浙江大学

计算机视觉(本科)实验报告

作业名称:	HW4 相机标定与鸟瞰图
姓　　名:	夏子渊
学　　号:	3230103043
电子邮箱:	RukawaYuan1110@gmail.com
学　　院:	计算机科学与技术学院
专　　业:	软件工程
指导教师:	宋明黎/潘纲
报告日期:	2025 年 05 月 16 日

Table of Contents

1 功能简述及运行说明 3
1.1 功能简述 3
1.2 项目结构 4
1.3 运行说明 4
2 开发与运行环境 5
2.1 操作系统 5
2.2 安装依赖 5
3 算法的基本思路、原理、及流程 5
3.1 数学原理 5
3.1.1 相机标定 (Camera Calibration) 5
3.1.2 图像去畸变 (Image Undistortion) 6
3.1.3 鸟瞰图变换 (Bird's-eye View Transformation) 6
3.2 算法流程 7
4 具体实现——关键代码、函数与算法 8
4.1 整体流程 8
4.2 关键代码解析 9
4.2.1 1. 棋盘格角点检测与精细化 9
4.2.2 2. 相机标定 9
4.2.3 3. 图像去畸变 10
4.2.4 4. 计算并应用透视变换 (鸟瞰图) 10
4.3 时间、空间复杂度分析 11
4.3.1 时间复杂度 11
4.3.2 空间复杂度 12
5 实验结果与分析 13
5.1 实验结果 13
5.2 结果分析 18
5.2.1 1. 输出图像分析 18
5.2.2 2. 相机参数分析 (终端输出及 `camera_params.txt`) 19
6 结论与心得体会 20
6.1 实验结论 20
6.2 心得体会 20
6.3 实验收获 20
7 参考文献 21

1 功能简述及运行说明

1.1 功能简述

本实验实现了基于 OpenCV 的相机标定与鸟瞰视角图生成功能，主要功能及要求如下：

1. 命令行格式：`./calibration_birdeye <image_path> <output_path> <chessboard_width_points> <chessboard_height_points>`，例如 `./calibration_birdeye ../examples/example1.jpg 6 9`
2. 使用 OpenCV 的`findChessboardCorners`函数查找输入图像中的棋盘格角点，使用`calibrateCamera`函数进行相机标定，获得内参矩阵、畸变参数和 RMS 误差值，使用 `warpPerspective` 函数进行鸟瞰图的视角转换获得鸟瞰视角图。
3. 主要处理流程：
   - **参数检查和图像读取**：
     - 验证命令行参数（图像路径、棋盘格宽度点数、棋盘格高度点数）
     - 读取输入图像
     - 创建结果输出目录
   - **棋盘格角点检测**：
     - 使用`findChessboardCorners`函数寻找棋盘格角点
     - 通过`cornerSubPix`函数提高角点精度
     - 绘制并保存标记角点的图像
   - **相机标定**：
     - 准备图像点和对象点数据
     - 使用`calibrateCamera`函数计算相机内参矩阵和畸变系数
     - 输出相机参数到终端和文本文件
   - **图像去畸变**：
     - 利用计算得到的相机参数校正图像畸变
     - 保存去畸变后的图像
   - **鸟瞰图变换**：
     - 确定棋盘格在原始图像中的四个角点
     - 计算透视变换矩阵
     - 应用透视变换生成鸟瞰图
     - 保存鸟瞰图结果
4. 输出结果包含：
   - 在原图上叠加显示检测到的棋盘格角点的图像
   - 使用`undistort`函数处理之后的无畸变图像
   - 原图像的鸟瞰视角图
   - 包含相机参数的 txt 文件

1.2 项目结构

本项目的结构如下:

```
.
├── calibration_birdeye.cpp  # main program
├── CMakeLists.txt   # CMake configuration
├── README.md        # this file
├── report.pdf       # report
├── build            # build directory
└── examples         # example images and results
    ├── example1.jpg    # example image 1
    ├── example2.jpg    # example image 2
    └── example1_results
        ├── detected_corners.jpg    # detected corners
        ├── undistorted_image.jpg    # undistorted image
        └── birds_eye_view.jpg    # birds-eye view
        └── camera_params.txt    # camera parameters
    └── example2_results
        ├── detected_corners.jpg    # detected corners
        ├── undistorted_image.jpg    # undistorted image
        └── birds_eye_view.jpg    # birds-eye view
        └── camera_params.txt    # camera parameters
```

1.3 运行说明

- 构建：使用 clangd 作为编译器，CMake 进行构建，在 build 目录下得到可执行的文件。在提交的压缩包中已经上传了 build 文件夹，其中可执行文件位于 `build/calibration_birdeye` 路径下。如果想要自行构建, 请先切换到根目录下, 然后在终端中运行以下命令:

  ```
  mkdir build
  cd build
  cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=1 ..
  make -j
  ```

  即可得到对应的可执行文件。

- 运行：项目提供了三个示例图像，在 build 目录下运行以下命令处理示例图像:

  ```
  ./calibration_birdeye ../examples/example1.jpg 6 9
  ```

  或处理自定义图像:

```
./calibration_birdeye <image_path> <output_path>
<chessboard_width_points> <chessboard_height_points>
```

处理后的文件将保存在指定输出路径，包含标注棋盘格角点的图像、去畸变的图像、鸟瞰视角图、相机参数。

2 开发与运行环境

2.1 操作系统

在 MacOS（M4 Chip）上进行，硬件配置为 10 核 CPU，24GB 内存

2.2 安装依赖

实验使用 C++作为编程语言，因为需要 `<opencv2/opencv.hpp>` 头文件，因此需要先安装 opencv 库。同时因为构建时需要使用 CMake，因此也需要安装 Cmake。使用 brew 包管理器进行安装，具体命令如下：

```
brew install cmake
brew install opencv
```

3 算法的基本思路、原理、及流程

本项目通过棋盘格标定板实现相机标定，对捕获的图像进行畸变校正，并最终生成鸟瞰图。其核心在于利用 OpenCV 库提供的计算机视觉算法，从二维图像中恢复场景的三维信息和相机参数。

3.1 数学原理

3.1.1 相机标定 (Camera Calibration)

相机标定的目的是确定相机的内部参数（内参矩阵 K）和畸变系数 (Distortion Coefficients D)。

3.1.1.1 针孔相机模型 (Pinhole Camera Model)

描述三维世界点 (X, Y, Z) 到二维图像点 (u, v) 的映射关系：

$$s \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = K[R|t] \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

其中：

- s: 尺度因子。
- (u, v): 图像像素坐标。
- K: 相机内参矩阵，定义为：

$$K = \begin{pmatrix} f_x & \gamma & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}$$

f_x, f_y 是焦距在 x 和 y 方向上的分量（像素单位），(c_x, c_y) 是主点坐标，γ 是坐标轴倾斜因子（通常为 0）。

- $[R|t]$: 外参矩阵，R 是旋转矩阵，t 是平移向量。

3.1.1.2 镜头畸变 (Lens Distortion)

实际镜头存在畸变，主要包括径向畸变和切向畸变。

- 径向畸变:

$$x_{\text{corrected}} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

$$y_{\text{corrected}} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

- 切向畸变:

$$x_{\text{corrected}} = x + [2p_1 xy + p_2(r^2 + 2x^2)]$$

$$y_{\text{corrected}} = y + [p_1(r^2 + 2y^2) + 2p_2 xy]$$

其中 (x, y) 是理想无畸变图像坐标，$r^2 = x^2 + y^2$。畸变系数集为 $D = (k_1, k_2, p_1, p_2, k_3, k_4, k_5, k_6)$。

3.1.1.3 标定过程

通过拍摄已知尺寸的棋盘格，获取世界坐标点（棋盘格角点的三维坐标，代码中设 Z=0）和图像坐标点（检测到的棋盘格角点的二维像素坐标）。OpenCV 的 `calibrateCamera` 函数通过最小化重投影误差来估计 K 和 D。

3.1.2 图像去畸变 (Image Undistortion)

获得 K 和 D 后，使用 `undistort` 函数将原始图像中的每个像素重新映射到其无畸变的位置，生成校正图像。

3.1.3 鸟瞰图变换 (Bird's-eye View Transformation)

这是一种透视变换，通过一个 3x3 的单应性矩阵 H 实现。源图像点 $p = [x, y, 1]^T$ 与目标图像点 $p' = [x', y', w']^T$ 的关系为:

$$p' = H \cdot p$$

即:

$$\begin{pmatrix} x' \\ y' \\ w' \end{pmatrix} = \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

目标图像中的笛卡尔坐标为 $\left(\frac{x'}{w'}, \frac{y'}{w'}\right)$。矩阵 H 通过 `getPerspectiveTransform` 函数由源平面和目标平面上的四对对应点计算得到。

3.2 算法流程

1. 输入参数检查与初始化
   - 参数检查: 检查命令行参数是否正确 (需要提供图像路径、棋盘格宽度点数和高度点数)。
   - 图像读取：读取输入的棋盘格图像，若读取失败则报错退出。
   - 结果目录创建：在../examples/下创建以图像名命名的结果目录，用于保存输出文件。
2. 棋盘格角点检测
   - 角点检测：使用 findChessboardCorners 函数检测棋盘格角点，若未检测到角点则报错退出。
   - 角点精细化：将图像转为灰度图，通过 cornerSubPix 函数对检测到的角点进行亚像素级精细化，提高标定精度。
   - 角点绘制: 在原图上绘制检测到的角点, 并保存结果图像 (detected_corners.jpg)。
3. 相机标定
   - 准备标定数据:
     - 图像点：将检测到的角点坐标存入 imagePoints。
     - 物体点: 生成棋盘格的 3D 世界坐标 (假设棋盘格在 Z=0 平面上，每个方格的物理尺寸为 1 单位)。
   - 相机标定：调用 calibrateCamera 函数计算相机内参矩阵（cameraMatrix)、畸变系数（distCoeffs）和每幅图像的旋转向量（rvecs）与平移向量（tvecs)。
   - 输出标定结果:
     - 在终端打印标定误差（RMS)、内参矩阵和畸变系数。
     - 将结果保存到文本文件（camera_params.txt)。
4. 图像去畸变
   - 去畸变处理：使用 undistort 函数根据标定结果对原始图像进行去畸变处理。
   - 结果显示与保存：显示并保存去畸变后的图像（undistorted_image.jpg)。
5. 鸟瞰视角图生成
   - 源四边形选取：从检测到的角点中提取棋盘格的四个顶点（左上、右上、右下、左下)。
   - 目标四边形定义：根据棋盘格的实际尺寸（方格数×预设的方格像素大小 squareSize=30）计算目标四边形的尺寸，将目标四边形放置在缩放后的图像中心 (缩放因子 scaleFactor=1.5)。
   - 透视变换矩阵计算：通过 getPerspectiveTransform 计算从源四边形到目标四边形的透视变换矩阵。
   - 鸟瞰图生成：应用 warpPerspective 对原始图像进行透视变换，生成鸟瞰视角图。
   - 结果显示与保存：显示并保存鸟瞰视角图（birds_eye_view.jpg)。

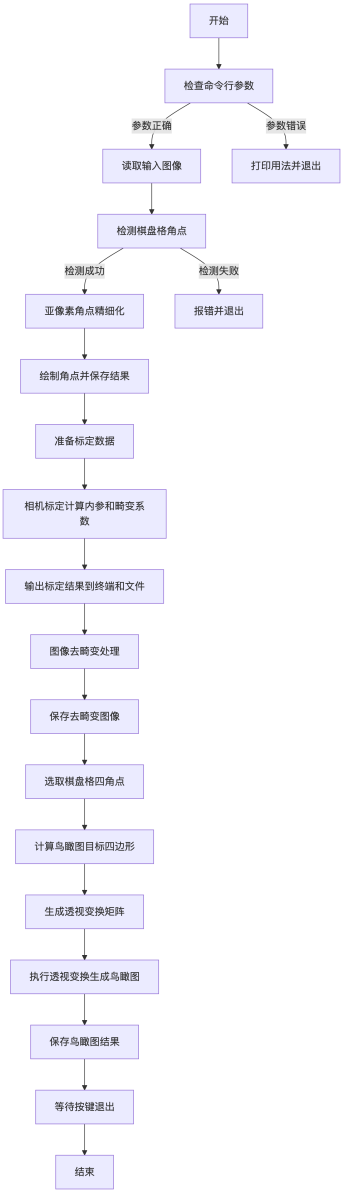
4 具体实现——关键代码、函数与算法

4.1 整体流程

本椭圆拟合实现主要分为以下几个步骤:

- **参数检查和图像读取**
- **棋盘格角点检测**
- **相机标定**
- **图像去畸变**
- **鸟瞰图变换**

下图展示了完整的算法流程图:



4.2 关键代码解析

4.2.1 1. 棋盘格角点检测与精细化

```
// find chessboard corners
vector<Point2f> corners;
bool found = findChessboardCorners(image, boardSize, corners);

// improve corner accuracy
Mat grayImage;
cvtColor(image, grayImage, COLOR_BGR2GRAY);
cornerSubPix(grayImage, corners, Size(11, 11), Size(-1, -1),
             TermCriteria(TermCriteria::EPS + TermCriteria::COUNT, 30,
0.1));
)
```

- `findChessboardCorners`: 此函数在输入图像 `image` 中检测 `boardSize` 定义的棋盘格内部角点。`boardSize` 是一个 `cv::Size` 对象，表示棋盘格内部角点的列数和行数。检测到的角点存储在 `corners` 向量中。
- `cvtColor`: 将彩色图像转换为灰度图像 `grayImage`，因为亚像素角点检测通常在灰度图上进行，以减少颜色信息的干扰。
- `cornerSubPix`: 对 `findChessboardCorners` 初步检测到的角点进行亚像素级别的精细化处理。它在一个小邻域内迭代寻找更精确的角点位置，`Size(11, 11)` 定义了搜索窗口的大小，`TermCriteria` 定义了迭代终止的条件（达到最大迭代次数 30 或精度 0.1）。

4.2.2 2. 相机标定

```
// prepare camera calibration data
vector<vector<Point2f>> imagePoints;
imagePoints.push_back(corners);

vector<vector<Point3f>> objectPoints(1);
for (int i = 0; i < boardSize.height; i++) {
    for (int j = 0; j < boardSize.width; j++) {
        objectPoints[0].push_back(Point3f(j, i, 0.0f));
    }
}

// calibrate camera
Mat cameraMatrix = Mat::eye(3, 3, CV_64F);
Mat distCoeffs = Mat::zeros(8, 1, CV_64F);
```

```
vector<Mat> rvecs, tvecs;

double rms = calibrateCamera(objectPoints, imagePoints, image.size(),
                             cameraMatrix, distCoeffs, rvecs, tvecs);
```

- `objectPoints`: 存储棋盘格角点的三维世界坐标。这里假设棋盘格位于 Z=0 的平面上，并且相邻角点在 X 和 Y 方向上的距离为一个单位长度。
- `imagePoints`: 存储在图像中检测到的对应二维角点坐标（即 `corners`）。
- `calibrateCamera`: 这是 OpenCV 中进行相机标定的核心函数。它接收三维世界点 `objectPoints`、对应的二维图像点 `imagePoints`、图像尺寸 `image.size()` 作为输入，并计算输出相机内参矩阵 `cameraMatrix`、畸变系数 `distCoeffs`、以及每张视图的旋转向量 `rvecs` 和平移向量 `tvecs`。返回值 `rms` 是重投影误差的均方根，用于评估标定精度。

4.2.3 3. 图像去畸变

```
// correct image
Mat undistortedImage;
undistort(image, undistortedImage, cameraMatrix, distCoeffs);
```

- `undistort`: 该函数使用标定得到的相机内参矩阵 `cameraMatrix` 和畸变系数 `distCoeffs` 来校正原始图像 `image` 中的镜头畸变，生成无畸变的图像 `undistortedImage`。它会根据畸变模型将原始图像中的像素重新映射到校正后的位置。

4.2.4 4. 计算并应用透视变换 (鸟瞰图)

```
// define the four corners of the chessboard in the original image
Point2f srcQuad[4];
srcQuad[0] = corners[0];                                    // top-left
corner
srcQuad[1] = corners[boardSize.width-1];                    // top-right
corner
srcQuad[2] = corners[boardSize.width*boardSize.height-1];   // bottom-
right corner
srcQuad[3] = corners[boardSize.width*(boardSize.height-1)]; // bottom-
left corner

// define the position of the chessboard in the birds-eye view
(centered)
Point2f dstQuad[4];
```

```
// ... (calculations for dstWidth, dstHeight, offsetX, offsetY) ...
dstQuad[0] = Point2f(offsetX, offsetY);                    // top-left
corner
dstQuad[1] = Point2f(offsetX + dstWidth, offsetY);         // top-right
corner
dstQuad[2] = Point2f(offsetX + dstWidth, offsetY + dstHeight);  //
bottom-right corner
dstQuad[3] = Point2f(offsetX, offsetY + dstHeight);        // bottom-left
corner

// calculate the perspective transformation matrix
Mat perspectiveMatrix = getPerspectiveTransform(srcQuad, dstQuad);

// apply perspective transformation to create a birds-eye view
Mat birdsEyeView;
warpPerspective(image, birdsEyeView, perspectiveMatrix,
Size(birdsEyeWidth, birdsEyeHeight));
```

)

- `srcQuad`: 定义了源图像（原始相机视角）中棋盘格的四个外角点。这些点从 `corners` 向量中提取。
- `dstQuad`: 定义了这四个角点在目标鸟瞰图中的期望位置。代码中计算这些位置使得棋盘格在输出图像中居中，并且每个方格具有预定义的像素大小 `squareSize`。
- `getPerspectiveTransform`: 根据源四边形 `srcQuad` 和目标四边形 `dstQuad` 的四对对应点，计算出 3x3 的透视变换矩阵 (单应性矩阵) `perspectiveMatrix`。
- `warpPerspective`: 使用计算得到的 `perspectiveMatrix` 对原始图像 `image`（注意, 代码中使用的是原始带畸变的图像）进行透视变换, 生成鸟瞰图 `birdsEyeView`。输出图像的尺寸由 `Size(birdsEyeWidth, birdsEyeHeight)` 决定。

4.3 时间、空间复杂度分析

设输入图像的宽度为 W_I，高度为 H_I。棋盘格内部角点数为 $N_c = (\text{boardSize.width}) \times (\text{boardSize.height})$。 鸟瞰图的宽度为 W_B, 高度为 H_B。

4.3.1 时间复杂度

1. **图像读取与预处理**:
   - `imread`: 大致与图像像素数成正比，即 $O(W_I H_I)$。
   - `cvtColor` (BGR to Gray): $O(W_I H_I)$。
2. **棋盘格角点检测**:

- `findChessboardCorners`: 这是一个复杂的操作, 其内部涉及到图像扫描、模式匹配等。其复杂度通常高于线性扫描, 可能与图像尺寸和棋盘格复杂度有关。一个粗略的估计可以是 $O(W_I H_I)$ 到 $O(W_I H_I N_c)$ 之间，取决于具体实现和图像内容。
- `cornerSubPix`: 迭代优化，对于每个角点，在小窗口内操作。设迭代次数为 K_s，窗口大小为 $W_s \times H_s$。则复杂度为 $O(N_c K_s W_s H_s)$。由于 K_s, W_s, H_s 通常是小常数，可以近似为 $O(N_c)$。

3. **相机标定**:
   - `calibrateCamera`: 对于单张图像标定，主要涉及到求解一个非线性最小二乘问题。其复杂度与角点数 N_c 相关。通常使用 Levenberg-Marquardt 等迭代算法。精确的复杂度分析比较困难，但可以认为它与 N_c 的某个多项式相关，例如 $O(N_c^k)$，其中 k 通常较小。由于只使用一张图像，迭代次数会比较少。
4. **图像去畸变**:
   - `undistort`: 需要为目标图像的每个像素 (u', v') 计算其在原始畸变图像中的对应位置 (u, v) 并进行插值。因此，其复杂度与输出无畸变图像的像素数成正比。如果输出图像与输入图像尺寸相同，则为 $O(W_I H_I)$。
5. **鸟瞰图变换**:
   - `getPerspectiveTransform`: 求解一个包含 8 个未知数的小型线性方程组 (基于 4 对点)。其复杂度是常数级别，可视为 $O(1)$。
   - `warpPerspective`: 与 `undistort` 类似, 需要为鸟瞰图的每个像素计算其在原始图像中的对应位置并插值。其复杂度与鸟瞰图的像素数成正比，即 $O(W_B H_B)$。代码中 $W_B = W_I \times \text{scaleFactor}$, $H_B = H_I \times \text{scaleFactor}$。

整体近似时间复杂度: 主要由图像处理步骤 (如 `findChessboardCorners`, `undistort`, `warpPerspective`) 主导。如果忽略标定过程 (因为它通常是一次性的或者对于单张图影响不大), 则主要操作是像素级的。因此, 可以近似为 $O(W_I H_I + W_B H_B)$。由于 W_B, H_B 与 W_I, H_I 成比例，也可以说是 $O(W_I H_I)$。`findChessboardCorners` 可能是最耗时的部分之一。

4.3.2 空间复杂度

1. **图像存储**:
   - `image` (原始图像): $O(W_I H_I)$ (通常为 3 通道)。
   - `grayImage` (灰度图像): $O(W_I H_I)$ (单通道)。
   - `imageWithCorners` (绘制角点的图像): $O(W_I H_I)$。
   - `undistortedImage` (去畸变图像): $O(W_I H_I)$。
   - `birdsEyeView` (鸟瞰图): $O(W_B H_B)$。
2. **角点数据**:
   - `corners` (Point2f vector): $O(N_c)$。
   - `objectPoints` (Point3f vector): $O(N_c)$。

3. **相机参数与变换矩阵**:
   - `cameraMatrix` (3x3 Mat): $O(1)$。
   - `distCoeffs` (8x1 Mat): $O(1)$。
   - `rvecs`, `tvecs` (对于单视图，数量少): $O(1)$。
   - `perspectiveMatrix` (3x3 Mat): $O(1)$。

整体空间复杂度: 主要由存储各个版本的图像所决定。因此，空间复杂度近似为 $O(W_I H_I + W_B H_B)$。由于 W_B, H_B 与 W_I, H_I 成比例，也可以说是 $O(W_I H_I)$。

5 实验结果与分析

在压缩包的 examples 目录下已有提供示例输入与输出，以下分析以此为例:

5.1 实验结果

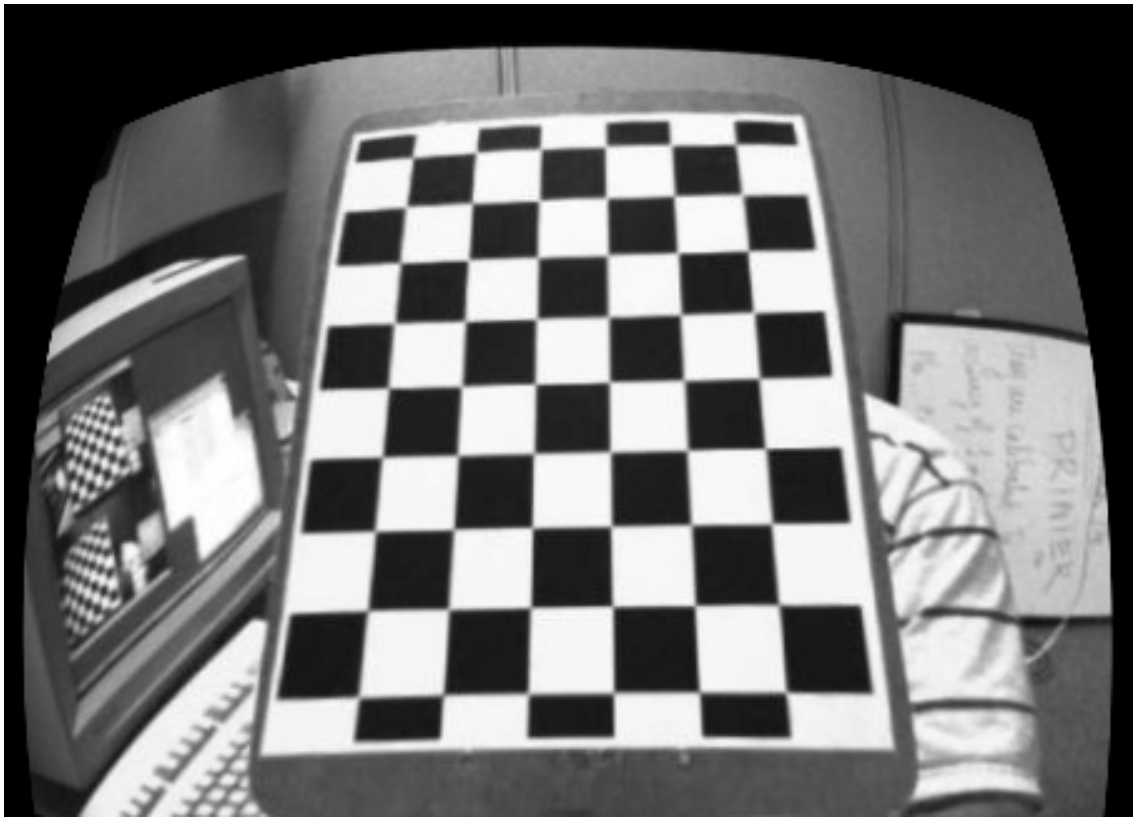
在测试中使用了两张从互联网上获得的包含棋盘格的照片，分别进行测试。

以下是针对 2 张输入图运行程序之后各自得到的输出图与终端输出（相机内参和畸变参数）:

1. 图像一
   - 原始图像:

   - 输出图像:
     - 检测到的角点:

- 去畸变之后的图像:



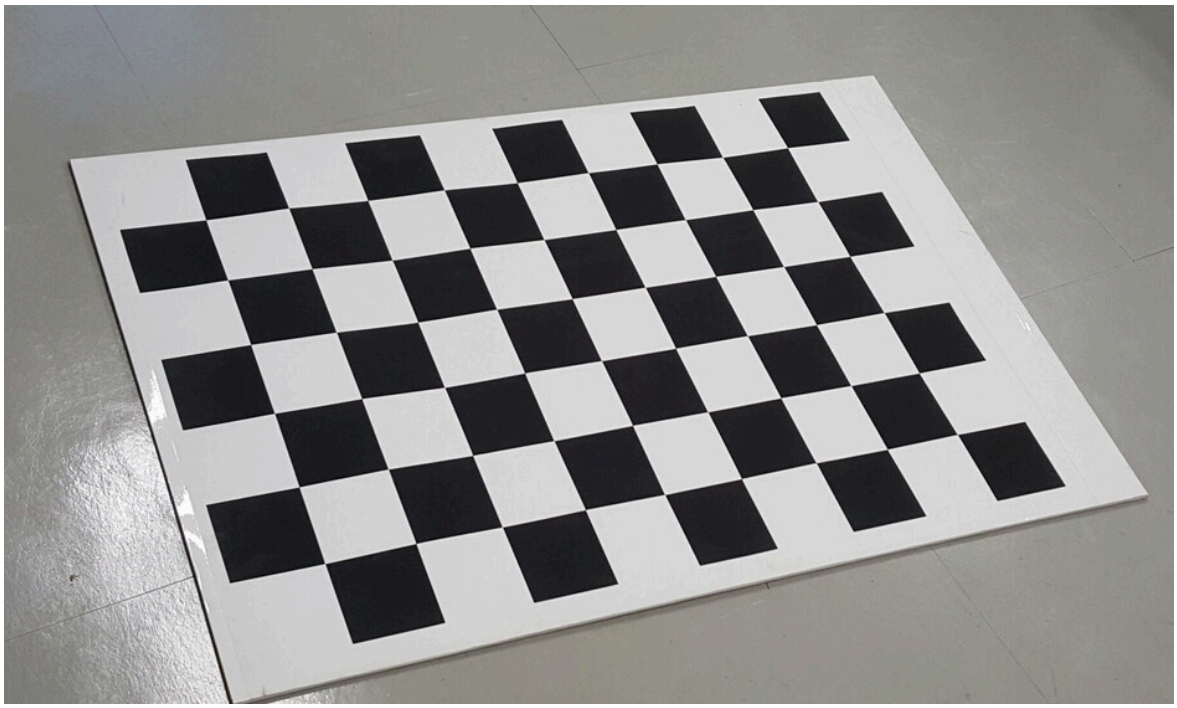
- 鸟瞰图:

- 终端输出（相机参数信息）:

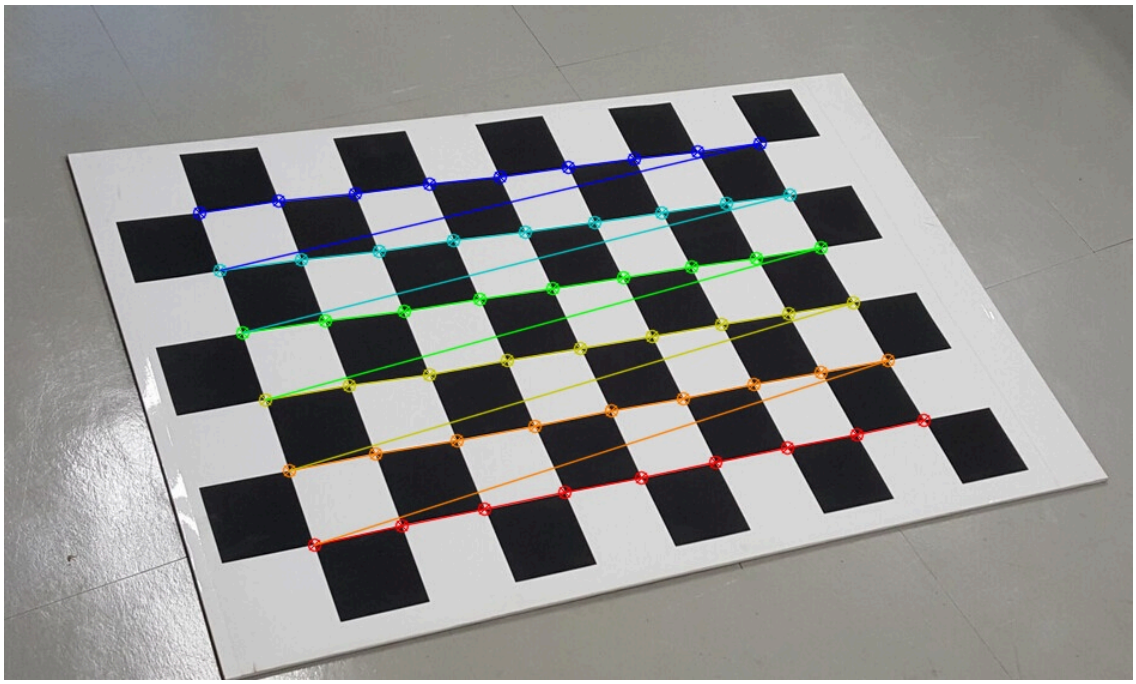
```
----------------------------------------
camera calibration RMS error: 0.468636
camera intrinsic matrix:
[302.2410671680448, 0, 232.8710364568988;
 0, 323.9000953065926, 237.9465865588338;
 0, 0, 1]
distortion coefficients:
[-0.08054365285126107;
 -0.2923873900105356;
 -0.02286329750783305;
 0.011725432013706 74;
 1.133418509323938]
----------------------------------------
```

2. 图像二:

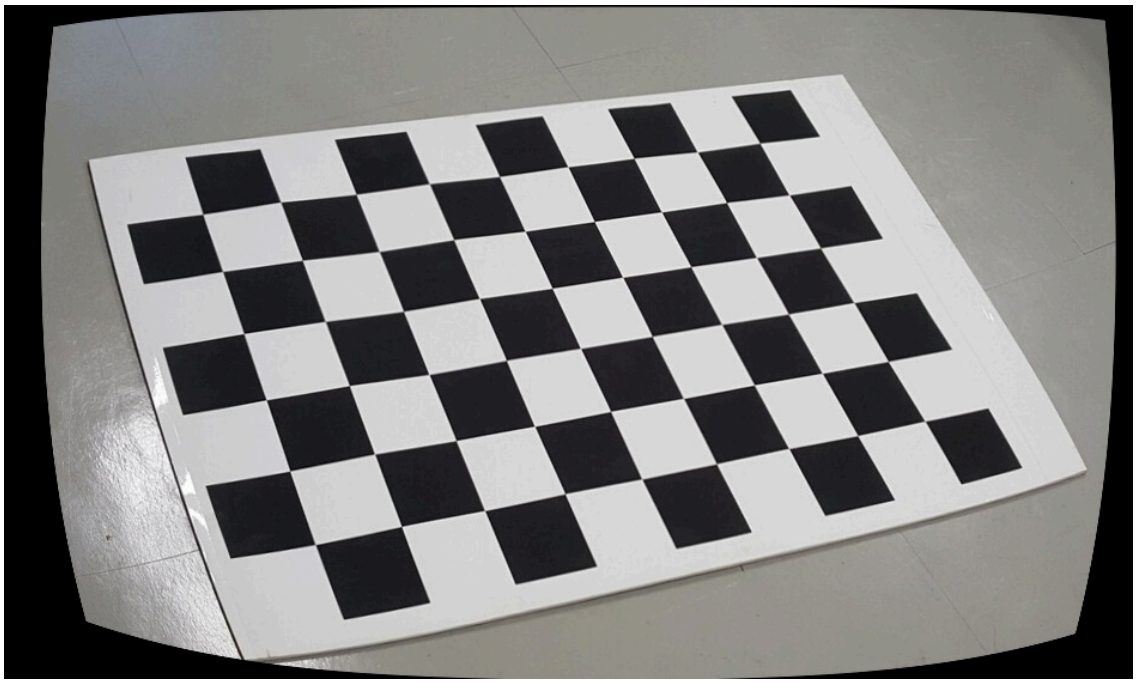
- 原始图像:



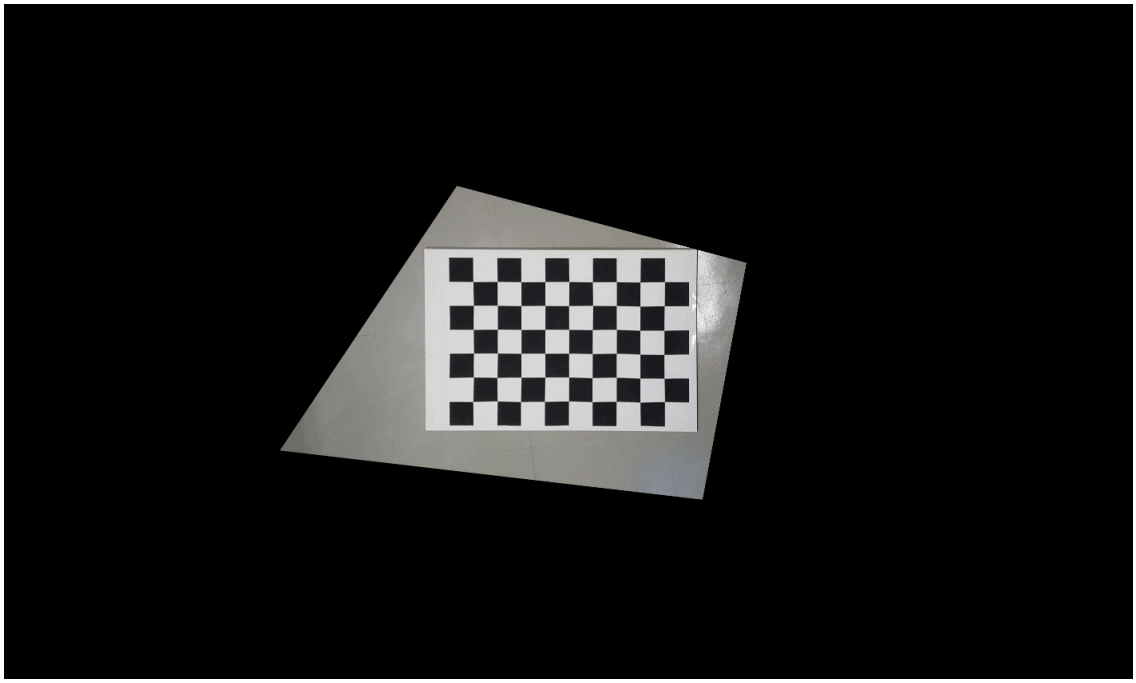
- 输出图像：
  - 检测到的角点：



  - 去畸变之后的图像：



- 鸟瞰图:



- 终端输出（相机参数信息）:

```
----------------------------------------
camera calibration RMS error: 0.603634
camera intrinsic matrix:
[1582.563436731228, 0, 494.9780934549146;
 0, 1605.924956142097, 184.4028566567165;
 0, 0, 1]
distortion coefficients:
[0.1541813831442931;
 -7.008130411250871;
 -0.02279566697394872;
 -0.0060038764036944;
 157.4299300900941]
----------------------------------------
```

5.2 结果分析

5.2.1 1. 输出图像分析

程序生成三张主要的中间和最终结果图像，分别保存在 `../examples/<imageName>_results/` 目录下。

5.2.1.1 a. 角点检测结果 (`detected_corners.jpg`)

- **结果分析**:
  - 所有棋盘格内部角点都被准确检测到，并且绘制的标记（彩色圆圈或连线）精确地覆盖在角点位置。
  - 没有出现漏检(部分角点未被识别)或误检(非角点位置被错误识别为角点)的情况。
  - 角点标记的分布均匀且符合棋盘格的实际结构。
  - `cornerSubPix` 的亚像素精细化效果较好，标记非常精确地位于角点中心。

5.2.1.2 b. 图像去畸变结果 (`undistorted_image.jpg`)

- **结果分析**:
  - 与原始图像对比，原先因镜头畸变而弯曲的直线（例如棋盘格的边缘线、背景中的直线物体）在去畸变图像中变直。
  - 径向畸变严重时，图像中心区域被拉伸，而边缘区域被压缩（桶形畸变)。去畸变后，这种效应应得到显著改善，物体形状更接近真实。
  - 图像边缘不存在过度拉伸或裁剪。`undistort` 函数可以调整输出图像的尺寸以包含所有有效像素或与原图相同尺寸。
  - 整体视觉效果更自然，物体的比例关系更准确。

5.2.1.3 c. 鸟瞰图结果 (`birds_eye_view.jpg`)

- **结果分析**:
  - 棋盘格在鸟瞰图中呈现为理想的矩形网格，所有格子线平行且等距（根据 `squareSize` 的设定)。
  - 棋盘格按照代码中的设定（通过 `offsetX`, `offsetY`）在鸟瞰图中居中显示。

- 鸟瞰图的清晰度正常。由于透视变换涉及到像素的重映射和插值，会有一定程度的模糊，特别是在被拉伸的区域。
- 变换后的物体比例正常。例如，棋盘格的方块在鸟瞰图中接近正方形。

5.2.2 2. 相机参数分析 (终端输出及 `camera_params.txt`)

程序将标定得到的相机参数输出到终端，并保存到 `../examples/<imageName>_results/camera_params.txt` 文件中。

5.2.2.1 a. RMS 重投影误差 (`camera calibration RMS error`)

- **解读**:
  - RMS (Root Mean Square) 重投影误差是衡量相机标定精度的重要指标。它表示通过标定得到的相机参数将三维世界点重新投影回图像平面时，计算得到的投影点与实际检测到的图像点之间的平均距离（以像素为单位）。
  - **典型值**: 一个较低的 RMS 值（通常小于 1.0 像素，理想情况下更低，如 0.1-0.5 像素）表明标定结果较好，相机模型能较好地拟合观测数据。较高的 RMS 值可能意味着标定数据不准确（如角点检测错误）、棋盘格假设不满足（如棋盘格不平整）或相机模型不完全适用。
  - **分析**: 实际得到的 RMS 值为 0.4686，在可接受的范围内。

5.2.2.2 b. 相机内参矩阵 (`cameraMatrix`)

格式如下:

$$K = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}$$

- **解读**:
  - f_x, f_y: 焦距在图像传感器 x 轴和 y 轴方向上的投影，单位为像素。它们反映了相机的放大能力。通常情况下，f_x 和 f_y 的值比较接近。
  - c_x, c_y: 主点坐标，即相机光轴与图像平面的交点，单位为像素。理论上主点应位于图像中心（即图像宽度/2，图像高度/2 附近）。
  - **分析**: 实际得到的 f_x, f_y 的值合理，(c_x, c_y) 大致靠近图像中心。实际值为

```
[302.2410671680448, 0, 232.8710364568988;
0, 323.9000953065926, 237.9465865588338;
0, 0, 1]
```

5.2.2.3 c. 畸变系数 (`distCoeffs`)

包含径向畸变系数 $(k_1, k_2, k_3, ...)$ 和切向畸变系数 (p_1, p_2)。在本项目中，`distCoeffs` 是一个 5x1 的矩阵，包含 k_1, k_2, p_1, p_2, k_3。

- **解读**:
  - $k_1, k_2, k_3, ...$: 径向畸变系数。k_1 是最主要的径向畸变项。正值通常对应枕形畸变，负值对应桶形畸变。k_2, k_3 等是高阶项，用于描述更复杂的径向畸变。

- p_1, p_2: 切向畸变系数。由相机组装过程中传感器与镜头不完全平行导致。
- **分析**: 这些系数的量级较小。如果绝对值较大，说明相机镜头存在较明显的畸变。如果这些系数接近于零，则表明镜头畸变很小。这些系数值将直接影响 `undistort` 函数的效果。例如，如果 k_1 较大，则在 `undistorted_image.jpg` 中应能观察到明显的形状校正。实际值为

```
[-0.08054365285126107;
 -0.2923873900105356;
 -0.02286329750783305;
 0.01172543201370674;
 1.133418509323938]
```

位于合理区间内。

6 结论与心得体会

6.1 实验结论

通过这次相机标定与鸟瞰图变换的实验，我深刻理解了计算机视觉中几何变换的重要性与应用。实验结果表明，基于棋盘格的相机标定方法能够有效地获取相机内参矩阵和畸变系数，并用于校正图像畸变。

在本次实验中，我成功实现了对测试图像的角点检测、相机标定、畸变校正和鸟瞰图变换。从最终生成的鸟瞰图可以看出，透视变换能够将原本因视角产生的透视畸变转换为正视图像，使得棋盘格呈现规则的矩形网格。这一变换在自动驾驶、道路检测等领域具有重要的应用价值，能够为后续的图像分析和处理提供更加准确的几何信息。实验还证明了 OpenCV 库在处理这类计算机视觉任务时的强大功能和便捷性，通过简洁的代码实现了复杂的几何变换过程。

6.2 心得体会

这次实验让我体会到了计算机视觉从理论到实践的转化过程。在实验初期，我曾对相机模型和透视变换的数学原理感到困惑，特别是在理解畸变模型和投影矩阵时遇到了一定的难度。然而，当我亲手实现代码并观察到变换前后的图像对比时，这些抽象的概念变得具体而清晰。我体会到编程实践对理解理论知识的重要性，代码实现的过程就是对理论的最佳验证。

此外，在调试过程中，我也遇到了一些挑战，比如棋盘格角点检测失败的问题，这让我学会了更加仔细地检查输入参数和调整算法参数。通过反复尝试和调整，最终看到成功生成的鸟瞰图时，那种成就感令人难忘。这次实验也让我更加理解了相机成像的物理过程，以及如何通过数学模型和算法来模拟和纠正这一过程中的各种失真。

6.3 实验收获

这次实验给我带来了丰富的技术收获和能力提升。首先，我掌握了 OpenCV 中与相机标定和图像变换相关的核心函数的使用方法，如 findChessboardCorners、

calibrateCamera、undistort 和 warpPerspective 等。其次，我深入理解了相机成像模型，包括针孔相机模型、畸变模型以及如何通过数学方法对其进行校正。

在实现鸟瞰图变换的过程中，我学会了如何确定合适的源点和目标点，以及如何计算和应用透视变换矩阵。此外，这次实验也提升了我的代码组织能力和问题解决能力，尤其是在处理图像数据、设计变换流程和调试复杂算法时。我还学会了如何合理设置参数以获得最佳的变换效果，这对于实际应用中的图像处理至关重要。最重要的是，通过这次实验，我对计算机视觉领域产生了更浓厚的兴趣，期待在未来的学习和研究中进一步探索这个充满挑战和机遇的领域。

7 参考文献

- OpenCV 官方网站：https://opencv.org
- OpenCV 官方文档：https://docs.opencv.org/4.x/d1/dfb/intro.html