Pinhole camera

PSI-Visics, ESAT

KU Leuven

1 Theoretical background

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

$$sp = K[R|t]P' \tag{1}$$

or

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

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, the intrinsic 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 the 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 $z \neq 0$):

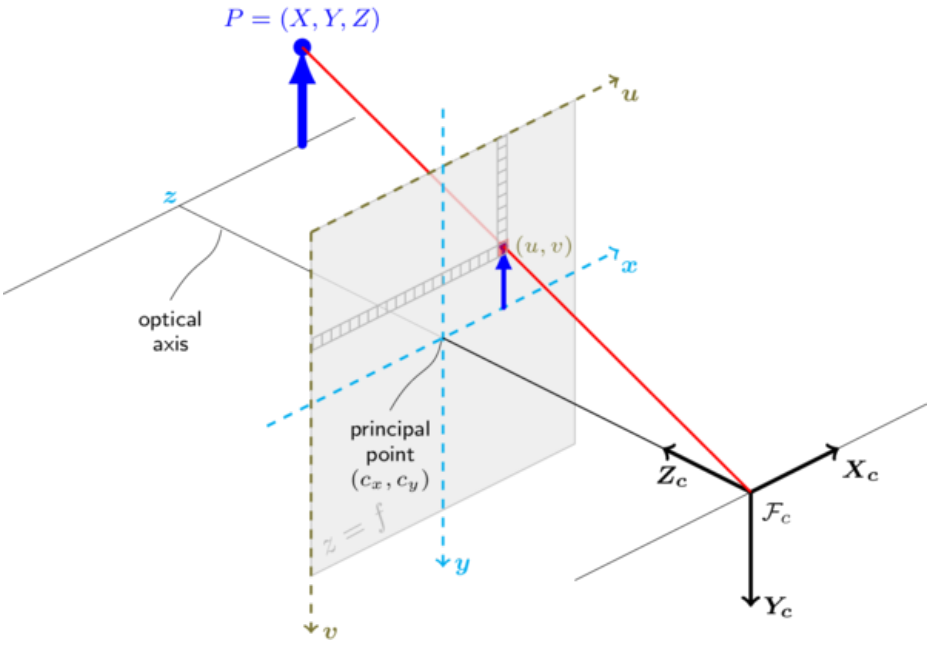
$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t \tag{3}$$

$$x' = x/z \tag{4}$$

$$y' = y/z \tag{5}$$

$$u = f_x * x' + c_x \tag{6}$$

$$v = f_y * y' + c_y \tag{7}$$



The following figure illustrates the pinhole camera model.

Today's cheap pinhole cameras introduce a lot of distortion to images. Two major distortion types are radial and tangential. Due to the radial distortion, straight lines will appear curved. Its effect is more noticeable as we move away from the image center. The tangential distortion occurs because the image taking lens is not aligned perfectly parallel to the image plane and causes some image areas to look closer than expected. Taking into account the distortion, the above model can be extended as:

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

$$x' = x/z \tag{9}$$

$$y' = y/z \tag{10}$$

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

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

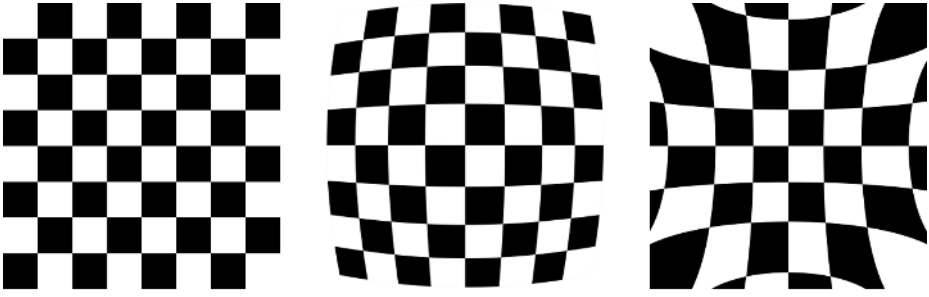
$$\text{where} \quad r^2 = x'^2 + y'^2 \tag{13}$$

$$u = f_x * x'' + c_x \tag{14}$$

$$v = f_y * y'' + c_y \tag{15}$$

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

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

2 Calibration and Distortion

Assuming the exercise materials are already extracted, activate your environment. E.g., run

conda activate < *environment_name* >

in the terminal. Then, launch the jupyter notebook:

jupyter notebook undistortion.ipynb

Look for "todo:" in the code and complete the missing parts cell by cell. Individual cells of the notebook can be run by pressing Shift+Enter.

2.1 Camera Calibration

Calibrate the camera to find the intrinsic and distortion parameters. This is done by finding the correspondences between the set of 2D and 3D points and solving the Eq. 8 – 15:

1. Set up the paths for the directory with the calibration images **calib_path** and the distorted test image(s) **test_img_path** (use "\\" instead of "\" for Windows paths).
   Load the photos of a 9×6 calibration pattern taken from different views;
2. Find the corners of the checkerboard using **cv2.findChessboardCorners**. Refine them with **cv2.cornerSubPix**;
3. Check if the corners are found correctly;
4. Find the camera parameters using **cv2.calibrateCamera**.

Why is a checkerboard pattern typically used for camera calibration?
What kind of distortion do you observe?

2.2 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:
$x_{corrected} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + [2p_1 xy + p_2(r^2 + 2x^2)]$
$y_{corrected} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + [p_1(r^2 + 2y^2) + 2p_2 xy]$

Such a correction is performed using the normalized coordinates, not the pixel coordinates. The complete mapping looks as follows:
$(u, v) \rightarrow (x, y) \rightarrow (x_{corrected}, y_{corrected}) \rightarrow (u_{corrected}, v_{corrected})$

5. Follow the instructions in "todos" and obtain the undistortive mapping;
6. After computing the mapping, undistort the test image using **cv2.remap**. Compare this result to the undistortion implemented by OpenCV.

Is the undistortion implemented correctly?

3 Projection

Launch the jupyter notebook:
jupyter notebook projection.ipynb
Look for "todo:" in the code and complete the missing parts cell by cell. Individual cells of the notebook can be run by pressing Shift+Enter.

3.1 2.5D to 3D

1. Load the libs, nothing to do here;
2. Set up the path to "3D" directory **root_dir** in your notebook.
   Load the RGB and the corresponding depth images;
3. Check that the images and the corresponding depth maps are loaded correctly;
4. Given f_x, f_y, c_x, c_y, create the intrinsics matrix $\mathbf{K}$ and its inverse **K_inv**;
5. (a) Generate a set of points p in homogeneous pixel coordinates:

   $$\begin{bmatrix} 0\,0\,0\,0\,... & w-1 & ... & w-1 \\ 0\,1\,2\,3\,... & 0 & ... & h-1 \\ 1\,1\,1\,1\,... & 1 & ... & 1 \end{bmatrix}$$

   Use **np.linspace**, **np.meshgrid**, **np.transpose**, **np.reshape**;
   (b) Compute $P_{imgplane} = K^{-1}p$. (a) and (b) are the same for all images. Starting from (c), all steps should be performed separately for different images ;
   (c) Flatten the depth maps Z (use np.reshape(..., (h*w))). Backproject the points to 3D by scaling them by Z: $P = Z * P_{imgplane}$;
   (d) 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**);
6. If the backprojection is implemented correctly, you should be able to see some nice point clouds and navigate through them.

3.2 3D to 2D

7. (a) Create a 4×4 3D transformation matrix M. M can represent a simple translation of 1 meter in Z axis;
   (b) Apply the 3D transformation M to the obtained point clouds: $P' = MP$, where , and $P = \begin{bmatrix} X_1 & ... & X_n \\ Y_1 & ... & Y_n \\ Z_1 & ... & Z_n \\ 1 & ... & 1 \end{bmatrix}$
   (c) Compute the pixel coordinates for the transformed point cloud, but keep Z (we will need it later): $\begin{bmatrix} x \\ y \\ z \end{bmatrix} = K \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$, $\begin{bmatrix} u \\ v \\ z \end{bmatrix} = \begin{bmatrix} x/z \\ y/z \\ z \end{bmatrix}$
8. Now the pixel coordinates in the new view are known, and 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 none of the aforementioned conditions holds. The point is considered to be occluded if there exists another point that projects to the same pixel, but has smaller

positive depth Z.

9. Visualize the images. What do you notice in the generated images?

Check what happens if you apply different transformations to the point clouds.

4 References

1. OpenCV calibration: `https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_calib3d/py_calibration/py_calibration.html`
2. NumPy arrays: `https://numpy.org/doc/stable/reference/arrays.html`