

Pinhole camera, distortion, projection

Visics, ESAT

KU Leuven

1 Theory

In pinhole camera model, a scene view is formed by projecting 3D points into the image plane using a perspective transformation:

$$sp = K[R|t]P' \quad (1)$$

or

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \quad (2)$$

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;
- K is a camera matrix, or a matrix of intrinsic parameters;
- (c_x, c_y) is a principal point that is usually at the image center;
- f_x, f_y are the focal lengths expressed in pixel units.

Thus, if an image from the camera is scaled by a factor, all of these parameters should be scaled by the same factor. The matrix of intrinsic parameters does not depend on the scene viewed. So, once estimated, it can be re-used as long as the focal length is fixed. The joint rotation-translation matrix $[R|t]$ is called a matrix of extrinsic parameters. It is used to describe the camera motion around a static scene, or vice versa, rigid motion of an object in front of a still camera. That is, $[R|t]$ translates coordinates of a point (X, Y, Z) to a coordinate system, fixed with respect to the camera. The transformation above is equivalent to the following (when $z \neq 0$):

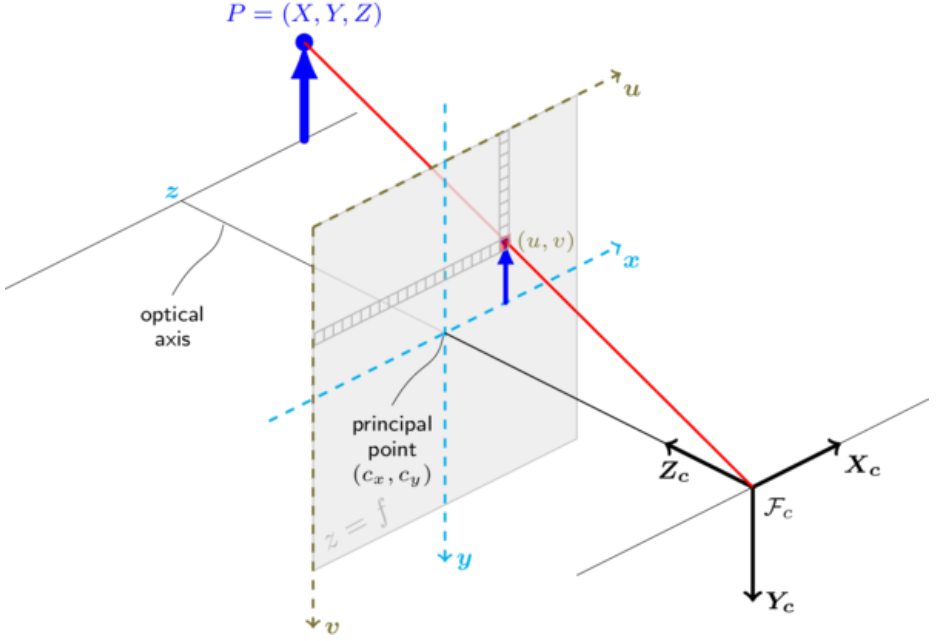
$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t \quad (3)$$

$$x' = x/z \quad (4)$$

$$y' = y/z \quad (5)$$

$$u = f_x * x' + c_x \quad (6)$$

$$v = f_y * y' + c_y \quad (7)$$



The following figure illustrates the pinhole camera model.

Today's cheap pinhole cameras introduces a lot of distortion to images. Two major distortions are radial distortion and tangential distortion. Due to radial distortion, straight lines will appear curved. Its effect is more as we move away from the center of image. Tangential distortion occurs because image taking lens is not aligned perfectly parallel to the imaging plane and causes some areas in image look nearer than expected. So, the above model is extended as:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t \quad (8)$$

$$x' = x/z \quad (9)$$

$$y' = y/z \quad (10)$$

$$x'' = x' \frac{1 + k_1 r^2 + k_2 r^4 + k_3 r^6}{1 + k_4 r^2 + k_5 r^4 + k_6 r^6} + 2p_1 x' y' + p_2 (r^2 + 2x'^2) \quad (11)$$

$$y'' = y' \frac{1 + k_1 r^2 + k_2 r^4 + k_3 r^6}{1 + k_4 r^2 + k_5 r^4 + k_6 r^6} + p_1 (r^2 + 2y'^2) + 2p_2 x' y' \quad (12)$$

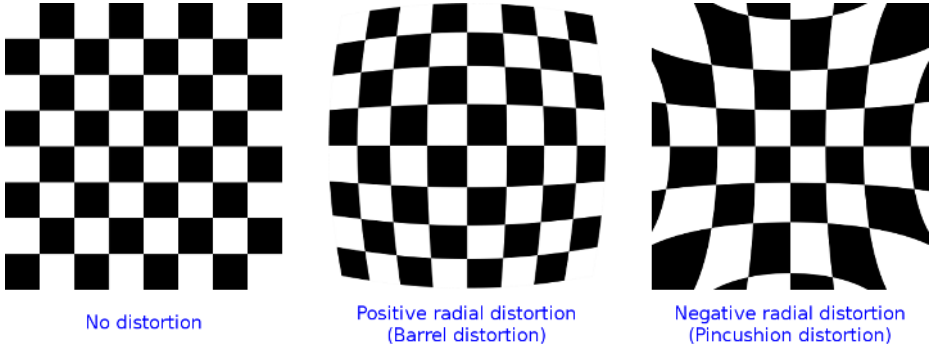
$$\text{where } r^2 = x'^2 + y'^2 \quad (13)$$

$$u = f_x * x'' + c_x \quad (14)$$

$$v = f_y * y'' + c_y \quad (15)$$

k_1, k_2, k_3, k_4, k_5 , and k_6 are radial distortion coefficients. p_1 and p_2 are tangential distortion coefficients.

The next figure shows two common types of radial distortion: barrel distortion (typically $k_1 > 0$) and pincushion distortion (typically $k_1 < 0$).



The distortion coefficients do not depend on the scene viewed. Thus, they also belong to the intrinsic camera parameters. And they remain the same regardless of the captured image resolution. If, for example, a camera has been calibrated on images of 320 x 240 resolution, absolutely the same distortion coefficients can be used for 640 x 480 images from the same camera while f_x, f_y, c_x , and c_y need to be scaled appropriately.

In this exercise we will do the following:

- Backproject 2D image into 3D space using depth map from RGBD camera;
- Generate a novel view by applying geometrical transformations to the obtained point clouds and projecting them back into the image;
- Find intrinsic and distortion parameters of the camera via calibration;
- Undistort an image taken by large-FOV camera.

2 Projection

Launch the jupyter notebook: jupyter notebook projection.ipynb. Individual cells of notebook can be run by pressing Shift+Enter.

Unzip the contents.zip and set up the path to "3D" directory in your notebook. Check that the images and the corresponding depth maps are loaded correctly.

2.1 2.5D to 3D

1. Given f_x, f_y, c_x, c_y , create the intrinsics matrix K ;

2. Generate a set of points p in pixel coordinates:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & \dots & w & \dots & w \\ 0 & 1 & 2 & 3 & \dots & 1 & \dots & h \\ 1 & 1 & 1 & 1 & \dots & 1 & \dots & 1 \end{bmatrix}$$

Use `np.linspace`, `np.meshgrid`, `np.transpose`, `np.reshape`. These set of operations is the same for all image frames. Starting from the next step, all steps should be performed separately for different frames;

3. Flatten the depth maps Z (use `np.reshape(..., (h*w))`);
4. Backproject these points to 3D: $P = Z * K^{-1}p$. Inverse can be computed with `np.linalg.inv`;
5. Add color information to these 3D points (you can flatten RGB images in a way similar to depth maps, and concatenate it to 3D point coordinates using `np.concatenate`).

If the backprojection is implemented correctly, you should be able to see some nice point clouds and navigate through them.

2.2 3D to 2D

1. Apply geometric transformation to obtained point clouds: $P' = M * P$, where

$$M \text{ is a } 3 \times 4 \text{ matrix, and } P = \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}. M \text{ can represent a simple translation}$$

of 1 meter in Z axis.

2. Project the transformed point cloud in the pixel coordinates, but keep Z (we

$$\text{will need it): } \begin{bmatrix} x \\ y \\ z \end{bmatrix} = K \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}, \begin{bmatrix} u \\ v \\ z \end{bmatrix} = \begin{bmatrix} x/z \\ y/z \\ z \end{bmatrix}$$

3. Now that the pixel coordinates in the new view are known, you can use color information (which does not change in 3D) to generate the images, which would look so as the photo was taken from this view. However, some points might be reprojected out of the image, have non-positive Z , or be occluded. To account for this, iterate over all points, and before assigning the corresponding color to the corresponding image pixel make sure that the aforementioned conditions hold. The point is considered to be occluded if there exists another point that projects to the same pixel, but has smaller positive depth Z .
4. Check what happens if you apply different transformations to the point clouds;
5. What do you notice in the generated images?

3 Distortion

Launch the jupyter notebook: `jupyter notebook undistortion.ipynb`.

3.1 Camera Calibration

Assuming already unzipped contents, set up the paths for calibration and distorted test images. Then, calibrate the camera to find intrinsic and distortion parameters. This is done by finding the correspondences between the set of 2D and 3D points:

1. Load photos of 9×6 calibration pattern taken from different views;
2. Find the corners of the checkerboard using `cv2.findChessboardCorners`;
3. Refine them with `cv2.cornerSubPix`;
4. Check if the corners are found correctly;
5. Find the camera parameters using `cv2.calibrateCamera`.

Why is a checkerboard pattern typically used for camera calibration?

What kind of distortion do you observe?

3.2 Undistortion

Now that the camera parameters are known, you can compute the undistortive mapping. In this exercise we consider k_4 , k_5 and k_6 to be 0. Thus:

$$x_{corrected} = x(1 + k_1r^2 + k_2r^4 + k_3r^6) + [2p_1xy + p_2(r^2 + 2x^2)]$$

$$y_{corrected} = y(1 + k_1r^2 + k_2r^4 + k_3r^6) + [p_1(r^2 + 2y^2) + 2p_2xy]$$

Such a correction is applied not in the pixel coordinates. The complete mapping looks as follows:

$$(u, v) \rightarrow (x, y) \rightarrow (x_{corrected}, y_{corrected}) \rightarrow (u_{corrected}, v_{corrected})$$

After computing the mapping, undistort the test image using `cv2.remap`. Compare this result to the undistortion implemented by OpenCV.