Camera Calibration Using OpenCV

## What is this?

Primarily, finding the quantities internal to the camera that affect the imaging process

* Position of image center in the image

// It is typically not at (width/2, height/2) of image

* Focal length
* Different scaling factors for row pixels and column pixels
* Lens distortion (pin-cushion effect) //Never Mind, we will come to that

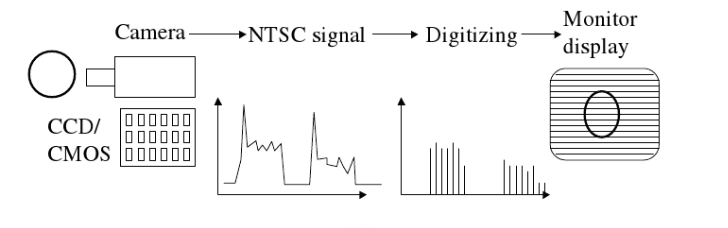
## Why do we do it?

• Camera pixels are not necessarily square

• Camera output may be analog (NTSC)

• Image may be obtained by digitizing card

(A/D converter samples NTSC signal)



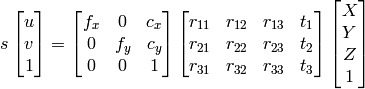
So Good calibration is important when we need to

* Reconstruct a world model: Virtual L.A. project
* Interact with the world
* Robot, hand-eye coordination

**So we need to relate the image(2-D) in the eye of the camera with the actual world (3-D) what we can visualize by our eye by some mean.**

## How To Relate? Matrix!

The functions in this section use the so-called **pinhole camera model.** That is, a scene view is formed by projecting 3D points into the image plane using a perspective transformation.



* Where $(X, Y, Z)$ are the coordinates of a 3D point in the world coordinate space
* $(u, v)$ are the coordinates of the projection point in pixels(2D)
* $(cx, cy)$ is a principal point (that is usually at the image center),
* $fx, fy$ are the focal lengths expressed in pixel-related units.

# Camera Matrix

The matrix containing fx,fy .. is called  **camera matrix, or a matrix of intrinsic parameters**. And once we get those parameters somehow, that will be fixed for different views(basically different position of camera for same image or plane) if we don’t use a zoom camera which is our case. :-)

If an image is scaled by some factor then these parameters also change accordingly.

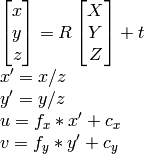
## Example

Image 1=[1000\*500] fx=200,fy=300,cx=100,cy=500

Image2=[500\*250] then the parameters will change and now will be 100,150,50,250 resp.

# Matrix Of Extrinsic Parameters

* The next matrix with all r\*\*s and t\*s create the *matrix of extrinsic parameters*.
* R stands for rotation and T for translation. The notation is **[R|t].**
* It is used to describe the camera motion around a static scene, or vice versa, rigid motion of an object in front of still camera .
* That is, $[R|t]$ translates coordinates of a point(3D) $(X, Y, Z)$ to some coordinate system(2D system), fixed with respect to the camera. The transformation above is equivalent to the following (when $z \ne 0$).

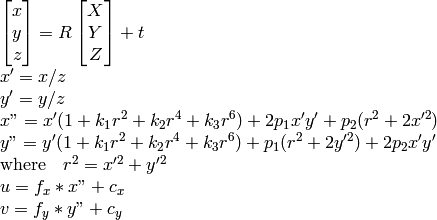


Here u and v are our point of interest meaning the corresponding 2D transformation.

R is the rotation Matrix and t is translational matrix.

Simple??

But ,in real life nothing is perfect. So of course we have some distortion.And then the relation becomes



* $k_1$, $k_2$, $k_3$ are radial distortion coefficients
* $p_1$, $p_2$ are tangential distortion coefficients
* Don’t worry, The distortion coefficients do not depend on the scene viewed, thus they also belong to the intrinsic camera parameters.. nd they remain the same regardless of the captured image resolution. That is ***NO SCALING .***

## Example:

Image1 =[480\*360] Image2=[240\*180]

Both have different fx,fy,cx,cy but the same k1,k2,k3 and p1,p2.

* 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.

**Now enough of Theory. Now lets think how to implement all the things we have learned!**

* Project 3D points to the image plane given intrinsic and extrinsic parameters
* Compute extrinsic parameters given intrinsic parameters, a few 3D points and their projections.
* Estimate intrinsic and extrinsic camera parameters from several views of a known calibration pattern (i.e. every view is described by several 3D-2D point correspodences).
* Estimate the relative position and orientation of the stereo camera “heads” and compute the rectification transformation that makes the camera optical axes parallel. //

## Fun1:When we have some 3D and 2D points and we want the intrinsic and extrinsic parameters

#### Syntax: double calibrateCamera(const vector<vector<Point3f> >& objectPoints,

#### Const vector<vector<Point2f>>& imagePoints, Size imageSize,  [Mat](http://opencv.willowgarage.com/documentation/cpp/core_basic_structures.html#Mat)&cameraMatrix,

#### [Mat](http://opencv.willowgarage.com/documentation/cpp/core_basic_structures.html#Mat)& distCoeffs, vector<[Mat](http://opencv.willowgarage.com/documentation/cpp/core_basic_structures.html#Mat)>& rvecs, vector<[Mat](http://opencv.willowgarage.com/documentation/cpp/core_basic_structures.html#Mat)>& tvecs, int flags=0)

#### Parameters(INPUT )

* **objectPoints**– **The vector of vectors of points** on the calibration pattern in its coordinate system, one vector per view. If the same calibration pattern is shown in each view and it’s fully visible then all the vectors will be the same, although it is possible to use partially occluded patterns, or even different patterns in different views - then the vectors will be different. The points are 3D, but since they are in the pattern coordinate system, then if the rig is planar, it may have sense to put the model to the XY coordinate plane, so that Z-coordinate of each input object point is 0
* ***imagePoints*** – The vector of vectors of the object point projections on the calibration pattern views, one vector per a view. The projections must be in the same order as the corresponding object points.
* **imageSize –** recquired only to nitialize the intrinsic camera matrix
* ***distCoeffs*** – The output 5x1 or 1x5 vector of distortion coefficients (k_1, k_2, p_1, p_2[, k_3])

### Parameters (Output):

* **rvecs:** The output vector of rotation vectors[we will come to that]
* **tvecs:** output translation vector //both of them are estimated in each pattern view

#### Different Flags:

* **CV\_CALIB\_USE\_INTRINSIC\_GUESS :**  cameraMatrix contains the valid initial values of fx, fy, cx, cy that are optimized further. Otherwise, (cx, cy) is initially set to the image center ( imageSize is used here), and focal distances are computed in some least-squares fashion.[forget it ].Then we can use FindExtrinsicCameraParams2  to directly calculate the extrinsic parameters.
* **CV\_CALIB\_FIX\_PRINCIPAL\_POINT** : The principal point is not changed during the global optimization, it stays at the center or at the other location specified when CV\_CALIB\_USE\_INTRINSIC\_GUESS is set too.
* **CV\_CALIB\_FIX\_ASPECT\_RATIO** : The ratio of fx to fy which won’t change in different vews as well as image sizes. When CV\_CALIB\_USE\_INTRINSIC\_GUESS is not set, the actual input values of fx and fy are ignored, only their ratio is computed and used further.
* **CV\_CALIB\_ZERO\_TANGENT\_DIST** : as the name says p1 and p2 which will be zero.

**We will use a chess-board to calibrate the camera . The algo is the following**.

1. First, it computes the initial intrinsic parameters (the option only available for planar calibration patterns) or reads them from the input parameters. The distortion coefficients are all set to zeros initially (unless some of CV\_CALIB\_FIX\_K? are specified).
2. The initial camera pose is estimated as if the intrinsic parameters have been already known. This is done using *FindExtrinsicCameraParams2*
3. After that the global Levenberg-Marquardt optimization algorithm is run to minimize the reprojection error, i.e. the total sum of squared distances between the observed feature points imagePoints and the projected (using the current estimates for camera parameters and the poses) object points objectPoints .

The function returns the final re-projection error. It is basically what it should be the projection of 3D into 2D in our visualization (loss of data) vs some algorithm to retain all that data by some slight deffernet projection.

## Fun2:When we have the camera matrix and we need some useful camera characteristic

#### Syntax: void calibrationMatrixValues(const [Mat](http://opencv.willowgarage.com/documentation/cpp/core_basic_structures.html#Mat)&  cameraMatrix, Size imageSize, double apertureWidth, double apertureHeight, double& fovx, double& fovy, double& focalLength, Point2d& principalPoint , double&  aspectRatio)

#### Parametrs :

apertureWidth- Physical width of the sensor

apertureHeight – Physical height of the sensor

* fovx – The output field of view in degrees along the horizontal sensor axis
* fovy – The output field of view in degrees along the vertical sensor axis
* aspectRatio – f_y/f_x

## FUN3: It Combines two rotation-and-shift

## Transformations.

Syntax: void  **composeRT**(const [Mat](http://opencv.willowgarage.com/documentation/cpp/core_basic_structures.html#Mat)&**rvec1**, const [Mat](http://opencv.willowgarage.com/documentation/cpp/core_basic_structures.html#Mat)& **tvec1**, const [Mat](http://opencv.willowgarage.com/documentation/cpp/core_basic_structures.html#Mat)&**rvec2**, const [Mat](http://opencv.willowgarage.com/documentation/cpp/core_basic_structures.html#Mat)&**tvec2**, [Mat](http://opencv.willowgarage.com/documentation/cpp/core_basic_structures.html#Mat)&**rvec3**, [Mat](http://opencv.willowgarage.com/documentation/cpp/core_basic_structures.html#Mat)&**tvec3**)

Where

* d??d?? – The optional output derivatives of rvec3 or tvec3 w.r.t. rvec? or tvec?
* \*vec# is the 1st/2nd/3rd rotational or translational vector

## FUN4:How To Find The Corners Of A Chess-Board?

**Syntax:** bool **findChessboardCorners**(const [Mat](http://opencv.willowgarage.com/documentation/cpp/core_basic_structures.html#Mat)&**image**,

Size **patternSize**, vector<Point2f>& **corners**,

Int **flags**=**CV\_CALIB\_CB\_ADAPTIVE\_THRESH**+ **CV\_CALIB\_CB\_NORMALIZE\_IMAGE**)

***Parameters:***

* *image* – Source chessboard view; it must be an 8-bit grayscale or color image
* *patternSize* – The number of inner corners per chessboard row and column ( patternSize = cvSize(points \_ per \_ row,points \_ per \_ colum) = cvSize(columns,rows) )
* *corners* – The output array of corners detected
* Various operation flags, can be 0 or a combination of the following values:
* **CV\_CALIB\_CB\_ADAPTIVE\_THRESH** use adaptive thresholding to convert the image to black and white, rather than a fixed threshold level (computed from the average image brightness).
* **CV\_CALIB\_CB\_NORMALIZE\_IMAGE** normalize the image gamma with *EqualizeHist* before applying fixed or adaptive thresholding.
* **CV\_CALIB\_CB\_FILTER\_QUADS** use additional criteria (like contour area, perimeter, square-like shape) to filter out false quads that are extracted at the contour retrieval stage.

**So now ,lets talk about how to calibrate the camera in steps .**

1. First we have to read the settings. After reading the file ,we have an additional post-process function that checks for the validity of the input. Only if all of them are good will be the ***goodInput*** variable true.
2. 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 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 toCALIBRATED one.]
3. The formation of equations depends on patterns of the input.e.g. the cornors in chessboard form the pattern.There after we will use the  [findChessboardCorners](http://opencv.itseez.com/trunk/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html#findchessboardcorners) function.
4. Then again in case of cameras we only take camera images after an input delay time passed. This is in order to allow for the user to move the chessboard around and as getting different images. Same images mean same equations, and same equations at the calibration will form an ill-posed problem, so the calibration will fail. For square images the position of the corners are only approximate. We may improve this by calling the [cornerSubPix](http://opencv.itseez.com/trunk/modules/imgproc/doc/feature_detection.html#cornersubpix) function. This way will get a 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.
5. **Show state and result for the user, plus command line control of the application**. The showing part consists of a text output on the live feed, and for video or camera input to show the “capturing” frame we simply bitwise negate the input image. If we only ran the calibration and got the camera matrix plus the distortion coefficients we may just as correct the image with the [undistort](http://opencv.itseez.com/trunk/modules/imgproc/doc/geometric_transformations.html#undistort) function:
6. Then we wait for an input key and if this is u we toggle the distortion removal, if it is g we start all over the detection process (or simply start it), and finally for the ESC key quit the application.
7. **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 append this after the loop. Taking advantage of this now I’ll expand the [undistort](http://opencv.itseez.com/trunk/modules/imgproc/doc/geometric_transformations.html#undistort) function, which is in fact first a call of the[initUndistortRectifyMap](http://opencv.itseez.com/trunk/modules/imgproc/doc/geometric_transformations.html#initundistortrectifymap) to find out the transformation matrices and then doing the transformation with the [remap](http://opencv.itseez.com/trunk/modules/imgproc/doc/geometric_transformations.html#remap) function. Because, after a successful calibration the map calculation needs to be done only once, by using this expanded form you may speed up your application:

## Using A Function For Calibration And Saving.

Because the calibration needs to be only once per camera it makes sense to save them 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.

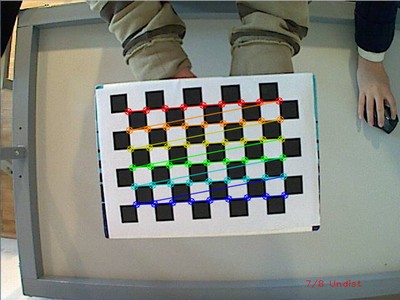
Therefore in the first function we just split up these two processes. Because we want to save many of the calibration variables we’ll create these variables here and pass on both of them to the calibration and saving function. Again, I’ll not show the saving part as that has little in common with the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

**We do the calibration with the help of the**[**calibrateCamera**](http://opencv.itseez.com/trunk/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html#calibratecamera)**function**.

**STEPS:**

1. 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
2. The image points. This is a vector of *Point2f* vector that for each input image contains where the important points (corners for chessboard, and center of circles for the circle patterns) were found. We already collected this from what the [findChessboardCorners](http://opencv.itseez.com/trunk/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html#findchessboardcorners) or the [findCirclesGrid](http://opencv.itseez.com/trunk/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html#findcirclesgrid) function returned. We just need to pass it on.
3. . If we used the fix aspect ratio option we need to set the f_x to zero.
4. The distortion coefficient matrix. Initialize with zero.
5. The function will calculate for all the views the rotation and translation vector that transform the object points (given in the model coordinate space) to the image points (given in the world coordinate space). The 7th and 8th parameters are an output vector of matrices containing in the ith position the rotation and translation vector for the ith object point to the ith image point.
6. The final argument is a 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.
7. The function returns the average re-projection error. This number gives a good estimation of just how exact is 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 [projectPoints](http://opencv.itseez.com/trunk/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html#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 calculate for all the calibration images.

So let this be the input :



We can clearly see the distortion. After applying the distortion removal we get:

