Camera Calibration

1.1 Background Theory of Calibration

Lens are responsible for capturing lights and bend it towards the internal digital sensor.

Moreover, the refraction of light will induce anomalies on the reproduced image. In this project, we considered two types of lens distortions: radial and tangential. Distortions are severe in pinhole camera, which is the camera model assumed by OpenCV. Radial distortions are commonly caused by wide angle lens, while tangential distortion occurs when the lens is not parallel to the imaging plane. They can be approximated as (left: radial; right: tangential):

$$\begin{bmatrix} x_{distorted} \\ y_{distorted} \end{bmatrix} = (1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \begin{bmatrix} x \\ y \end{bmatrix} \quad \begin{bmatrix} x_{distorted} \\ y_{distorted} \end{bmatrix} = \begin{bmatrix} 2p_1 xy + p_2(r^2 + 2x^2) \\ p_1(r^2 + 2y^2) + 2p_2 xy \end{bmatrix} + \begin{bmatrix} x \\ y \end{bmatrix}$$

And the distortion coefficients are $(k_1, k_2, p_1, p_2, k_3)$. Moreover, to achieve perspective transformation, we can project the points obtained from the resultant camera image onto a image plane. The camera matrix can be described mathematically as:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{1}$$

where (X, Y, Z), (u, v), (c_x, c_y), (f_x, f_y), r_{mn} represents the 3D coordinate of the point from the image (mm), the 2D coordinate of the projected point (px), the principal point of the image plane(px), the focal lengths (px), and (the rotation matrix with the translation vector) of the camera respectively. The 1st and 2nd matrices are the camera's intrinsic matrix K and extrinsic matrix $[r1, r2, r3, t]$.

By taking snapshots of a predefined pattern (e.g. a box) at various angles as shown in Figure 1, we can identify the locations of specific corners within the patterns from the images, as well as the expected locations of these points with the given physical measurements of the box (width: 106mm; height: 105mm). A system of equations can then be formed. To simplify the computations, the printed 2D image will be fixed on a surface, i.e., $Z = 0$ and (r_{13}, r_{23}, r_{33}) can be eliminated from the matrix.

1.2 Calibration Methodology

The camera model used in the following experiment was Dericam W3 Webcam with a resolution of 2MP, 110° Field of View (FOV), 3.6mm lens size. Lightings were around 800 lux. More importantly, the printed image used for calibration was firmly attached to a hard surface (a ring binder), ensuring Z equals to 0 in Equation 1. More details of the setup can be seen in Figure 2. Our algorithm was implemented with OpenCv4.1 in C++. It was able to consistently identify the 4 corner points shown in Figure 1 in order. *(Point 0: the corner with the triangular bend; Point 1: corner opposite to Point 0; Point 2: the line segment between Point 0 and 2 will always consist of the concave point)*

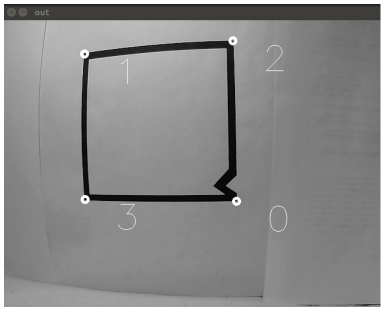

Figure 1: Corners used for Calibrations

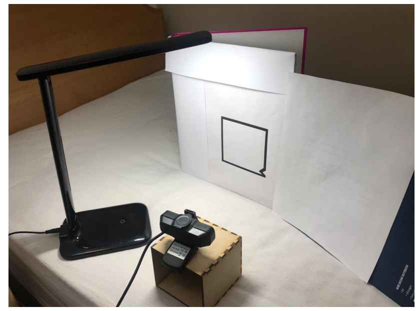

Figure 2: Experiment Setup

During the experiment, a real-time video stream was provided to the program, once it located the corners, it verified with the user whether those are accurate data. In the end, points from 12 images were identified as valid and had used for calibration. In terms of the performance matrix, the RMS re-projection error was used, which indicates the accuracy of the found parameters.

1.3 Calibration Results

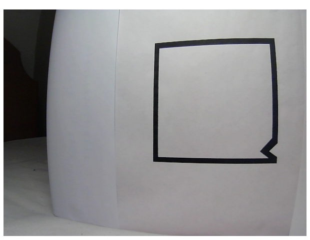

Figure 3: Before

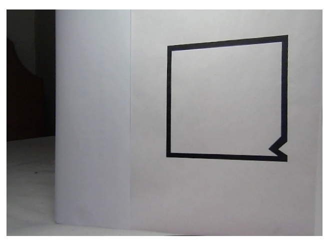

Figure 4: After

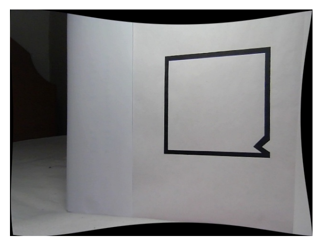

Figure 5: After (Uncropped)

Table 1: Obtained Camera Matrix and Distortion Coefficients

Camera Matrix					**Distortion Coefficients**				
f_x	f_y	c_x	c_y		k_1	k_2	p_1	p_2	k_3
538.513	536.397	358.637	224.381		-0.35315	0.00873	0.00256	-0.00026	0.15053

The program has attained a RMS re-projection error of 0.84. The results were visualised in Figure 4 and Table 1. Based on trigonometry, the focal length can be roughly estimated with $\alpha = 2\tan^{-1}(\frac{w}{2f})$, α being the horizontal FOV and w being the image width, which is 640 px in this experiment. Our calculated f_x is 491.089 px, which is within a reasonable range from the calibration results, where f_x was 538.513. Moreover, f_y as shown in Table 1 was slightly less than f_x, which also follows intuitions, as the vertical FOV is usually less than the horizontal one.

In terms of distortion corrections, we can observe that the severe ”fish eye” effect has been alleviated in Figure 4. The result is in cohesion with the relative large k_1 and k_3 obtained in Table 1, which are responsible for correcting radial distortion. Since tangential distortions was not visible in Figure 3, not much correction is required, thus, the obtained p_1 and p_2 was in the order of 10^{-3}.

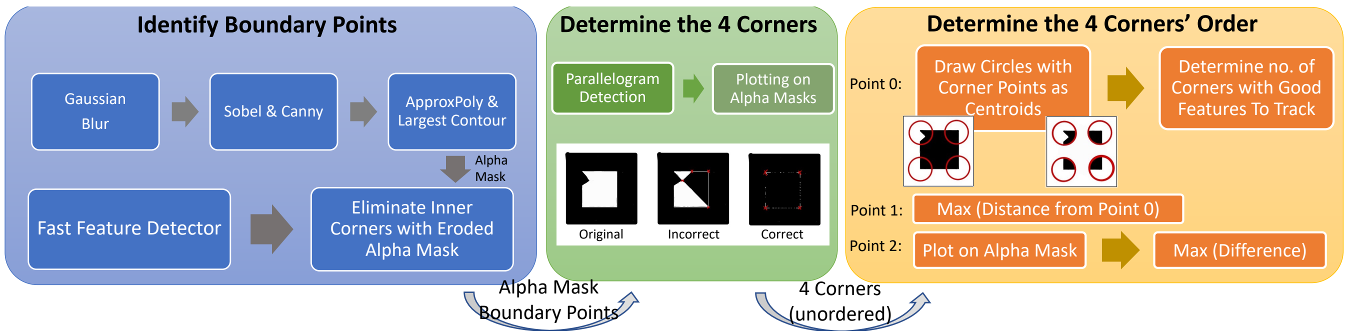

Figure 6: Image Processing Pipeline for Corners' Localisation

2 Image Processing

2.1 Image Processing Algorithms

The full pipeline for accurate corners' localisation was divided into 3 phases as shown in Figure 6: 1) Find boundary points; 2) Determine the 4 corners of the box from the list of potential points; 3) Map the 4 points to our defined Point 0 to 3 (in Figure 1). Main features of our pipeline include the mixed use of corners detection methodologies and the creative use of alpha mask.

Phase 1: On one hand, we first de-noised the image with a Gaussian Blur filter, then applied Sobol and Canny edge detectors to identify contours, and finally filled the largest contour to generate the alpha mask. On the other hand, to achieve corner detection that is robust to background variations and is applicable to our real-time data, FAST Feature Detector [1] was used. Given a test point I_p, FAST consider the intensity levels of 16-neighbouring pixels, whether they are brighter than the tested pixels by a certain threshold $I_p + t$. To accelerate the algorithm, the intensity levels comparison were conducted in a hierarchy manner. Although FAST Feature Detector is highly accurate in determining points lying on the edges, it generates a number of points that lies on the inner edge as well. To eliminate these extra points, we form an additional mask with the outer edges of the box plus additional margins through erosion and dilation, in which these margins must be less than the $\frac{1}{2}$ line weight. Applying this additional mask to the output of the FAST feature detector will give us a set of points lying at the outer edges, where they are referred to as boundary points.

Phase 2: We would then like to identify the four corner points of the box from the set of boundary points. To derive a shape detection solution that is robust to out-of-plane rotation, parallel lines detection was used instead of right angle ones. Moreover, to verify whether 4 points are the box corners given that they lie on the boundary, they can be plotted on the alpha mask with the $fillConvexPoly$ function, and only the right combination of points will generate an all-black image as shown in the green block of Figure 6.

Phase 3: Since the order of the identified corners are also important, we have derived different approach in identifying Point 0, 1, 2. For Point 0 (the corner with a triangular bend), we noticed that *Non-Point0* points will always contain 3 corners upon applying circular masks as shown in the orange block of Figure 6. Hence, another feature extractor - Good Features to Track [2] was applied. It is a modified version of the well known Harris Corner Detector, which works on the principle that if the image patch consists of a corner, moving in any direction will result in a significant change in intensity. Compare with Harris, the scoring function was changed from $R = \lambda_1\lambda_2 - k(\lambda_1 + \lambda_2)^2$

to $R = \min(\lambda_1 \lambda_2)$. For Point 1 (the opposite corner of Point 0), it will usually be the point that is furthest away from Point 0 in terms of Euclidean Distance. For Point 2 (concave line segment between itself and Point 0), we can also plot the line on the alpha mask. If the bend doesn't exist, the extra line will cause minimal changes to the mask. Vice Versa.

2.2 Limitations

Finally, since some of the assumptions made may not hold in certain cases (e.g., in phase 1, the largest contour may not be the box), we acknowledged the limitations of our image processing pipeline. Improving this component will be the key to overall system performance improvements. Nevertheless, the OpenCV data requirement[1] was still satisfied in our experiment. As a side note, we have also experimented with a Sony RXii camera, unfortunately, it was too difficult to collect sufficient valid data for useful analysis.

2.3 Perspective Correction

Below, we will attempt to explain the minimum amount of point required for camera calibration and how an improved solution can be obtained based on our knowledge on camera parameters. Referring to Equation 1, let the multiplication of the intrinsic camera matrix K and extrinsic camera matrix $[r1, r2, r3, t]$ be $H_{3\times3}$, each point on the pattern will then generate an equation:

$$[x_i, y_i, 1]^T = H_{3\times3}[X_i, Y_i, 1]^T [x_i, y_i, 1]^T \qquad = [h_1, h_2, h_3][X_i, Y_i, 1]^T$$

and h_1, h_2, h_3 are column vectors of H

2.4 Minimum Number of Points:

Since H has 8 Degree of Freedom (DOF) [2] and each observation has 2 elements (x, y), hence, a minimum of 4 points are required to determine H. In a special scenario, when the camera axes aligned with the world axes, the camera rotational matrix will become an identity matrix. Then, the DOF of H will be further reduced, and 3 points from the pattern will be sufficient to perform a reliable perspective correction.

2.5 An Improved Solution:

To compute K from H, the fact that r_1,r_2 are rotational vectors and K is invertable can be utilised. With $r_1^T r_2 = 0$ and $\mid r_1 \mid = \mid r_2 \mid = 1$:

$$h_1^T K^{-T} K^{-1} h_2 = 0 \qquad h_1^T K^{-T} K^{-1} h_1 = h_2^T K^{-T} K^{-1} h_2$$

We can then define $B = K^{-T} K^{-1}$. Since each planes provide us with two equations and B has 5-6 DOF, 3 different views of the plane is required to generate a good solution. This piece of knowledge has not been fully utilised in our experiment, hence, our re-projection error was quite high.

[1] OpenCV documentation stated that at least 10 images are required for reasonable calibration results

[2] H consists of 9 elements but it was estimated up to a scale

References

[1] Edward Rosten and Tom Drummond. “Machine Learning for High-Speed Corner Detection”. In: vol. 3951. July 2006. ISBN: 978-3-540-33832-1. DOI: 10.1007/11744023_34.

[2] Jianbo Shi and Tomasi. “Good features to track”. In: *1994 Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*. 1994, pp. 593–600. DOI: 10.1109/CVPR.1994.323794.

Appendix

Full code and testing results can be reviewed at https://github.com/elimkwan/Camera-Calibration. Below shows the snippet code of the *main.cpp* file, which contains the overall pipeline and the calibrations functions.

```
static double computeReprojectionErrors( const vector<vector<Point3f> >& objectPoints,
                                         const vector<vector<Point2f> >& imagePoints,
                                         const vector<Mat>& rvecs, const vector<Mat>& tvecs,
                                         const Mat& cameraMatrix , const Mat& distCoeffs,
                                         vector<float>& perViewErrors)
{
    vector<Point2f> imagePoints2;
    int i, totalPoints = 0;
    double totalErr = 0, err;
    perViewErrors.resize(objectPoints.size());

    for( i = 0; i < (int)objectPoints.size(); ++i )
    {
        cv::projectPoints( Mat(objectPoints[i]), rvecs[i], tvecs[i], cameraMatrix,
                       distCoeffs, imagePoints2);
        err = cv::norm(Mat(imagePoints[i]), Mat(imagePoints2), 4); //L2 is 4

        int n = (int)objectPoints[i].size();
        perViewErrors[i] = (float) std::sqrt(err*err/n);
        totalErr        += err*err;
        totalPoints     += n;
    }

    return std::sqrt(totalErr/totalPoints);
}


bool runCalibration(Size& imageSize, Mat& cameraMatrix, Mat& distCoeffs,
                    vector<vector<Point2f>> imagePoints, vector<Mat>& rvecs, vector<Mat>& tvecs,
                    vector<float>& reprojErrs,  double& totalAvgErr){

    cameraMatrix = Mat::eye(3, 3, CV_64F);
    distCoeffs = Mat::zeros(8, 1, CV_64F);

    float squareWidth = 106; //106mm 400.62992126px
```

```
    float squareHeight = 105; //105mm 396.8503937px
    vector<vector<Point3f>> objectPoints(1);;
    objectPoints[0].push_back(Point3f(0,0,0));
    objectPoints[0].push_back(Point3f(squareHeight,squareWidth,0));
    objectPoints[0].push_back(Point3f(squareHeight,0,0));
    objectPoints[0].push_back(Point3f(0,squareWidth,0));

    objectPoints.resize(imagePoints.size(),objectPoints[0]);

    // //Find intrinsic and extrinsic camera parameters
    double rms = calibrateCamera(objectPoints, imagePoints, imageSize,
    cameraMatrix,
                                 distCoeffs, rvecs, tvecs,  cv::CALIB_FIX_K4| cv
    ::CALIB_FIX_K5);

    cout << "Re-projection error reported by calibrateCamera: "<< rms << endl;

    bool ok = checkRange(cameraMatrix) && checkRange(distCoeffs);

    totalAvgErr = computeReprojectionErrors(objectPoints, imagePoints,
                                            rvecs, tvecs, cameraMatrix,
    distCoeffs, reprojErrs);

    return ok;
}

bool saveCalibration(string name, Mat& cameraMatrix, Mat& distCoeffs){

    outstream.open(name, std::ios_base::app);
    if (outstream){

        outstream << "Camera Coefficient"<< endl;
        for (int r=0; r < cameraMatrix.rows; r++){
            for (int c=0; c < cameraMatrix.cols; c++){
                double value = cameraMatrix.at<double>(r,c);
                outstream << value << endl;
            }
        }

        outstream << "Distortion Coefficients "<< endl;
        for (int r=0; r < distCoeffs.rows; r++){
            for (int c=0; c < distCoeffs.cols; c++){
                double value = distCoeffs.at<double>(r,c);
                outstream << value << endl;
            }
        }
        return true;
    }

    return false;


}

Mat nextImage(){
    Mat result;
    if( inputCapture.isOpened() )
```

```
    {
        Mat view0;
        inputCapture >> view0;
        view0.copyTo(result);
    }
    return result;
}


int main(int argc, char* argv[])
{
    vector<cv::String> fn;
    string img_dir =  "/code/box-example/image/*";
    glob(img_dir, fn, false);
    size_t count_fn = fn.size();
    Mat cur_img;
    vector<vector<Point2f> > imagePoints;
    // Calibration Param
    cv::Size imageSize;
    Mat cameraMatrix, distCoeffs;
    vector<Mat> rvecs, tvecs;
    vector<float> reprojErrs;
    double totalAvgErr;
    int num_valid_photo = 0;

    inputCapture.open(1,cv::CAP_V4L2);
    if( !inputCapture.isOpened() )
    {
        cout << "Cant find camera" << endl;
        return 0;
    }

    for(size_t i=0; i<count_fn; i++){
    // while(num_valid_photo < 20){

        cout << "Image: " << i << endl;

        bool blinkOutput = false; //to flip the image

        //load in an image
        cur_img = imread(fn[i]);
        // cur_img = nextImage();
        // imshow("video stream",cur_img);
        // waitKey(100);

        //find the presence of the box
        vector<Point2f> pointBuf;
        FindBox box;
        bool found = box.getBox(cur_img, pointBuf);

        if (found){
            num_valid_photo ++;

            std::string name = "./image_v/" + to_string(num_valid_photo) +".jpg";
            cv::imwrite(name, cur_img);

            imagePoints.push_back(pointBuf);
```

```
            blinkOutput = true;
        }

        if(blinkOutput){
            bitwise_not(cur_img, cur_img);
        }


    }
    cout << "Number of Valid Photos: " << num_valid_photo << endl;
    cout << "Avg_Reprojection_Error: " << totalAvgErr << endl;;

    if (num_valid_photo > 0){
        imageSize = cur_img.size();
        runCalibration(imageSize, cameraMatrix, distCoeffs,
                    imagePoints, rvecs, tvecs,
                    reprojErrs, totalAvgErr);
        saveCalibration("result", cameraMatrix, distCoeffs);

        Mat view, rview, map1, map2;
        initUndistortRectifyMap(cameraMatrix, distCoeffs, Mat(),
            getOptimalNewCameraMatrix(cameraMatrix, distCoeffs, imageSize, 1,
    imageSize, 0),
            imageSize, CV_16SC2, map1, map2);


        vector<cv::String> fn2;
        string img_dir2 =  "/code/box-example/image_v/*";
        glob(img_dir2, fn2, false);
        for(int i = 0; i < (int)fn2.size(); i++ )
        {
            view = imread(fn2[i], 1);

            if(view.empty())
                continue;
            remap(view, rview, map1, map2, INTER_LINEAR);

            std::string name2 = "./image_un/undistorded_" + to_string(i) +".jpg";
            cv::imwrite(name2, rview);

            imshow("Image View", rview);
            waitKey(0);
            // char c = (char)waitKey();
            // if( c  == ESC_KEY || c == 'q' || c == 'Q' )
            //     break;
        }

    } else {
        cout << "Not enough photos" << endl;
    }

    inputCapture.release();
    outstream.close();
    return 0;
}
```

Below shows the module for corner detection. *getBox()* is the main wrapper function.

```
#include "findBox.hpp"
bool FindBox::getBox (Mat& img, vector<Point2f>& pointBuf){
    Mat dst, grey_img;

    //turn to gray scale
    cvtColor(img, grey_img, COLOR_BGR2GRAY);

    //dilate
    int dilation_size = 0;
    Mat element = getStructuringElement( MORPH_RECT,
                        Size( 2*dilation_size + 1, 2*dilation_size+1 ),
                        Point( dilation_size, dilation_size ) );
    cv::dilate(grey_img, dst, element);

    vector<Point> poly;
    vector<Point> corners = getCorners(dst, img, poly);

    if (corners.size() < 4 || corners.size() >45){
        cout << "Failed. No. of detected corners:" << corners.size() << endl;
        return false;
    }

    std::rotate(corners.begin(),
                corners.begin()+2, // this will be the new first element
                corners.end());

    Mat mask = cv::Mat::zeros(dst.rows, dst.cols, CV_8U);
    const Point* ppt[1] = { &poly[0] };
    int num = poly.size();
    int npt[] = {num};
    fillPoly(mask,
            ppt,
            npt,
            1,
            Scalar( 255, 255, 255 ),
            8);
    int mask_sz = countNonZero(mask);
    if (mask_sz < 1000){
        cout << "Failed2. Mask too small." << endl;
        return false;
    }

    vector<Point> sq;
    vector<vector<int>> comb =  getcomb(corners.size(), 4);
    vector<vector<Point>> pot_sq;
    vector<double> ptw;
    bool found =false;
    for (auto c: comb){
        sq = {corners[c[0]],corners[c[1]],corners[c[2]],corners[c[3]]};
        //check area of bounding box
        cv::RotatedRect rectangle = cv::minAreaRect(sq);
        auto sz = rectangle.size;
        float area = sz.height*sz.width;
        if ( area < FindBox::SQ_HALF_AREA){
            continue;
        }
```

```
        double white_space = checkSquare(mask, mask_sz, sq);
        if (white_space < FindBox::SQ_AREA_THRES){
            found = true;
            cout << "Detected area of white: " << white_space << endl;
            ptw.push_back(white_space);
            pot_sq.push_back(sq);
        }
    }
    if (!found){
        cout << "Cannot find 4 pt for rectangle" << endl;
        return false;
    }
    sq = SquareSanityCheck(pot_sq, mask_sz, ptw, mask);

    //find the main corner point
    vector<Point> ans(4);
    int g=0;
    bool found_main = false;
    vector<float> shapesz_vec;
    for (g; g<4; g++){
        cv::Mat mask2 = cv::Mat::zeros(img.rows, img.cols, CV_8U);
        cv::circle(mask2, sq[g], 50, (255), -1, FILLED, 0);
        cv::Mat res;
        cv::bitwise_and(mask2 , mask, res);
        vector<Point> p;
        goodFeaturesToTrack(res, p, 6, 0.5, 5);
        if (p.size() != 3){
            found_main =  true;
            break;
        }
    }
    if (!found_main){
        cout << "cant find main corner" << endl;
        // g = 0;
        return false;
    }
    ans[0] = sq[g];
    sq.erase(sq.begin()+g);

    //find the opposite corner of the main corner
    vector<float> dist;
    for (auto points: sq){
        dist.push_back(getDistance(ans[0], points));
    }
    g = std::distance(dist.begin(),std::max_element(dist.begin(), dist.end()));
    ans[1] = sq[g];
    sq.erase(sq.begin()+g);

    // find the corner that connect with the main point through a the broken line segment
    vector<float> num_nonzero;
    for (auto points: sq){
        Mat test_img = mask.clone();
        cv::line(test_img, ans[0], points, (255),1);
        num_nonzero.push_back(countNonZero(test_img));
    }
    g = (num_nonzero[0] > num_nonzero[1])? 0 : 1;
```

```
    ans[2] = sq[g];
    sq.erase(sq.begin()+g);
    ans[3] = sq[0];

    for (auto elem: ans){
        pointBuf.push_back(Point2f(float(elem.x), float(elem.y)));
    }

    cornerSubPix(grey_img, pointBuf, Size(11,11), Size(-1,-1), TermCriteria(
    TermCriteria::EPS+TermCriteria::COUNT, 30, 0.1 ));

    cout << "Acceptable square? " << endl;
    for (int k=0; k<pointBuf.size(); k++){
        putText(grey_img, to_string(k), Point2f(pointBuf[k].x+50, pointBuf[k].y
    +50),0,2,(0,0,255));
        circle(grey_img,pointBuf[k] , 5, (0, 0, 255), 3);
    }
    imshow("out", grey_img);
    waitKey(50);
    char user_ans;
    cin >> user_ans ;
    if (user_ans == 'y'){
        return true;
    }

    return false;
}

float FindBox::getDistance(Point p1, Point p2){
    return (pow(p1.x - p2.x,2) + pow(p1.y - p2.y,2));
}

vector<Point> FindBox::SquareSanityCheck(vector<vector<Point>> pot_sq, int
    mask_sz, vector<double> white_space,const Mat& ref_mat){
    vector<float> arr;
    for (auto sq: pot_sq){
        Mat oursq = cv::Mat::zeros(ref_mat.rows, ref_mat.cols, CV_8U);
        fillConvexPoly(oursq,sq,Scalar(255,255,255));

        Mat res;
        cv::bitwise_xor(oursq, ref_mat, res);
        float a = countNonZero(res)*100/mask_sz;
        arr.push_back(a);
    }

    return pot_sq[std::distance(arr.begin(),std::min_element(arr.begin(), arr.end
    ()))];

}

double FindBox::checkSquare(const Mat& arg_mat, int mask_sz, vector<Point>
    arg_corners){

    vector<vector<int>> comb = getcomb(4,2);
    for(auto c: comb){
        if (getDistance(arg_corners[c[0]], arg_corners[c[1]]) < SQ_PROXIMITY){
            cout << "Points too close together to be considered as square" <<
```

```
    endl;
            return 100;
        }
    }
    Mat mat = arg_mat.clone();

    fillConvexPoly(mat,arg_corners,Scalar(0,0,0));
    int count = countNonZero(mat);
    double per = count*100/mask_sz;

    vector<Point> corners;
    if (per < FindBox::SQ_AREA_THRES){
        bool parallel = detectparallels(arg_corners, corners);
        if (parallel){
            return per;
        }
    }
    return 100;
}

bool FindBox::detectparallels(const vector<Point>& arr, vector<Point>& output){

    vector<vector<int>> comb = getcomb(4,2);
    vector<float> slopes;
    for (auto c: comb){
        slopes.push_back(getSlope(arr[c[0]].x, arr[c[0]].y, arr[c[1]].x, arr[c
    [1]].y));
    }

    std::sort(slopes.begin(),slopes.end());
    vector<float> compare(3);

    compare[0] = abs(slopes[1]-slopes[0]);
    compare[1] = abs(slopes[3]-slopes[2]);
    compare[2] = abs(slopes[5]-slopes[4]);

    for (auto elem: slopes){
        cout << "slope: " << elem << endl;
    }

    int count = 0;
    for (auto elem:compare){
        if (elem < FindBox::SQ_PARAL_THRES){
            count ++;
        }
    }

    if (count > 1){
        cout << "Can detect parallel!!!" << endl;
        return true;
    }
    cout << "Cannot detect parallel" << endl;
    return false;

}

float FindBox::getSlope(float x1, float y1, float x2, float y2){
```

```
    vector<float> ans;
    float m = (y2-y1)/(x2-x1);
    return m;
}



vector<vector<int>> FindBox::getcomb(int N, int K)
{
    vector<vector<int>> results;
    std::string bitmask(K, 1); // K leading 1's
    bitmask.resize(N, 0); // N-K trailing 0's

    // print integers and permute bitmask
    do {
        vector<int> arr;
        for (int i = 0; i < N; ++i) // [0..N-1] integers
        {
            if (bitmask[i]) arr.push_back(i);
        }
        results.push_back(arr);
    } while (std::prev_permutation(bitmask.begin(), bitmask.end()));

    return results;
}


vector<Point> FindBox::getCorners(Mat& grey_mat, Mat& mat, vector<Point>&
    poly_mask){
    float masksizeThres,  FastDectorThres;
    bool cam = FindBox::SONY;
    if (cam){
        masksizeThres = 5000;
        FastDectorThres = 10;
    } else {
        masksizeThres = 1000;
        FastDectorThres = 5;
    }

    Mat grad_x, grad_y, abs_grad_x, abs_grad_y, sobel_mat, canny_mat;
    vector<vector<Point>> contours;
    // vector<Point> poly;

    int scale = 1;
  int delta = 0;
  int ddepth = CV_16S;

    //Blur img
    GaussianBlur(grey_mat, grey_mat, Size(5,5), 0, 0, BORDER_DEFAULT );

    //Sobel
    // Gradient X
  Sobel( grey_mat, grad_x, ddepth, 1, 0, 3, scale, delta, BORDER_DEFAULT );
  // Gradient Y
  Sobel( grey_mat, grad_y, ddepth, 0, 1, 3, scale, delta, BORDER_DEFAULT );
  convertScaleAbs( grad_x, abs_grad_x );
  convertScaleAbs( grad_y, abs_grad_y );
```

```
 addWeighted( abs_grad_x, 0.5, abs_grad_y, 0.5, 0, sobel_mat);

  //Canny
  int thresh = 50;
  Canny(sobel_mat, canny_mat, thresh, 150);


  vector<Vec4i> hierarchy;
  findContours(canny_mat, contours, hierarchy, RETR_EXTERNAL,
 CHAIN_APPROX_SIMPLE);

  vector<vector<Point>> poly(contours.size());
  for( size_t i = 0; i < contours.size(); i++ )
  {
      approxPolyDP( contours[i], poly[i], 3, true );
  }
  float max_area = 0;
  int max_poly_idx = 0;

  for(size_t i = 0 ; i < poly.size() ; i++){
      if (poly[i].size()>1){
          double a = contourArea(poly[i]);
          if (a > max_area){
              max_poly_idx = i;
          }
      }
  }
  poly_mask = poly[max_poly_idx];

  Mat mask = cv::Mat::zeros(mat.rows, mat.cols, CV_8U);
  const Point* ppt[1] = { &poly_mask[0] };
  int num = poly_mask.size();
  int npt[] = {num};
  fillPoly(mask,
          ppt,
          npt,
          1,
          Scalar( 255, 255, 255 ),
          8);
  int mask_sz = countNonZero(mask);
  if (mask_sz < masksizeThres){
      cout << "Failed1. Mask too small." << endl;
      vector<Point> dummy;
      return dummy;
  }

  Ptr<FastFeatureDetector> fastDetector = FastFeatureDetector::create(
 FastDectorThres, true);
  std::vector<cv::KeyPoint> keypoints;
  fastDetector->detect(grey_mat, keypoints);
  Mat mask2, mask3, mask4;
  int morph_size = 10;
  Mat element_er = getStructuringElement( MORPH_ELLIPSE, cv::Size( 2*morph_size
  + 10, 2*morph_size +10), cv::Point( morph_size, morph_size ) );
  erode(mask, mask2, element_er);
  dilate(mask, mask3, element_er);

```

```
    cv::bitwise_xor(mask2,  mask3, mask4);
    vector<Point> cen_final;

    for (auto elem: keypoints){
        uchar val = mask4.at< uchar >(elem.pt);
        if ( val == 255){
            cen_final.push_back(elem.pt);

        }
    }
    return cen_final;

}


cv::Point FindBox::getAngle(const cv::Point& p1, const cv::Point& p2, const cv::
    Point& p3){

    double angle = getAnglehelper(p1,p2,p3);
    if (angle < 110 && angle > 70){
        return p1;
    }

    angle = getAnglehelper(p2,p1,p3);
    if (angle < 110 && angle > 70){
        return p2;
    }

    angle = getAnglehelper(p3,p1,p2);
    if (angle < 110 && angle > 70){
        return p3;
    }

    return cv::Point(-1,-1);

}

double FindBox::getAnglehelper(const cv::Point& center, const cv::Point& point,
    const cv::Point& base){
    double angle = std::atan2(point.y - center.y, point.x - center.x) * 180 /
    3.141592;
    angle = (angle < 0) ? (360 + angle) : angle;
    angle = 360 - angle;
    return angle;
}
```