Pose estimation & camera movement tracking

Mašinska vizija, 2018.

Marija Janković

Back to camera calibration

- OpenCV calib3d module
- `cv::calibrateCamera()`
- In this routine, the method of calibration is to target the camera on a known structure that has many individual and identifiable points. By viewing this structure from a variety of angles, we can then compute the (relative) location and orientation of the camera at the time of each image as well as the intrinsic parameters of the camera
- To provide multiple views, we rotate and translate the object

cv::calibrateCamera()

```
double cv::calibrateCamera   (    InputArrayOfArrays      objectPoints,
                                  InputArrayOfArrays      imagePoints,
                                  Size                    imageSize,
                                  InputOutputArray        cameraMatrix,
                                  InputOutputArray        distCoeffs,
                                  OutputArrayOfArrays     rvecs,
                                  OutputArrayOfArrays     tvecs,
                                  OutputArray             stdDeviationsIntrinsics,
                                  OutputArray             stdDeviationsExtrinsics,
                                  OutputArray             perViewErrors,
                                  int                     flags = 0,
                                  TermCriteria            criteria =
TermCriteria(TermCriteria::COUNT+TermCriteria::EPS, 30, DBL_EPSILON)
)
```

cv::calibrateCamera()

- The algorithm performs the following steps:
  - Initialization
    - intrinsics (eye matrix or available as inputs)
    - distortion coeffs (zeros or available as inputs)
  - Estimate initial camera pose (extrinsics) using `cv::solvePnP()`
  - Run optimization algorithm to minimize the reprojection error, that is, the total sum of squared distances between the observed feature points (from the image) and the projected (using the current estimates for camera parameters and the poses) referent 3D object points.
- The function returns the final re-projection error.

We calibrated the camera, how will we use the results?

- With calibrated camera we have acquired important parameters:

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

$$x_{\text{corrected}}=x(1+k_1r^2+k_2r^4+k_3r^6)$$
$$y_{\text{corrected}}=y(1+k_1r^2+k_2r^4+k_3r^6)$$
$$x_{\text{corrected}}=x+[2p_1xy+p_2(r^2+2x^2)]$$
$$y_{\text{corrected}}=y+[p_1(r^2+2y^2)+2p_2xy]$$

  - intrinsics – focal length f and point of lens center on the imager (cx, cy)
  - extrinsics – rotation matrix R and translation matrix T
  - distortion coefficients – r1, r2, r3, p1, p2

- Perform camera calibration once again!!!
  - open camera_calibration_with_debug.cpp and set the path for in_VID5.xml and run it

Example 3 – look at the results

- Open out_camera_data
- What can we conclude from intrinsic matric?
  - cx, cy = ?
  - what did we expect?
  - focal length?
- How do the parameters change with resolution?
  - inputCapture.set(CV_CAP_PROP_FRAME_WIDTH, 1280);
  - inputCapture.set(CV_CAP_PROP_FRAME_HEIGHT, 720);
- How big are distortion coefficients?

Camera spec

- The C270 Logitech web camera has a sensor with pixels size of 2.8um, and a resolution of 1280x720,
  so the sensor size is 3.58 mm x 2.02 mm
- http://support.logitech.com/en_us/article/17556

Extrinsics vs rotation matrix vs rvec &tvec

- camera calibration function provides two 3element vectors `rvec` and `tvec`
- we expected a 4x4 extrinsic matrix with rotation matrix `R` 3x3, and translation vector `T` 3x1
- `T` is the same as `tvec`, but `R` is not the same as `rvec`
- What is the catch?

Rodrigues

- Defining the three angles of rotation does not uniquely define the rotation since the order of rotations changes the final result.
- Thus the rotation matrix R is not intuitive and it is complicated to understand which rotations will be applied
- Rodriques define a new way of defining a 3D rotation using rotation vector $\vec{r} = [r_x \quad r_y \quad r_z]$
  - r is a vector around which the system should be rotated for a single angle to achieve the wanted rotation
  - angle theta is defined through the norm of the r vector $\theta \leftarrow \mathrm{norm}(r)$
- A rotation vector is a convenient and most compact representation of a rotation matrix (since any rotation matrix has just 3 degrees of freedom).

cv::Rodrigues()

- We can use `cv::Rodrigues` to transform from camera vector to camera matrix or vice versa

$$\theta \leftarrow \mathrm{norm}(r)$$
$$r \leftarrow r/\theta$$
$$R = \cos\theta I + (1-\cos\theta) r r^T + \sin\theta \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ -r_y & r_x & 0 \end{bmatrix}$$

Inverse transformation can be also done easily, since

$$\sin(\theta) \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ -r_y & r_x & 0 \end{bmatrix} = \frac{R - R^T}{2}$$

We calibrated the camera, what’s next?

- We want to provide that the straight lines in the real world are also straight in the image
- Image should be undistorted using the acquired distortion coefficients

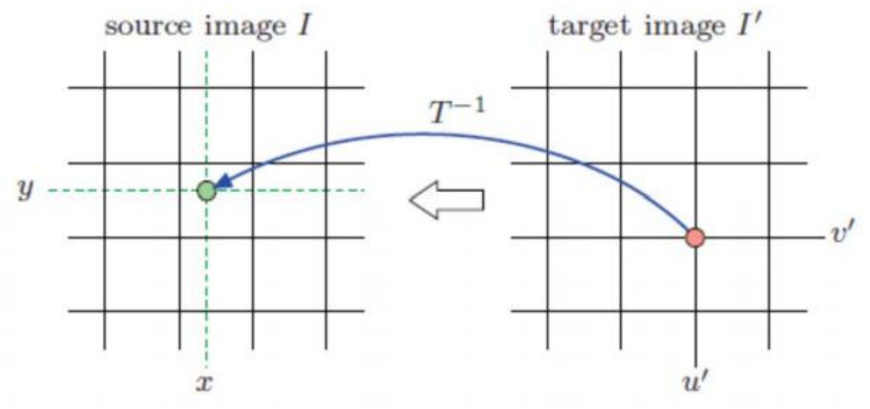
cv::undistort()

- transforms image to compensate radial and tangential lens distortion
- the function is simply a combination of `cv::initUndistortRectifyMap` and `cv::remap` (with bilinear interpolation)
- Those pixels in the destination image, for which there is no correspondent pixels in the source image, are filled with zeros (black)
- A particular subset of the source image that will be visible in the corrected image can be regulated by newCameraMatrix. Using `cv::getOptimalNewCameraMatrix` compute the appropriate newCameraMatrix depending on your requirements.

initUndistortRectifyMap()

- Takes calculated camera matrix, dist coeffs, rvec and tvecs and calculates the mapping between the wanted undistorted pixel and original distorted pixel.
- This is necessary for backward warping of the image.
- Beside the camera matrix, we provide new camera matrix we want if we changed the alpha in cv::**getOptimalNewCameraMatrix**
- Generated maps are then input for remap function which performs the actual warping

cv::initUndistortRectifyMap – backwards warping



$$x' \leftarrow (u - c_x)/f_x$$
$$y' \leftarrow (v - c_y)/f_y$$
$$r^2 \leftarrow x'^2 + y'^2$$

$$x'' \leftarrow x' \frac{1+k_1 r^2+k_2 r^4+k_3 r^6}{1+k_4 r^2+k_5 r^4+k_6 r^6} + 2p_1 x'y' + p_2(r^2 + 2x'^2) + s_1 r^2 + s_2 r^4$$
$$y'' \leftarrow y' \frac{1+k_1 r^2+k_2 r^4+k_3 r^6}{1+k_4 r^2+k_5 r^4+k_6 r^6} + p_1(r^2 + 2y'^2) + 2p_2 x'y' + s_3 r^2 + s_4 r^4$$

$$s \begin{bmatrix} x''' \\ y''' \\ z''' \end{bmatrix} = \begin{bmatrix} x'' \\ y'' \\ z'' \end{bmatrix}$$
$$map_x(u,v) \leftarrow x'''/f_x + c_x$$
$$map_y(u,v) \leftarrow y'''/f_y + c_y$$

cv::getOptimalNewCameraMatrix

- The function computes and returns the optimal new camera matrix based on the free scaling parameter.
- By varying this parameter, you may retrieve only sensible pixels alpha=0 , keep all the original image pixels if there is valuable information in the corners alpha=1 , or get something in between.
- When alpha>0 , the undistortion result is likely to have some black pixels corresponding to "virtual" pixels outside of the captured distorted image.
- The original camera matrix, distortion coefficients, the computed new camera matrix, and newImageSize should be passed to initUndistortRectifyMap to produce the maps for remap .

What’s next?

- With calibrated camera we have acquired important information and parameters:
  - intrinsics – focal length f and point of lens center on the imager (cx, cy)
  - extrinsics – rotation matrix R and translation matrix T
  - distortion coefficients
- Knowing the transformation between real world coordinates and pixels in the image we can:
  - extract information on pose or distance of real world objects
  - track object movements
  - recreate camera movement
  - render 3D structure from camera motion

Projections

- Once we have calibrated the camera, it is possible to unambiguously project points in the physical world to points in the image.
- This means that, given a location in the three-dimensional physical coordinate frame attached to the camera, we can compute where on the imager, in pixel coordinates, an external three-dimensional point should appear.

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}}_{\text{FIXED}} \underbrace{\begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix}}_{\text{Calculated for each frame again}} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Projections

- This transformation is accomplished by the OpenCV routine cv::projectPoints()

```
void cv::projectPoints(
        cv::InputArray objectPoints,         //vector<Point3f>
        cv::InputArray rvec,                 // Rotation *vector*
        cv::InputArray tvec,                 // Translation vector
        cv::InputArray cameraMatrix,         // 3x3 Camera intrinsics matrix
        cv::InputArray distCoeffs,           // 4, 5, or 8 elements vector,
        cv::OutputArray imagePoints,         // or vector<Point2f>
        cv::OutputArray jacobian = cv::noArray(), // Optional,
        double aspectRatio = 0               // If nonzero, fix
);
```

Three-Dimensional Pose Estimation

- single camera
  - It is possible to compute the pose of a known object with only one camera.
  - In addition to being a useful technique in its own right, understanding the single-camera pose estimation problem gives important insights into the multiple camera problem.
- multiple cameras
  - In the multicamera case, we use correspondences between what is seen from each of the separate cameras to draw conclusions about where the object is (i.e., by triangulation).
  - The advantage of such a technique is that it will work with even unknown objects or entire unknown scenes.
  - The disadvantage is that it requires multiple cameras.

Pose Estimation from a Single Camera

- An object is “known” to the extent that we have identified some number of keypoints on the object, whose location we know in the coordinate system of the object.
- Now if we are presented with the same object in a novel pose, we can look for those same keypoints.
- If we now want to figure out the relationship between the pose of the object and the camera, the essential observation is that for each point that we find, that point must lie on a particular ray emanating from a pixel location on the camera’s imager out through the aperture of the camera.
- Of course, individually we cannot know the distance from the camera to a particular point, but given many such constraints, a rigid object will only be able to meet all of those constraints one way.

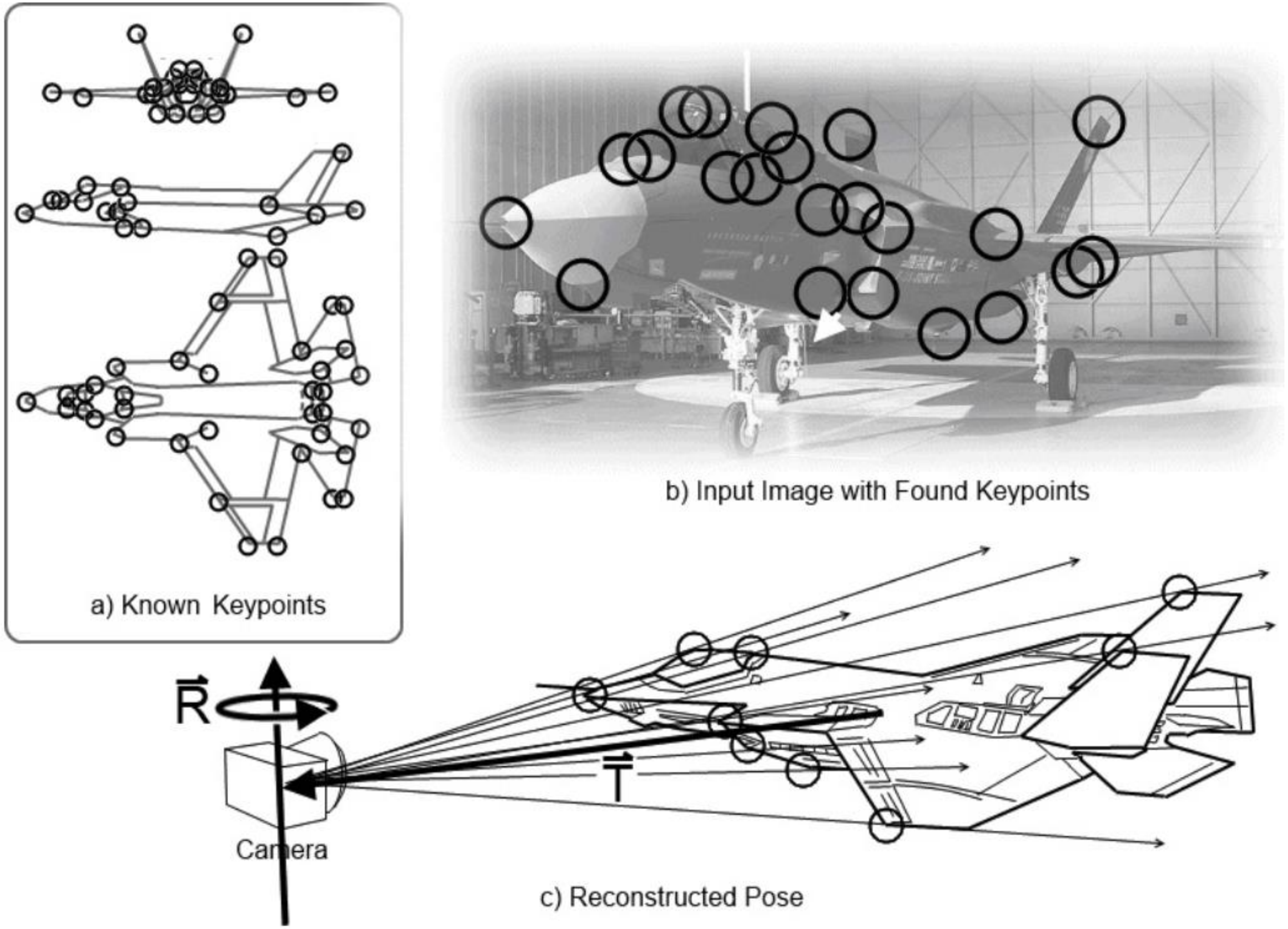


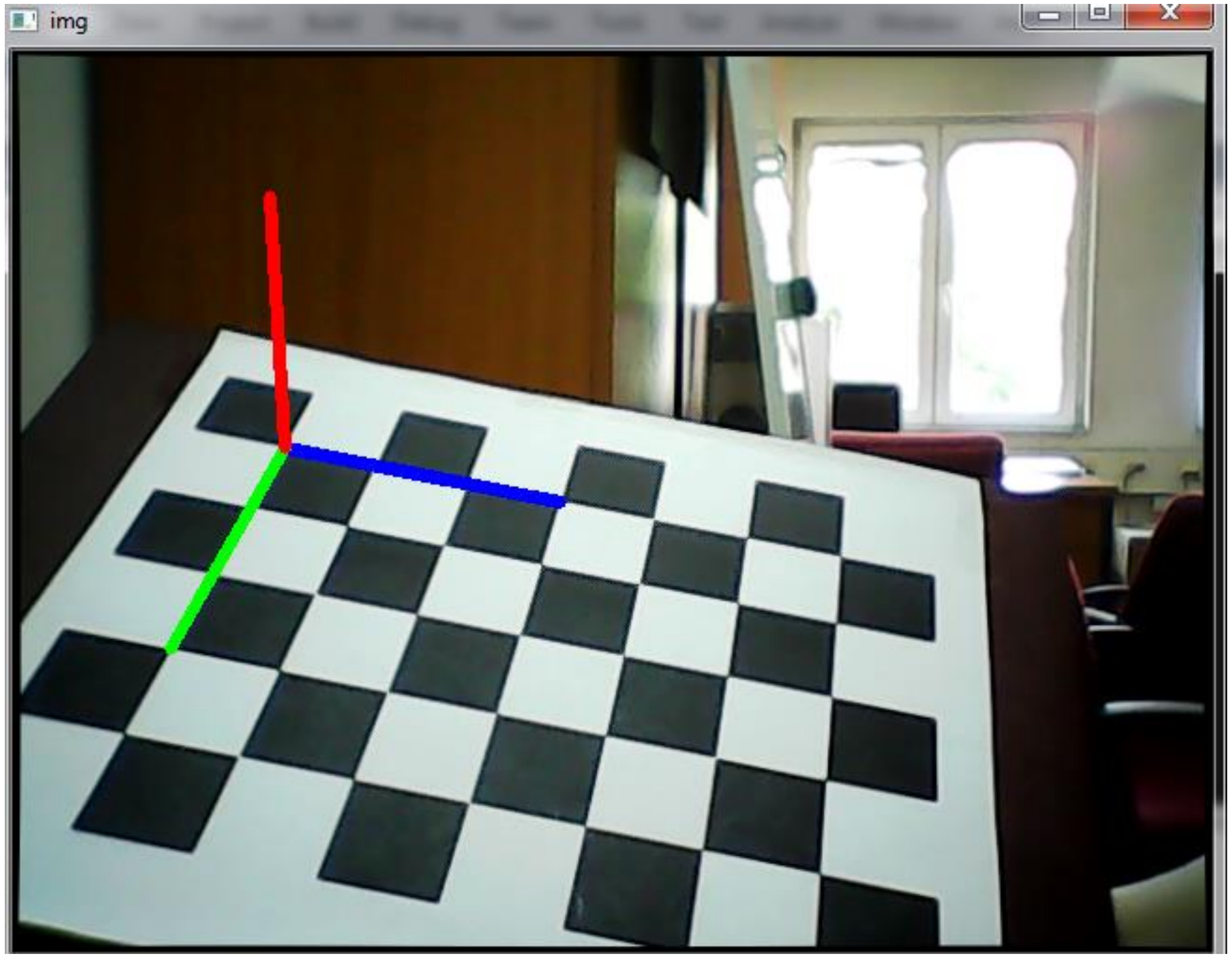
Figure 19-4. Given a known set of keypoints (a), which can be found in an image of the same object (b), it is possible to reconstruct the pose of the object relative to the camera (c)

Perspective N-Point (PNP) problem

- We know the relative positions of features in 3D world and, after detection, we have corresponding 2D pixel points of the subset of features. What’s next?
- This is a PNP problem, which is easily solved with `cv::solvePNP()`
- solvePNP provides perspective projection – R matrix and T matrix, knowing intrinsic and distortion parameters
- The perspective projection matrix, along with the intrinsics and distortion params enables us to project any 3D point to our image!
- Note that the PNP problem does not always have a unique solution. There are two important cases in which PNP cannot provide reliable results
  - when you just don’t have enough points
  - when the object is very far away.

Example – pose of planar object - chessboard

- We want to project 3D axis onto our chessboard chart in order to visually display the rotation of the chessboard plane.
- How do we approach this problem?

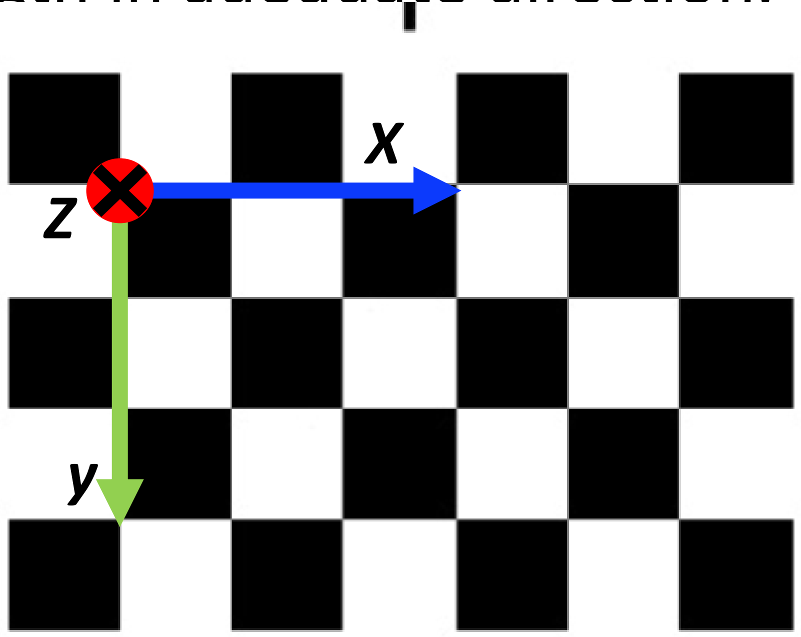


Example – pose of planar object - chessboard

- We have object we known 3D points and we can detect its 2D image points – corners of the chessboard
- Can we calculate something from this correspondence?
- Which camera parameters do we need?
  - intrinsics – In OpenCV this is called Camera Matrix
  - distortion coeffs
  - extrinsics
- Which parameters change with chessboard movement?

Example – pose of planar object - chessboard

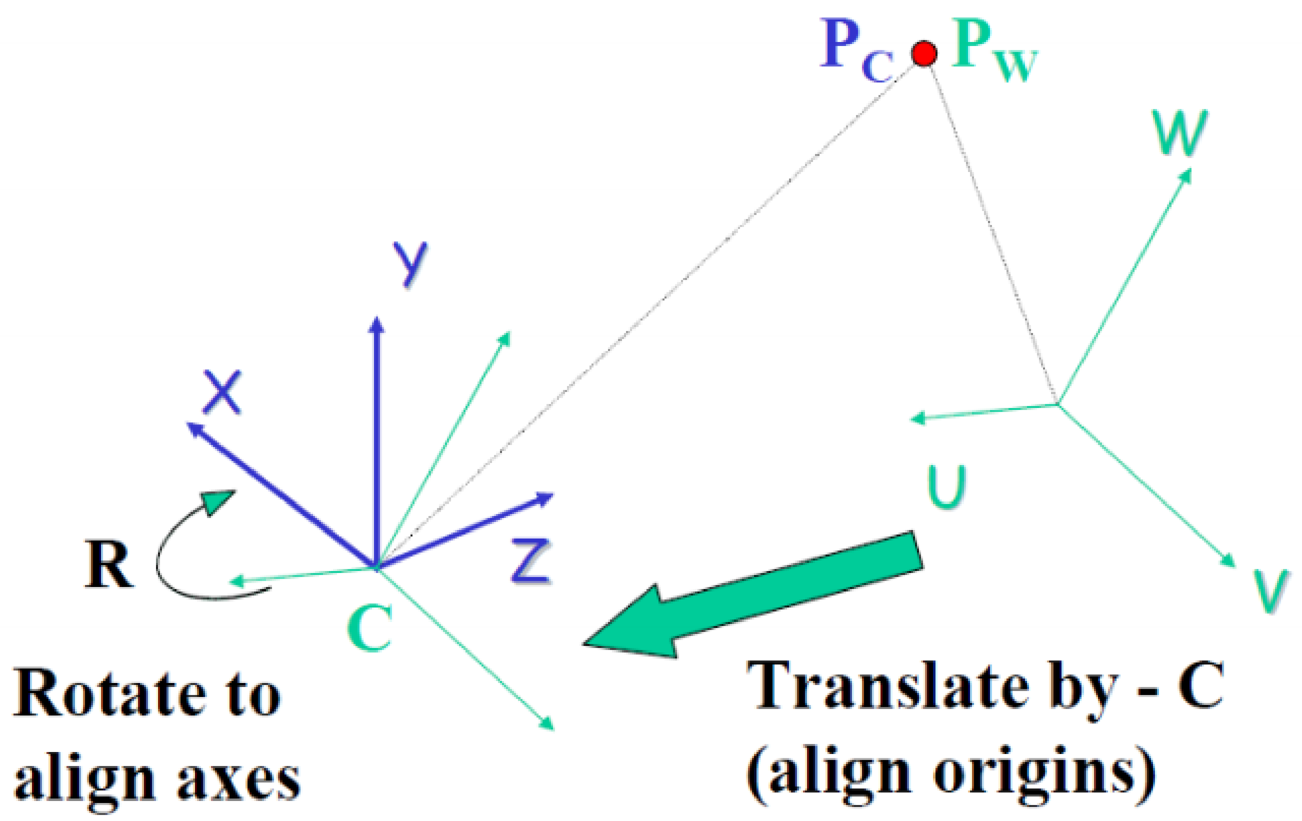
- Add XYZ axis to the chessboard
- In 3D world we will imagine axis like lines going from the first corner to third corner on x axis, from first to third on y axis and for z axis a line of the same length in adequate direction.



Example – pose of planar object - chessboard

- So the solution is simple, we only need to
  - find the perspective projection from 3D world to our 2D imager
  - and project the axis points to 2D
- Open file *poseEstimationSingleCamera_template.cpp*
- Code already reads out the intrinsics and distortion coeffs
- input params are provided through command line arguments
  - size of the calibration chart (14x9 corners)
  - file with calibration results (out_camera_data.xml)
  - camera ID (0 if it is the only camera, if there are several choose one)
  - *example of command line args : 14 9 out_camera_data.xml 0*

Retrieving camera movement



$$P_C = R\,(P_W - C)$$

Camera system vs world system

- We want to calculate the coordinates of the camera origin in the world coordinate system
- Which parameters will provide this connection?
- extrinsics

$$\begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = \underbrace{\begin{pmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 1 & -c_z \\ 0 & 0 & 0 & 1 \end{pmatrix}} \begin{pmatrix} U \\ V \\ W \\ 1 \end{pmatrix}$$

$$\begin{aligned} &\mathbf{R}\,(\mathbf{P_W} - \mathbf{C}) \\ &= \mathbf{R}\,\mathbf{P_W} - \mathbf{R}\,\mathbf{C} \\ &= \mathbf{R}\,\mathbf{P_W} + \mathbf{T} \end{aligned} \qquad \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

We want it the other way around

- Extrinsics provide transformation from world system to camera system
- We want the origin of the camera system given in the world coordinates, how will we achieve that?

Example – camera position estimation

- open *cameraTrajectoryEstimation_template.cpp*
- Code already reads out the intrinsics and distortion coeffs
- input params are provided through command line arguments
  - size of the calibration chart (14x9 corners)
  - file with calibration results (out_camera_data.xml)
  - camera ID (0 if it is the only camera, if there are several choose one)
  - *example of command line args : 14 9 out_camera_data.xml 0*

References

[1] http://docs.opencv.org/master/– OpenCV 3.2.0-dev Documentation

[2] Adrian Kaehler and Gary Bradski. 2016. *Learning OpenCV 3: Computer Vision in C++ with the OpenCV Library* (1st ed.). O'Reilly Media, Inc..

[3] Kenneth Dawson-Howe. 2014. *A Practical Introduction to Computer Vision with OpenCV* (1st ed.). Wiley Publishing.

[4] Zhang, Z. "A Flexible New Technique for Camera Calibration." *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Vol. 22, No. 11, 2000, pp. 1330–1334.

[5] https://www.mathworks.com/help/vision/ug/camera-calibration.html

[6] http://tnt.etf.bg.ac.rs/~oe4dos/Predavanja/OE4DOS%20-%20Poboljsanje%20kvaliteta%20slike%20-%20prostorne%20operacije.pdf