

HUYNH Cong Lap - Travail Seul
11419778 - Camera Calibration

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

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

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:

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

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

$$distortion_coefficients = (k_1 \ k_2 \ p_1 \ p_2 \ k_3)$$

Now for the unit conversion we use the following formula:

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

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 (in case of this TP)**
- Symmetrical circle pattern
- Asymmetrical circle pattern

We need to take snapshots of these patterns with camera and let OpenCV find them. Each found pattern results in a **new equation**. To solve the equation we need at least a **predetermined number of pattern snapshots** to form a well-posed equation system. 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.

Main Goal:

- Determine the distortion matrix
- Determine the camera matrix
- Take input from Camera, Video and Image file list
- Read configuration from YAML file
- Save the results into YAML file
- Calculate re-projection error

Main Process:

Find the pattern → Calibration → Compute Reprojection Errors → Undistort images

1. Find the pattern in the current input

We aim to finding major patterns in the input: in case of the chessboard this are corners. 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::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;
}
```

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. 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(11,11),
                     Size(-1,-1), TermCriteria( TermCriteria::EPS+TermCriteria::COUNT, 30, 0.1 ));
    }

    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.
    drawChessboardCorners( view, s.boardSize, Mat(pointBuf), found );
}
```

2. The calibration

Because the calibration needs to be done only once per camera, we need to save it after a successful calibration. This way later on we can just load these values into program.

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

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

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

We do the calibration with the help of the [cv::calibrateCamera](#) 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::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 + 1 % 2)*squareSize, i*squareSize, 0));
            break;
        default:
            break;
    }
}
```

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).

3. Compute Reprojection Errors

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);
}
```

4. Undistort images

We 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

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

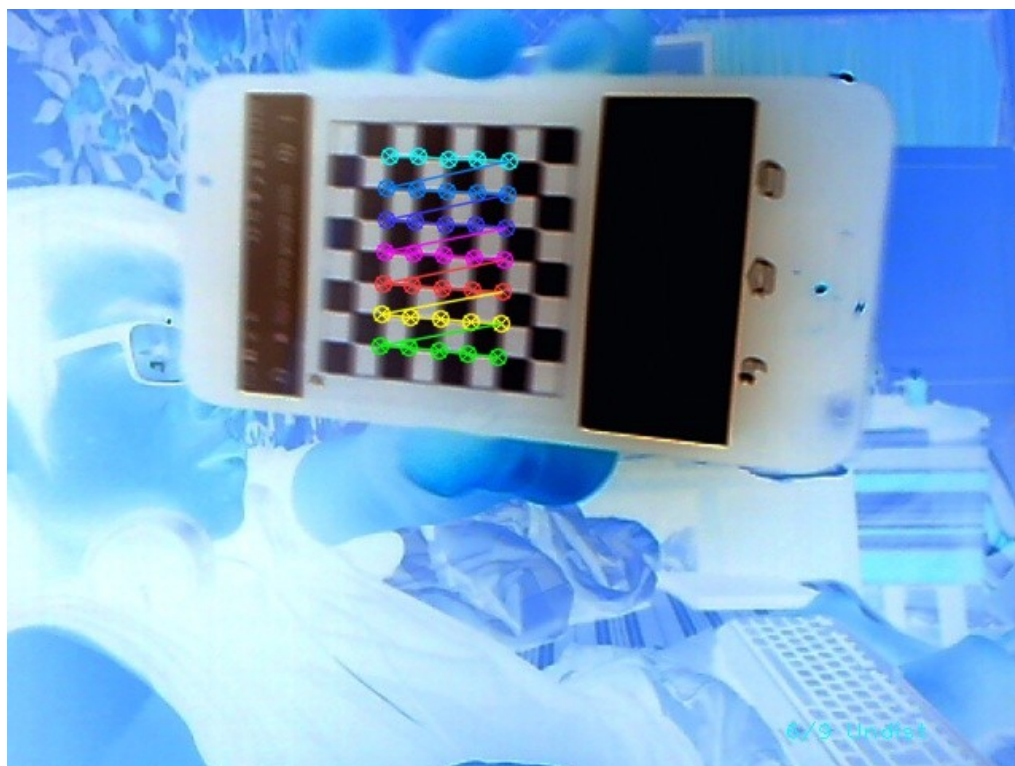
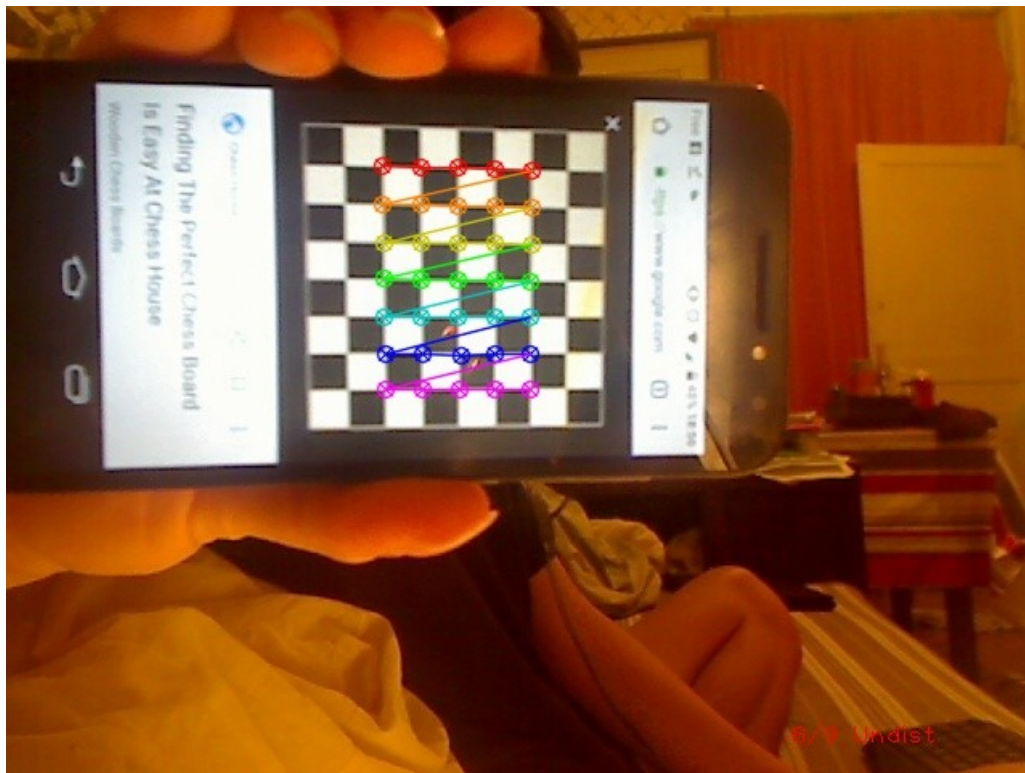
    if (s.useFisheye)
    {
        Mat newCamMat;
        fisheye::estimateNewCameraMatrixForUndistortRectify(cameraMatrix, distCoeffs, imageSize,
                                                            Matx33d::eye(), newCamMat, 1);
        fisheye::initUndistortRectifyMap(cameraMatrix, distCoeffs, 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;
    }
}
```


Result:

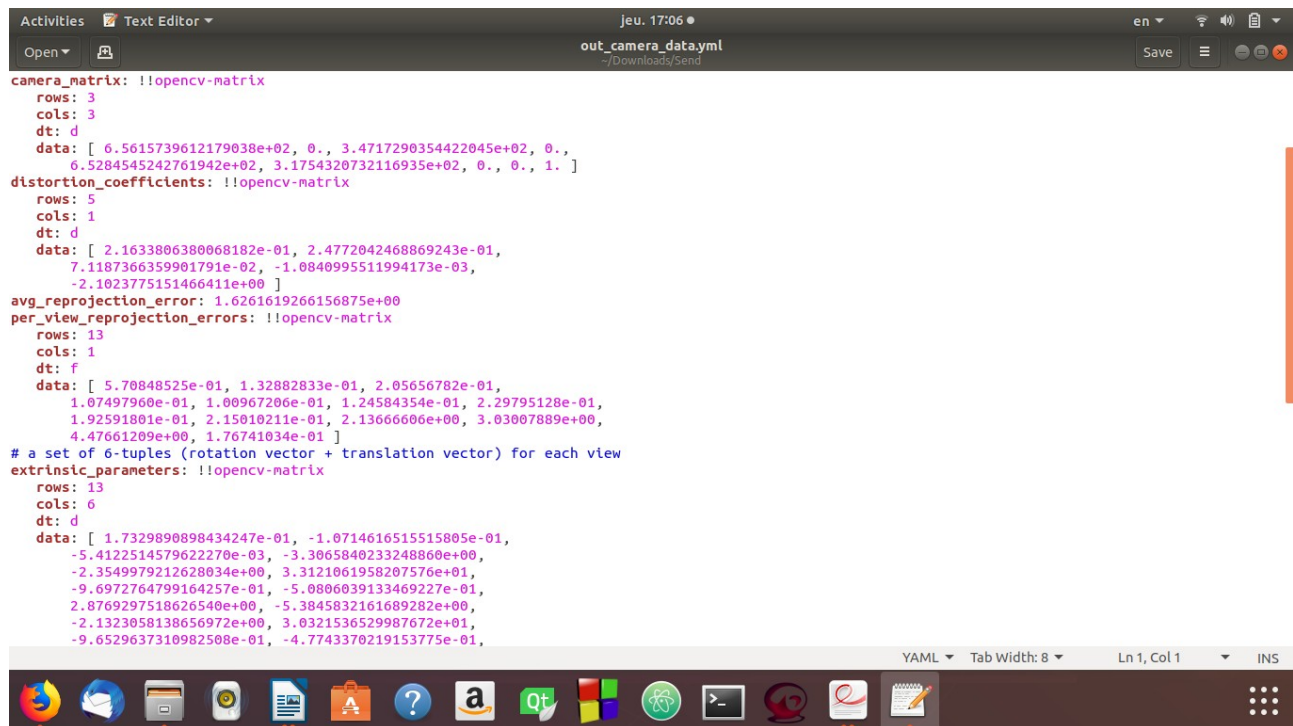
1. Find patterns:

Chessboard pattern found during the runtime of the application:



2 . Calibration:

After calibration step, we get a file .yaml output containing: **Distortion Matrix**, **Parameter Camera matrix - intrinsic (K)** and **parameters extrinsic (R|t)** as below

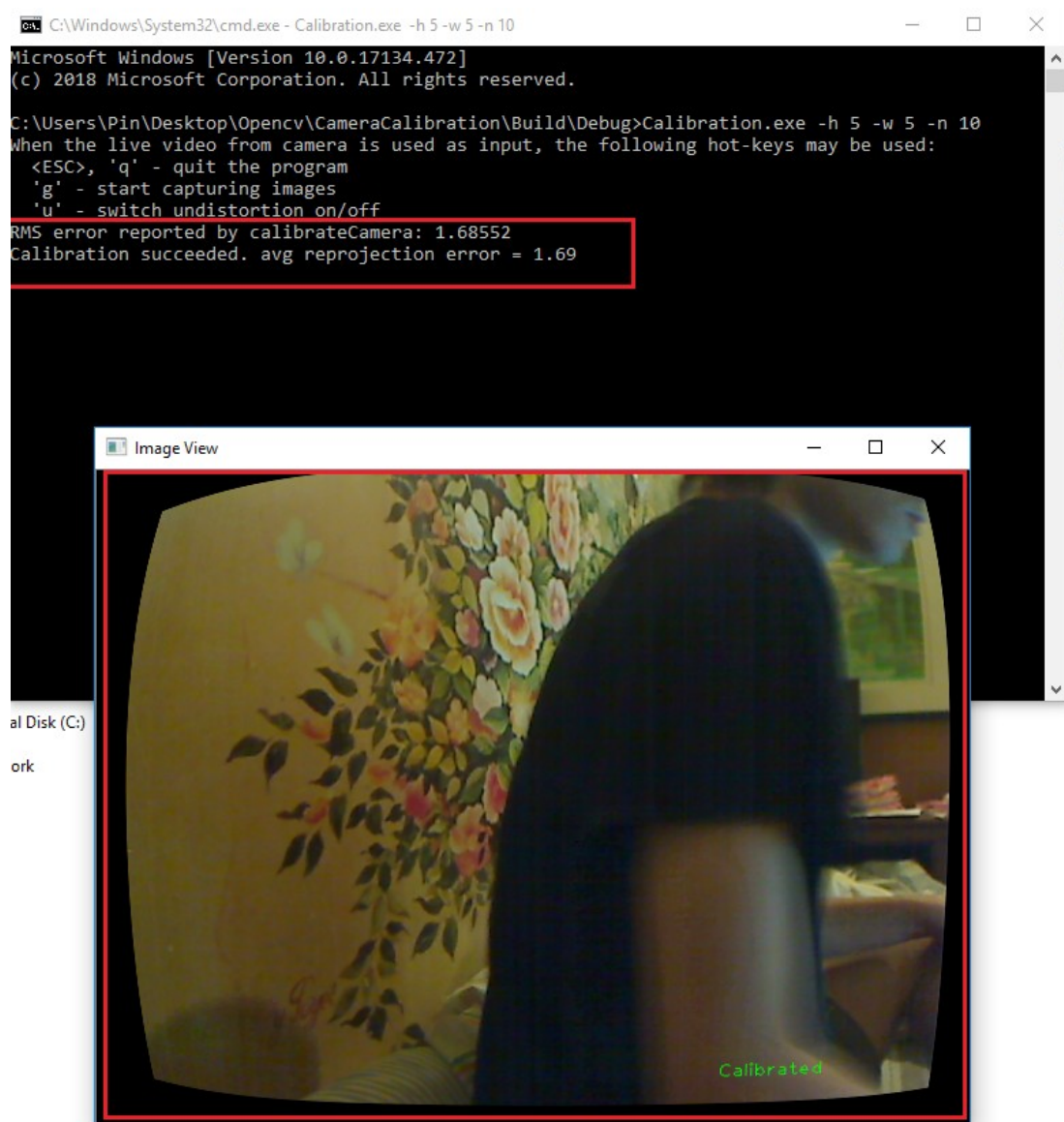


```
camera_matrix: !!opencv-matrix
  rows: 3
  cols: 3
  dt: d
  data: [ 6.5615739612179038e+02, 0., 3.4717290354422045e+02, 0.,
    6.5284545242761942e+02, 3.1754320732116935e+02, 0., 0., 1. ]
distortion_coefficients: !!opencv-matrix
  rows: 5
  cols: 1
  dt: d
  data: [ 2.1633806380068182e-01, 2.4772042468869243e-01,
    7.1187366359901791e-02, -1.0840995511994173e-03,
    -2.1023775151466411e+00 ]
avg_reprojection_error: 1.6261619266156875e+00
per_view_reprojection_errors: !!opencv-matrix
  rows: 13
  cols: 1
  dt: f
  data: [ 5.70848525e-01, 1.32882833e-01, 2.05656782e-01,
    1.07497960e-01, 1.00967206e-01, 1.24584354e-01, 2.29795128e-01,
    1.92591801e-01, 2.15010211e-01, 2.13666606e+00, 3.03007889e+00,
    4.47661209e+00, 1.76741034e-01 ]
# a set of 6-tuples (rotation vector + translation vector) for each view
extrinsic_parameters: !!opencv-matrix
  rows: 13
  cols: 6
  dt: d
  data: [ 1.7329890898434247e-01, -1.0714616515515805e-01,
    -5.4122514579622270e-03, -3.3065840233248860e+00,
    -2.3549979212628034e+00, 3.3121061958207576e+01,
    -9.6972764799164257e-01, -5.0806039133469227e-01,
    2.8769297518626540e+00, -5.3845832161689282e+00,
    -2.1323058138656972e+00, 3.0321536529987672e+01,
    -9.6529637310982508e-01, -4.7743370219153775e-01,
```

This file will be used in the next TP to reconstruct the matching point from 2D from 3D. We will use **Distortion Matrix** to undistort the image before doing detection and matching.

Then use **Camera matrix (K)** and **parameters extrinsic (R|t)** to obtain projection Matrix P, solve the equations $X = Px$ and $X = P'x'$ to get X from 3D

3.4 . Compute Reprojection Errors and Undistort images



The average projection error calculated in the terminal: 1.69

For distortion, as you can see in the Image view, the image after disorting won't be a Rectangle anymore, the border of width and height is now a curve after correct distortion.

Difficulty (problèmes rencontrés):

The quality of my camera is really bad so the result is not really stable.

As I said before, *similar images result in similar equations, and similar equations at the calibration step will form an ill-posed problem, so the calibration will fail*, to get a good result, we need to take at least a number of photo from camera (better > 10). And I have to move around to get different point of view camera.

END