Lab 3: Camera calibration

MCHA4400

Semester 2 2023

Introduction

In this lab, you will implement a planar camera model and perform camera calibration using the OpenCV API. You will process image frames taken from a GoPro camera, calibrate the camera, and then test the calibration with a basic augmented reality demo.

The lab should be completed within 4 hours. The assessment will be done in the lab at the end of your enrolled lab session(s). Once you complete the tasks, call the lab demonstrator to start your assessment.

The lab is worth 5% of your course grade and is graded from 0–5 marks.

LLM Tip

You are strongly encouraged to explore the use of Large Language Models (LLMs), such as GPT, PaLM or LLaMa, to assist in completing this activity. Bots with some of these LLMs are available to use via the UoN Mechatronics Slack team, which you can find a link to from Canvas. If you are are unsure how to make the best use of these tools, or are not getting good results, please ask your lab demonstrator for advice.

1 Camera calibration (3 marks)

The calibration grid shown in Figure 1 was used for the photos taken by the GoPro. Each grid cell has a spacing of 22 mm.

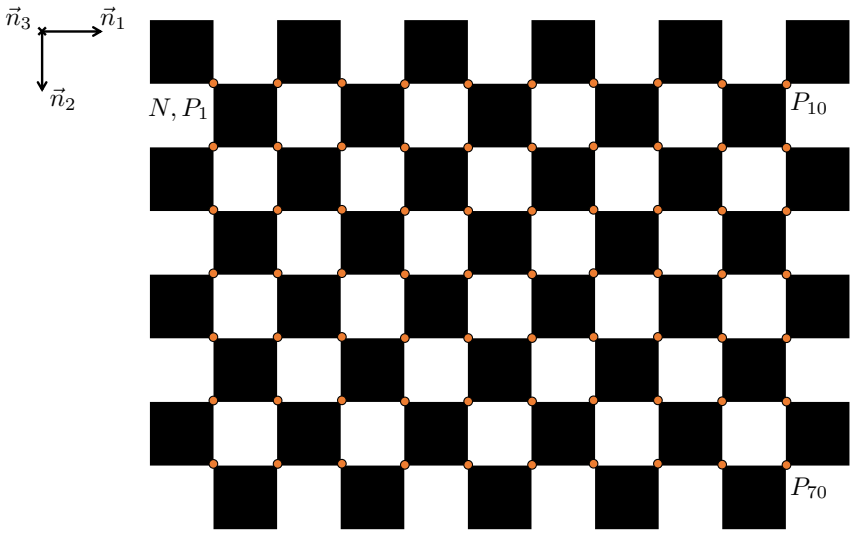

Figure 1: Camera calibration grid

Take the time to review the `struct` declarations within `src/Camera.h`. In particular, note that

- `Camera` contains the intrinsic camera parameters and some functions to evaluate the camera model (e.g., `worldToPixel`).

- `Chessboard` contains the geometry and dimensions of the calibration chessboard.
- `ChessboardImage` contains an image of the chessboard and the extrinsic camera parameters (pose) for that image.
- `ChessboardData` contains one instance of `Chessboard` and a collection of `ChessboardImage` objects.

Complete the implementation in `src/Camera.cpp` as follows:

1. In the `ChessboardImage::ChessboardImage` constructor, detect the corners of the chessboard image using `cv::findChessboardCorners`. To improve the accuracy using subpixel refinement, you may also use `cv::cornerSubPix`.
2. In the `Camera::calibrate` function, use `cv::calibrateCamera` to calibrate the camera model and set the appropriate members containing the intrinsic and extrinsic camera parameters.

Tip

The OpenCV function `cv::calibrateCamera` returns the camera pose, for each frame, in camera coordinates. The k^{th} element of documented argument `tvecs` corresponds to the translational vector $\mathbf{r}^c_{N/C}$ for the k^{th} calibration image and the k^{th} element of documented argument `rvecs` corresponds to the exponential parameterisation $\boldsymbol{\Theta}^c_n$ of the rotation matrix $\mathbf{R}^c_n = \exp \mathbf{S}(\boldsymbol{\Theta}^c_n)$ for the k^{th} calibration image, which can be found by calling `cv::Rodrigues` on the `rvecs[k]` variable. Note that unlike OpenCV, we will represent the camera position and orientation in world-fixed coordinates by $\mathbf{r}^n_{C/N}$ and $\mathbf{R}^n_c$, respectively.

3. In the `Camera::worldToVector` function, return the unit vector $\mathbf{u}^c_{P/C} = \frac{\mathbf{r}^c_{P/C}}{\|\mathbf{r}^c_{P/C}\|}$ that corresponds to the provided world position $\mathbf{r}^n_{P/N}$ and camera pose.
4. In the `Camera::vectorToPixel` function, use `cv::projectPoints` to map from a vector in camera coordinates, $\mathbf{r}^c_{P/C}$, to a pixel location in image coordinates, $\mathbf{r}^i_{Q/O}$.
5. In the `Camera::pixelToVector` function, return the unit vector $\mathbf{u}^c_{P/C}$ that corresponds to the provided pixel coordinates $\mathbf{r}^i_{Q/O}$. This can be done with the help of the `cv::undistortPoints` function.
6. In the `Camera::calcFieldOfView` function, calculate the horizontal, vertical and diagonal field of view of the camera, using the intrinsic parameters of the camera. This can be done by finding the angles between direction vectors that correspond to some of the edge and corner pixels in the image with the help of `Camera::pixelToVector`.
7. In the `Camera::isVectorWithinFOV` function, determine if the provided vector $\mathbf{r}^c_{P/C}$ lies within the camera field of view.
8. Build and run the application in verbose mode to display the calibration images and review the calibration results printed to the terminal.

Terminal

```
nerd@basement:~/MCHA4400/lab3$ cmake -G Ninja -B build && cd build
nerd@basement:~/MCHA4400/lab3/build$ ninja && ./lab3 -v
```

Images similar to Figure 2 should be displayed.

Figure 2: Example of calibration image with drawn chessboard corners in sequence.

2 Augmented reality (2 marks)

In the previous task, you found the camera matrix, distortion coefficients and camera poses. In this task, you will add some 3D geometry to the scene to validate the calibration of your camera. Complete the implementation in `src/Camera.cpp` as follows:

1. In the `ChessboardImages::drawBox` function, draw a rectangular prism with the base spanning the interior corners of the chessboard and height equal to 0.23 m (same height as the CodeNames box). To do this, generate a set of vertices for each line to draw in world coordinates and check that they lie within the camera field of view using `Camera::isWorldWithinFOV`. If so, find the image coordinates using `Camera::worldToPixel` and then draw them on the image using `cv::line`.
2. Build and run the application in verbose mode with box display.

   Terminal

   ```
   nerd@basement:~/MCHA4400/lab3/build$ ninja && ./lab3 -v -b
   ```

   Images similar to Figure 3 should be displayed.

> **Tip**
>
> To troubleshoot issues with drawing the box, try drawing only the base rectangle in the plane of the chessboard. Its vertices are guaranteed to be in the field of view for all the images used for camera calibration.

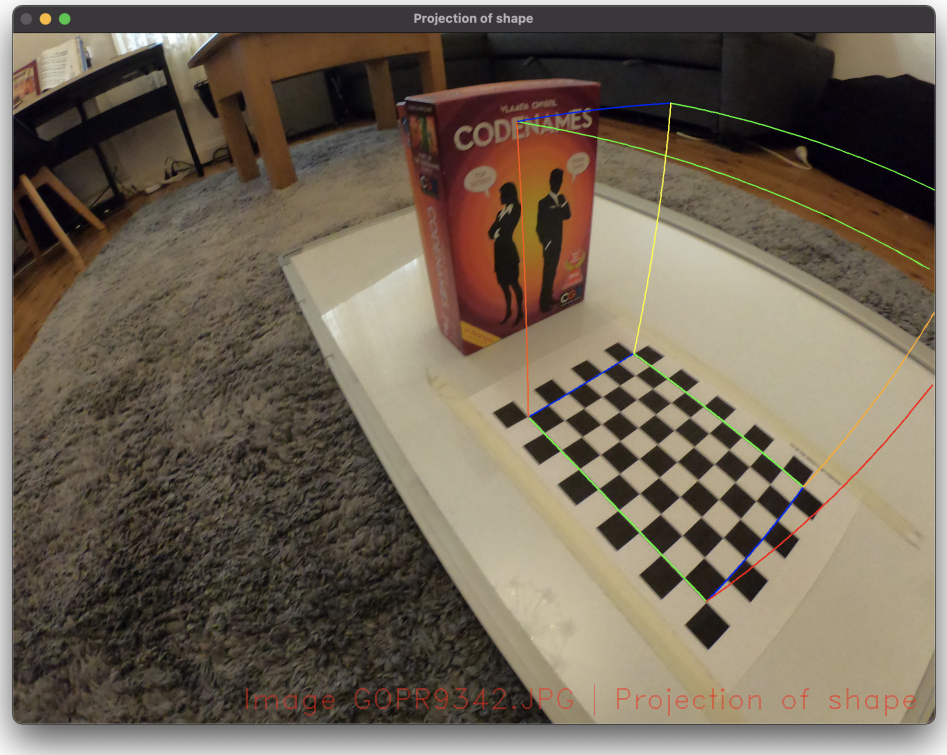

Figure 3: Box validation of camera calibration.