Camera calibration With OpenCV

Prev Tutorial: Camera calibration with square chessboard
Next Tutorial: Real Time pose estimation of a textured object

Original author	Bernát Gábor
Compatibility	OpenCV >= 4.0

Cameras have been around for a long-long time. However, with the introduction of the cheap *pinhole* cameras in the late 20th century, they became a common occurrence in our everyday life. Unfortunately, this cheapness comes with its price: significant distortion. Luckily, these are constants and with a calibration and some remapping we can correct this. Furthermore, with calibration you may also determine the relation between the camera's natural units (pixels) and the real world units (for example millimeters).

Theory

For the distortion OpenCV takes into account the radial and tangential factors. For the radial factor one uses the following formula:

$$x_{distorted} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) y_{distorted} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

So for an undistorted pixel point at (x, y) coordinates, its position on the distorted image will be $(x_{distorted} y_{distorted})$. The presence of the radial distortion manifests in form of the "barrel" or "fish-eye" effect.

Tangential distortion occurs because the image taking lenses are not perfectly parallel to the imaging plane. It can be represented via the formulas:

$$x_{distorted} = x + [2p_1 xy + p_2(r^2 + 2x^2)] y_{distorted} = y + [p_1(r^2 + 2y^2) + 2p_2 xy]$$

So we have five distortion parameters which in OpenCV are presented as one row matrix with 5 columns:

$$distortion\_coefficients = (k_1 \quad k_2 \quad p_1 \quad p_2 \quad k_3)$$

Now for the unit conversion we use the following formula:

$$\begin{bmatrix} x \\ y \\ w \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

Here the presence of w is explained by the use of homography coordinate system (and $w = Z$). The unknown parameters are f_x and f_y (camera focal lengths) and (c_x, c_y) which are the optical centers expressed in pixels coordinates. If for both axes a common focal length is used with a given a aspect ratio (usually 1), then $f_y = f_x * a$ and in the upper formula we will have a single focal length f. The matrix containing these four parameters is referred to as the *camera matrix*. While the distortion coefficients are the same regardless of the camera resolutions used, these should be scaled along with the current resolution from the calibrated resolution.

The process of determining these two matrices is the calibration. Calculation of these parameters is done through basic geometrical equations. The equations used depend on the chosen calibrating objects. Currently OpenCV supports three types of objects for calibration:

- Classical black-white chessboard
- ChArUco board pattern
- Symmetrical circle pattern
- Asymmetrical circle pattern

Basically, you need to take snapshots of these patterns with your camera and let OpenCV find them. Each found pattern results in a new equation. To solve the equation you need at least a predetermined number of pattern snapshots to form a well-posed equation system. This number is higher for the chessboard pattern and less for the circle ones. For example, in theory the chessboard pattern requires at least two snapshots. However, in practice we have a good amount of noise present in our input images, so for good results you will probably need at least 10 good snapshots of the input pattern in different positions.

Goal

The sample application will:

- Determine the distortion matrix
- Determine the camera matrix
- Take input from Camera, Video and Image file list
- Read configuration from XML/YAML file

- Save the results into XML/YAML file
- Calculate re-projection error

Source code

You may also find the source code in the `samples/cpp/tutorial_code/calib3d/camera_calibration/` folder of the OpenCV source library or download it from here. For the usage of the program, run it with `-h` argument. The program has an essential argument: the name of its configuration file. If none is given then it will try to open the one named "default.xml". Here's a sample configuration file in XML format. In the configuration file you may choose to use camera as an input, a video file or an image list. If you opt for the last one, you will need to create a configuration file where you enumerate the images to use. Here's an example of this. The important part to remember is that the images need to be specified using the absolute path or the relative one from your application's working directory. You may find all this in the samples directory mentioned above.

The application starts up with reading the settings from the configuration file. Although, this is an important part of it, it has nothing to do with the subject of this tutorial: *camera calibration*. Therefore, I've chosen not to post the code for that part here. Technical background on how to do this you can find in the **File Input and Output using XML / YAML / JSON files** tutorial.

Explanation

1. **Read the settings**

```
Settings s;
const string inputSettingsFile = parser.get<string>(0);
FileStorage fs(inputSettingsFile, FileStorage::READ); // Read the settings
if (!fs.isOpened())
{
    cout << "Could not open the configuration file: \"" << inputSettingsFile << "\"" << endl;
    parser.printMessage();
    return -1;
}
fs["Settings"] >> s;
fs.release();                                         // close Settings file
```

   For this I've used simple OpenCV class input operation. After reading the file I've an additional post-processing function that checks validity of the input. Only if all inputs are good then *goodInput* variable will be true.

2. **Get next input, if it fails or we have enough of them - calibrate**

   After this we have a big loop where we do the following operations: get the next image from the image list, camera or video file. If this fails or we have enough images then we run the calibration process. In case of image we step out of the loop and otherwise the remaining frames will be undistorted (if the option is set) via changing from *DETECTION* mode to the *CALIBRATED* one.

```
for(;;)
{
    Mat view;
    bool blinkOutput = false;

    view = s.nextImage();

    //-----  If no more image, or got enough, then stop calibration and show result -------------
    if( mode == CAPTURING && imagePoints.size() >= (size_t)s.nrFrames )
    {
      if(runCalibrationAndSave(s, imageSize,  cameraMatrix, distCoeffs, imagePoints, grid_width,
                               release_object))
          mode = CALIBRATED;
      else
          mode = DETECTION;
    }
    if(view.empty())          // If there are no more images stop the loop
    {
        // if calibration threshold was not reached yet, calibrate now
        if( mode != CALIBRATED && !imagePoints.empty() )
            runCalibrationAndSave(s, imageSize,  cameraMatrix, distCoeffs, imagePoints, grid_width,
                                  release_object);
        break;
    }
```

   For some cameras we may need to flip the input image. Here we do this too.

3. **Find the pattern in the current input**

The formation of the equations I mentioned above aims to finding major patterns in the input: in case of the chessboard this are corners of the squares and for the circles, well, the circles themselves. ChArUco board is equivalent to chessboard, but corners are mached by ArUco markers. The position of these will form the result which will be written into the *pointBuf* vector.

```
        vector<Point2f> pointBuf;

        bool found;

        int chessBoardFlags = CALIB_CB_ADAPTIVE_THRESH | CALIB_CB_NORMALIZE_IMAGE;

        if(!s.useFisheye) {
            // fast check erroneously fails with high distortions like fisheye
            chessBoardFlags |= CALIB_CB_FAST_CHECK;
        }

        switch( s.calibrationPattern ) // Find feature points on the input format
        {
        case Settings::CHESSBOARD:
            found = findChessboardCorners( view, s.boardSize, pointBuf, chessBoardFlags);
            break;
        case Settings::CHARUCOBOARD:
            ch_detector.detectBoard( view, pointBuf, markerIds);
            found = pointBuf.size() == (size_t)((s.boardSize.height - 1)*(s.boardSize.width - 1));
            break;
        case Settings::CIRCLES_GRID:
            found = findCirclesGrid( view, s.boardSize, pointBuf );
            break;
        case Settings::ASYMMETRIC_CIRCLES_GRID:
            found = findCirclesGrid( view, s.boardSize, pointBuf, CALIB_CB_ASYMMETRIC_GRID );
            break;
        default:
            found = false;
            break;
        }
```

Depending on the type of the input pattern you use either the **cv::findChessboardCorners** or the **cv::findCirclesGrid** function or **cv::aruco::CharucoDetector::detectBoard** method. For all of them you pass the current image and the size of the board and you'll get the positions of the patterns. **cv::findChessboardCorners** and **cv::findCirclesGrid** return a boolean variable which states if the pattern was found in the input (we only need to take into account those images where this is true!). `CharucoDetector::detectBoard` may detect partially visible pattern and returns coordunates and ids of visible inner corners.

Note

Board size and amount of matched points is different for chessboard, circles grid and ChArUco. All chessboard related algorithm expects amount of inner corners as board width and height. Board size of circles grid is just amount of circles by both grid dimentions. ChArUco board size is defined in squares, but detection result is list of inner corners and that's why is smaller by 1 in both dimentions.

Then again in case of cameras we only take camera images when an input delay time is passed. This is done in order to allow user moving the chessboard around and getting different images. Similar images result in similar equations, and similar equations at the calibration step will form an ill-posed problem, so the calibration will fail. For square images the positions of the corners are only approximate. We may improve this by calling the **cv::cornerSubPix** function. (`winSize` is used to control the side length of the search window. Its default value is 11. `winSize` may be changed by command line parameter `--winSize=<number>` .) It will produce better calibration result. After this we add a valid inputs result to the *imagePoints* vector to collect all of the equations into a single container. Finally, for visualization feedback purposes we will draw the found points on the input image using **cv::findChessboardCorners** function.

```
        if (found)                // If done with success,
        {
              // improve the found corners' coordinate accuracy for chessboard
                if( s.calibrationPattern == Settings::CHESSBOARD)
                {
                    Mat viewGray;
                    cvtColor(view, viewGray, COLOR_BGR2GRAY);
                    cornerSubPix( viewGray, pointBuf, Size(winSize,winSize),
                        Size(-1,-1), TermCriteria( TermCriteria::EPS+TermCriteria::COUNT, 30, 0.0001 ));
                }

                if( mode == CAPTURING &&  // For camera only take new samples after delay time
                    (!s.inputCapture.isOpened() || clock() - prevTimestamp > s.delay*1e-3*CLOCKS_PER_SEC) )
```

```
            {
                imagePoints.push_back(pointBuf);
                prevTimestamp = clock();
                blinkOutput = s.inputCapture.isOpened();
            }

            // Draw the corners.
            if(s.calibrationPattern == Settings::CHARUCOBOARD)
                drawChessboardCorners( view, cv::Size(s.boardSize.width-1, s.boardSize.height-1), Mat(pointBuf),
    found );
            else
                drawChessboardCorners( view, s.boardSize, Mat(pointBuf), found );
        }
```

4. **Show state and result to the user, plus command line control of the application**

   This part shows text output on the image.

```
        string msg = (mode == CAPTURING) ? "100/100" :
                      mode == CALIBRATED ? "Calibrated" : "Press 'g' to start";
        int baseLine = 0;
        Size textSize = getTextSize(msg, 1, 1, 1, &baseLine);
        Point textOrigin(view.cols - 2*textSize.width - 10, view.rows - 2*baseLine - 10);

        if( mode == CAPTURING )
        {
            if(s.showUndistorted)
                msg = cv::format( "%d/%d Undist", (int)imagePoints.size(), s.nrFrames );
            else
                msg = cv::format( "%d/%d", (int)imagePoints.size(), s.nrFrames );
        }

        putText( view, msg, textOrigin, 1, 1, mode == CALIBRATED ?  GREEN : RED);

        if( blinkOutput )
            bitwise_not(view, view);
```

   If we ran calibration and got camera's matrix with the distortion coefficients we may want to correct the image using **cv::undistort** function:

```
        if( mode == CALIBRATED && s.showUndistorted )
        {
            Mat temp = view.clone();
            if (s.useFisheye)
            {
                Mat newCamMat;
                fisheye::estimateNewCameraMatrixForUndistortRectify(cameraMatrix, distCoeffs, imageSize,
                                                                    Matx33d::eye(), newCamMat, 1);
                cv::fisheye::undistortImage(temp, view, cameraMatrix, distCoeffs, newCamMat);
            }
            else
              undistort(temp, view, cameraMatrix, distCoeffs);
        }
```

   Then we show the image and wait for an input key and if this is *u* we toggle the distortion removal, if it is *g* we start again the detection process, and finally for the *ESC* key we quit the application:

```
        imshow("Image View", view);
        char key = (char)waitKey(s.inputCapture.isOpened() ? 50 : s.delay);

        if( key  == ESC_KEY )
            break;

        if( key == 'u' && mode == CALIBRATED )
           s.showUndistorted = !s.showUndistorted;

        if( s.inputCapture.isOpened() && key == 'g' )
        {
            mode = CAPTURING;
            imagePoints.clear();
        }
```

5. **Show the distortion removal for the images too**

When you work with an image list it is not possible to remove the distortion inside the loop. Therefore, you must do this after the loop. Taking advantage of this now I'll expand the **cv::undistort** function, which is in fact first calls **cv::initUndistortRectifyMap** to find transformation matrices and then performs transformation using **cv::remap** function. Because, after successful calibration map calculation needs to be done only once, by using this expanded form you may speed up your application:

```
    if( s.inputType == Settings::IMAGE_LIST && s.showUndistorted && !cameraMatrix.empty())
    {
        Mat view, rview, map1, map2;

        if (s.useFisheye)
        {
            Mat newCamMat;
            fisheye::estimateNewCameraMatrixForUndistortRectify(cameraMatrix, distCoeffs, imageSize,
                                                                Matx33d::eye(), newCamMat, 1);
            fisheye::initUndistortRectifyMap(cameraMatrix, distCoeffs, Matx33d::eye(), newCamMat, imageSize,
                                             CV_16SC2, map1, map2);
        }
        else
        {
            initUndistortRectifyMap(
                cameraMatrix, distCoeffs, Mat(),
                getOptimalNewCameraMatrix(cameraMatrix, distCoeffs, imageSize, 1, imageSize, 0), imageSize,
                CV_16SC2, map1, map2);
        }

        for(size_t i = 0; i < s.imageList.size(); i++ )
        {
            view = imread(s.imageList[i], IMREAD_COLOR);
            if(view.empty())
                continue;
            remap(view, rview, map1, map2, INTER_LINEAR);
            imshow("Image View", rview);
            char c = (char)waitKey();
            if( c  == ESC_KEY || c == 'q' || c == 'Q' )
                break;
        }
    }
```

The calibration and save

Because the calibration needs to be done only once per camera, it makes sense to save it after a successful calibration. This way later on you can just load these values into your program. Due to this we first make the calibration, and if it succeeds we save the result into an OpenCV style XML or YAML file, depending on the extension you give in the configuration file.

Therefore in the first function we just split up these two processes. Because we want to save many of the calibration variables we'll create these variables here and pass on both of them to the calibration and saving function. Again, I'll not show the saving part as that has little in common with the calibration. Explore the source file in order to find out how and what:

```
bool runCalibrationAndSave(Settings& s, Size imageSize, Mat& cameraMatrix, Mat& distCoeffs,
                           vector<vector<Point2f> > imagePoints, float grid_width, bool release_object)
{
    vector<Mat> rvecs, tvecs;
    vector<float> reprojErrs;
    double totalAvgErr = 0;
    vector<Point3f> newObjPoints;

    bool ok = runCalibration(s, imageSize, cameraMatrix, distCoeffs, imagePoints, rvecs, tvecs, reprojErrs,
                             totalAvgErr, newObjPoints, grid_width, release_object);
    cout << (ok ? "Calibration succeeded" : "Calibration failed")
         << ". avg re projection error = " << totalAvgErr << endl;

    if (ok)
        saveCameraParams(s, imageSize, cameraMatrix, distCoeffs, rvecs, tvecs, reprojErrs, imagePoints,
                         totalAvgErr, newObjPoints);
    return ok;
}
```

We do the calibration with the help of the **cv::calibrateCameraRO** function. It has the following parameters:

- The object points. This is a vector of *Point3f* vector that for each input image describes how should the pattern look. If we have a planar pattern (like a chessboard) then we can simply set all Z coordinates to zero. This is a collection of the points where these important points are present. Because, we use a single pattern for all the input images we can calculate this just once and multiply it for all the other input views. We calculate the corner points with the *calcBoardCornerPositions* function as:

```
static void calcBoardCornerPositions(Size boardSize, float squareSize, vector<Point3f>& corners,
                                     Settings::Pattern patternType /*= Settings::CHESSBOARD*/)
{
    corners.clear();

    switch(patternType)
    {
    case Settings::CHESSBOARD:
    case Settings::CIRCLES_GRID:
        for (int i = 0; i < boardSize.height; ++i) {
            for (int j = 0; j < boardSize.width; ++j) {
                corners.push_back(Point3f(j*squareSize, i*squareSize, 0));
            }
        }
        break;
    case Settings::CHARUCOBOARD:
        for (int i = 0; i < boardSize.height - 1; ++i) {
            for (int j = 0; j < boardSize.width - 1; ++j) {
                corners.push_back(Point3f(j*squareSize, i*squareSize, 0));
            }
        }
        break;
    case Settings::ASYMMETRIC_CIRCLES_GRID:
        for (int i = 0; i < boardSize.height; i++) {
            for (int j = 0; j < boardSize.width; j++) {
                corners.push_back(Point3f((2 * j + i % 2)*squareSize, i*squareSize, 0));
            }
        }
        break;
    default:
        break;
    }
}
```

And then multiply it as:

```
vector<vector<Point3f> > objectPoints(1);
calcBoardCornerPositions(s.boardSize, s.squareSize, objectPoints[0], s.calibrationPattern);
objectPoints[0][s.boardSize.width - 1].x = objectPoints[0][0].x + grid_width;
newObjPoints = objectPoints[0];

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

Note

If your calibration board is inaccurate, unmeasured, roughly planar targets (Checkerboard patterns on paper using off-the-shelf printers are the most convenient calibration targets and most of them are not accurate enough.), a method from **[255]** can be utilized to dramatically improve the accuracies of the estimated camera intrinsic parameters. This new calibration method will be called if command line parameter `-d=<number>` is provided. In the above code snippet, `grid_width` is actually the value set by `-d=<number>` . It's the measured distance between top-left (0, 0, 0) and top-right (s.squareSize*(s.boardSize.width-1), 0, 0) corners of the pattern grid points. It should be measured precisely with rulers or vernier calipers. After calibration, newObjPoints will be updated with refined 3D coordinates of object points.

- The image points. This is a vector of *Point2f* vector which for each input image contains coordinates of the important points (corners for chessboard and centers of the circles for the circle pattern). We have already collected this from **cv::findChessboardCorners** or **cv::findCirclesGrid** function. We just need to pass it on.
- The size of the image acquired from the camera, video file or the images.
- The index of the object point to be fixed. We set it to -1 to request standard calibration method. If the new object-releasing method to be used, set it to the index of the top-right corner point of the calibration board grid. See **cv::calibrateCameraRO** for detailed explanation.

```
int iFixedPoint = -1;
if (release_object)
    iFixedPoint = s.boardSize.width - 1;
```

- The camera matrix. If we used the fixed aspect ratio option we need to set f_x:

```
    cameraMatrix = Mat::eye(3, 3, CV_64F);
    if( !s.useFisheye && s.flag & CALIB_FIX_ASPECT_RATIO )
        cameraMatrix.at<double>(0,0) = s.aspectRatio;
```

- The distortion coefficient matrix. Initialize with zero.

```
distCoeffs = Mat::zeros(8, 1, CV_64F);
```

- For all the views the function will calculate rotation and translation vectors which transform the object points (given in the model coordinate space) to the image points (given in the world coordinate space). The 7-th and 8-th parameters are the output vector of matrices containing in the i-th position the rotation and translation vector for the i-th object point to the i-th image point.
- The updated output vector of calibration pattern points. This parameter is ignored with standard calibration method.
- The final argument is the flag. You need to specify here options like fix the aspect ratio for the focal length, assume zero tangential distortion or to fix the principal point. Here we use CALIB_USE_LU to get faster calibration speed.

```
rms = calibrateCameraRO(objectPoints, imagePoints, imageSize, iFixedPoint,
                        cameraMatrix, distCoeffs, rvecs, tvecs, newObjPoints,
                        s.flag | CALIB_USE_LU);
```

- The function returns the average re-projection error. This number gives a good estimation of precision of the found parameters. This should be as close to zero as possible. Given the intrinsic, distortion, rotation and translation matrices we may calculate the error for one view by using the **cv::projectPoints** to first transform the object point to image point. Then we calculate the absolute norm between what we got with our transformation and the corner/circle finding algorithm. To find the average error we calculate the arithmetical mean of the errors calculated for all the calibration images.

```
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, bool fisheye)
{
    vector<Point2f> imagePoints2;
    size_t totalPoints = 0;
    double totalErr = 0, err;
    perViewErrors.resize(objectPoints.size());

    for(size_t i = 0; i < objectPoints.size(); ++i )
    {
        if (fisheye)
        {
            fisheye::projectPoints(objectPoints[i], imagePoints2, rvecs[i], tvecs[i], cameraMatrix,
                                   distCoeffs);
        }
        else
        {
            projectPoints(objectPoints[i], rvecs[i], tvecs[i], cameraMatrix, distCoeffs, imagePoints2);
        }
        err = norm(imagePoints[i], imagePoints2, NORM_L2);

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

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

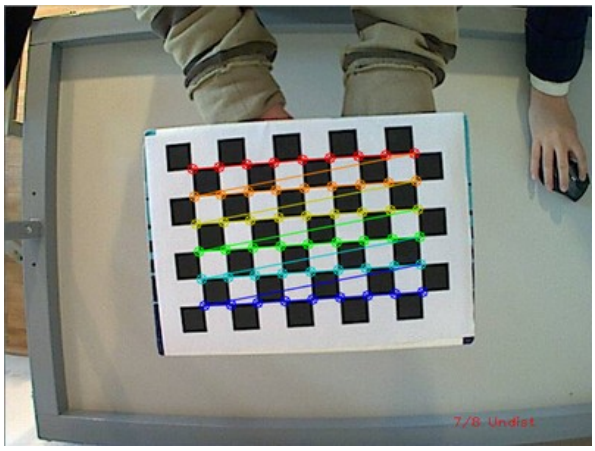
Results

Let there be this input chessboard pattern which has a size of 9 X 6. I've used an AXIS IP camera to create a couple of snapshots of the board and saved it into VID5 directory. I've put this inside the `images/CameraCalibration` folder of my working directory and created the following `VID5.XML` file that describes which images to use:

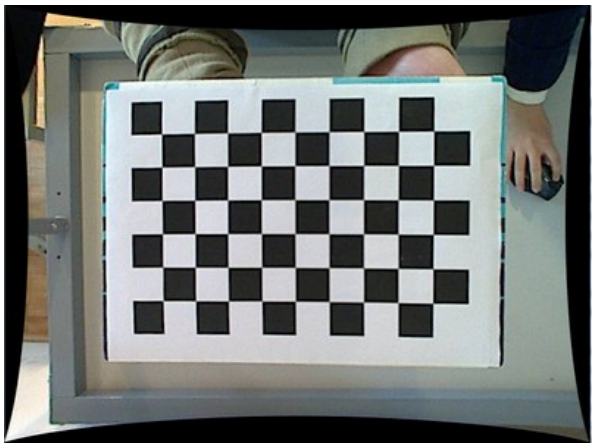
```
<?xml version="1.0"?>
<opencv_storage>
<images>
images/CameraCalibration/VID5/xx1.jpg
images/CameraCalibration/VID5/xx2.jpg
images/CameraCalibration/VID5/xx3.jpg
images/CameraCalibration/VID5/xx4.jpg
images/CameraCalibration/VID5/xx5.jpg
images/CameraCalibration/VID5/xx6.jpg
images/CameraCalibration/VID5/xx7.jpg
images/CameraCalibration/VID5/xx8.jpg
</images>
```

```
</opencv_storage>
```

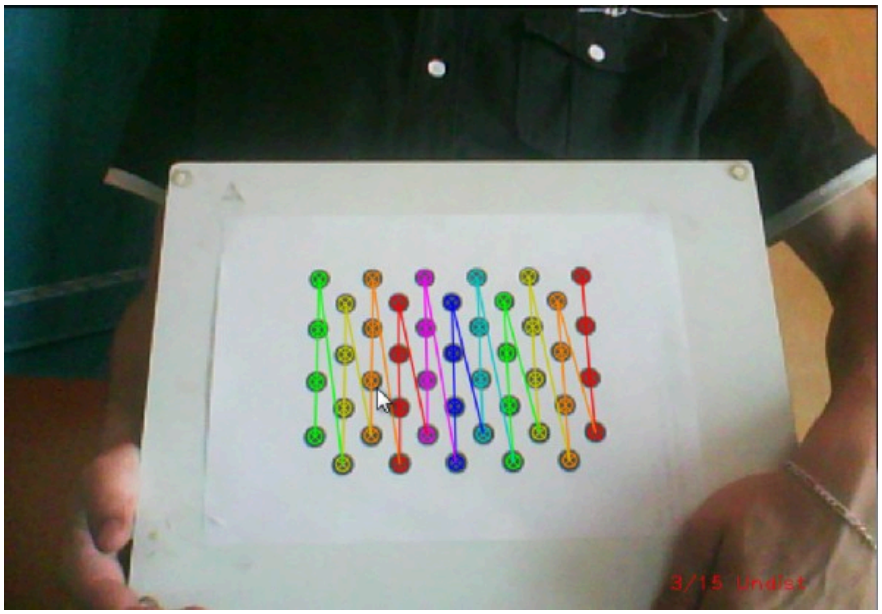
Then passed `images/CameraCalibration/VID5/VID5.XML` as an input in the configuration file. Here's a chessboard pattern found during the runtime of the application:



After applying the distortion removal we get:



The same works for this asymmetrical circle pattern by setting the input width to 4 and height to 11. This time I've used a live camera feed by specifying its ID ("1") for the input. Here's, how a detected pattern should look:



In both cases in the specified output XML/YAML file you'll find the camera and distortion coefficients matrices:

```
<camera_matrix type_id="opencv-matrix">
<rows>3</rows>
<cols>3</cols>
```

```
<dt>d</dt>
<data>
 6.5746697944293521e+002 0. 3.1950000000000000e+002 0.
 6.5746697944293521e+002 2.3950000000000000e+002 0. 0. 1.</data></camera_matrix>
<distortion_coefficients type_id="opencv-matrix">
<rows>5</rows>
<cols>1</cols>
<dt>d</dt>
<data>
 -4.1802327176423804e-001 5.0715244063187526e-001 0. 0.
 -5.7843597214487474e-001</data></distortion_coefficients>
```

Add these values as constants to your program, call the **cv::initUndistortRectifyMap** and the **cv::remap** function to remove distortion and enjoy distortion free inputs for cheap and low quality cameras.

You may observe a runtime instance of this on the YouTube here.