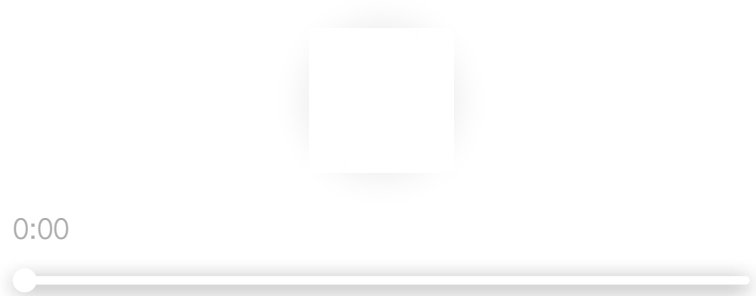
一份简单的车辆环视全景系统实现

赵亮 / 2019-10-05

分类: 自动驾驶 / **标签:** OpenCV, Python, 环视系统 / **字数:** 5720

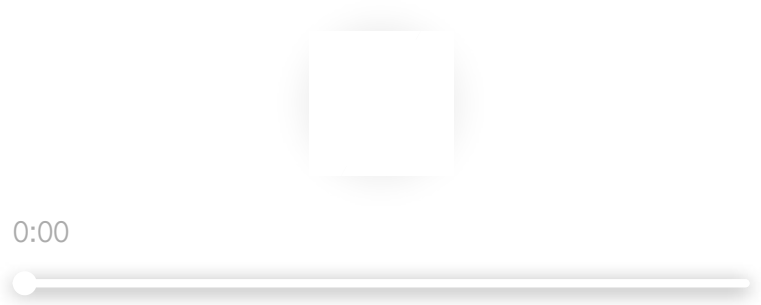
关于车辆的全景环视系统网上已经有很多的资料，然而几乎没有可供参考的代码，这一点对入门的新人来说非常不友好。这个项目的目的就是介绍全景系统的原理，并给出一份可以实际运行的、基本要素齐全的 Python 实现供大家参考。环视全景系统涉及的知识并不复杂，只需要读者了解相机的标定、透视变换，并懂得如何使用 OpenCV。项目代码维护在 github 上。

这个程序最初是在一辆搭载了一台 AGX Xavier 的无人小车上开发的，运行效果如下：



小车上搭载了四个 USB 环视鱼眼摄像头，相机传回的画面分辨率为 640x480，图像首先经过畸变校正，然后在射影变换下转换为对地面的鸟瞰图，最后拼接起来经过平滑处理后得到了上面的效果。全部过程在 CPU 中进行处理，整体运行流畅。

后来我把代码重构以后移植到一辆乘用车上 (处理器是同型号的 AGX)，得到了差不多的效果：



这个版本使用的是四个 960x640 的 csi 摄像头，输出的全景图分辨率为 1200x1600，在不进行亮度均衡处理时全景图处理线程运行速度大约为 17 fps，加入亮度均衡处理后骤降到只有 7 fps。我认为适当缩小分辨率的话 (比如采用 480x640 的输出可以将像素个数降低到原来的 1/6) 应该也可以获得流畅的视觉效果。

> **注：** 画面中黑色的部分是相机投影后出现的盲区，这是因为前面的相机为了避开车标的部位安装在了车头左侧且角度倾斜，所以视野受到了限制。想象一个人歪着脖子还斜着眼走路的样子 ...

这个项目的实现比较粗糙，仅作为演示项目展示生成环视全景图的基本要素，大家领会精神即可。我开发这个项目的目的是为了在自动泊车时实时显示车辆的轨迹，同时也用来作为我指导的实习生的实习项目。由于之前没有经验和参照，大多数算法和流程都是琢磨着写的，不见得高明，请大家多指教。代码是 Python 写成的，效率上肯定不如 C++，所以仅适合作学习和验证想法使用。

下面就来一一介绍我的实现步骤。

硬件和软件配置

我想首先强调的是，硬件配置是这个项目中最不需要费心的事情，在第一个小车项目中使用的硬件如下：

1. 四个 USB 鱼眼相机，支持的分辨率为 640x480|800x600|1920x1080 三种。我这里因为是需要在 Python 下面实时运行，为了效率考虑设置的分辨率是 640x480。
2. 一台 AGX Xavier。实际上在普通笔记本上跑一样溜得飞起。

第二个乘用车项目使用的硬件如下：

1. 四个 csi 摄像头，设置的分辨率是 960x640。
2. 一台 AGX Xavier，型号同上面的小车项目一样，不过外加了一个工控机接收 csi 摄像头画面。

我认为只要你只要有四个视野足够覆盖车周围的摄像头，再加一个普通笔记本电脑就足够进行全部的离线开发了。

软件配置如下：

1. 操作系统 Ubuntu 16.04/18.04.
2. Python>=3.
3. OpenCV>=3.
4. PyQt5.

其中 PyQt5 主要用来实现多线程，方便将来移植到Qt 环境。

项目采用的若干约定

为了方便起见，在本项目中四个环视相机分别用 `front` 、`back` 、`left` 、`right` 来指代，并假定其对应的设备号是整数，例如 0, 1, 2, 3。实际开发中请针对具体情况进行修改。

相机的内参矩阵记作 `camera_matrix` ，这是一个 3x3 的矩阵。畸变系数记作 `dist_coeffs` ，这是一个 1x4 的向量。相机的投影矩阵记作 `project_matrix` ，这是一个 3x3 的射影矩阵。

准备工作：获得原始图像与相机内参

首先我们需要获取每个相机的内参矩阵与畸变系数。我在项目中附上了一个脚本 run_calibrate_camera.py，你只需要运行这个脚本，通过命令行参数告诉它相机设备号，是否是鱼眼相机，以及标定板的网格大小，然后手举标定板在相机面前摆几个姿势即可。

以下是视频中四个相机分别拍摄的原始画面，顺序依次为前、后、左、右，并命名为 `front.png` 、`back.png` 、`left.png` 、`right.png` 保存在项目的 `images/` 目录下。

四个相机的内参文件分别为 `front.yaml`、`back.yaml`、`left.yaml`、`right.yaml`，这些内参文件都存放在项目的 yaml 子目录下。

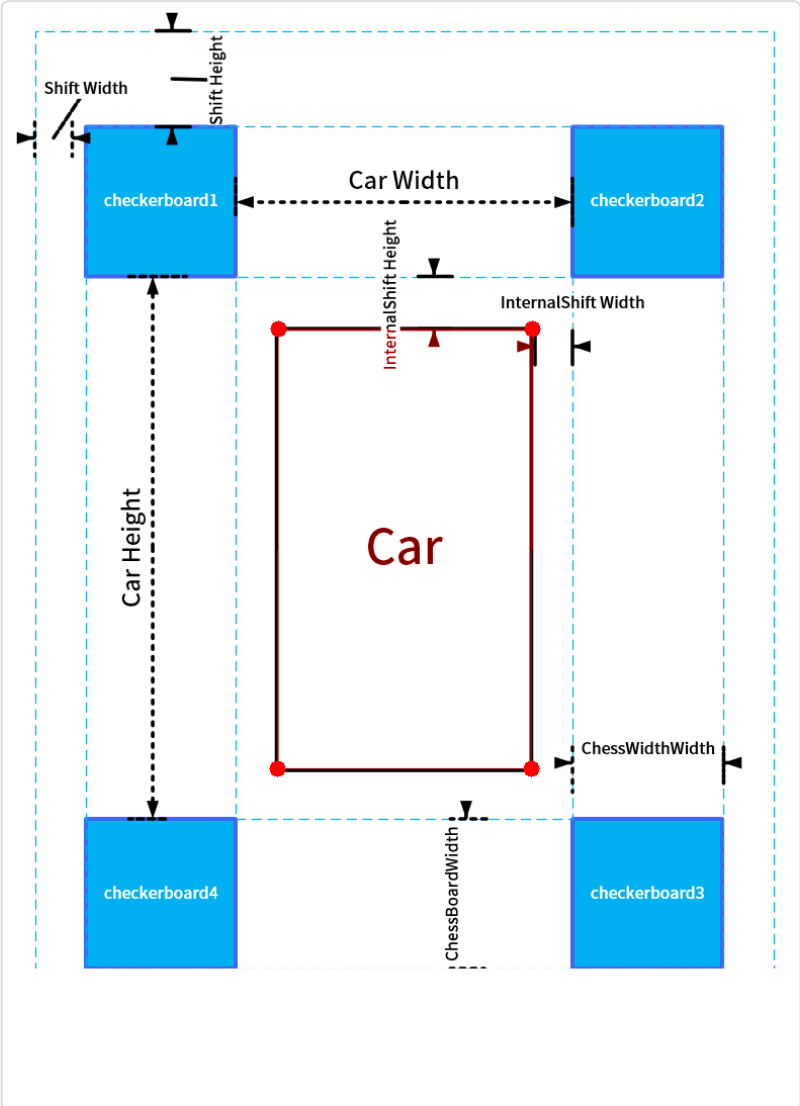
你可以看到图中地面上铺了一张标定布，这个布的尺寸是 `6mx10m`，每个黑白方格的尺寸为 `40cmx40cm`，每个圆形图案所在的方格是 `80cmx80cm`。我们将利用这个标定物来手动选择对应点获得投影矩阵。

设置投影范围和参数

接下来我们需要获取每个相机到地面的投影矩阵，这个投影矩阵会把相机校正后的画面转换为对地面上某个矩形区域的鸟瞰图。这四个相机的投影矩阵不是独立的，它们必须保证投影后的区域能够正好拼起来。

这一步是通过联合标定实现的，即在车的四周地面上摆放标定物，拍摄图像，手动选取对应点，然后获取投影矩阵。

请看下图：



首先在车身的四角摆放四个标定板，标定板的图案大小并无特殊要求，只要尺寸一致，能在图像中清晰看到即可。每个标定板应当恰好位于相邻的两个相机视野的重合区域中。

在上面拍摄的相机画面中车的四周铺了一张标定布，这个具体是标定板还是标定布不重要，只要能清楚的看到特征点就可以了。

然后我们需要设置几个参数：(以下所有参数均以厘米为单位)

- `innerShiftWidth`，`innerShiftHeight`：标定板内侧边缘与车辆左右两侧的距离，标定板内侧边缘与车辆前后方的距离。
- `shiftWidth`，`shiftHeight`：这两个参数决定了在鸟瞰图中向标定板的外侧看多远。这两个值越大，鸟瞰图看的范围就越大，相应地远处的物体被投影后的形变也越严重，所以应酌情选择。

- `totalWidth`，`totalHeight`：这两个参数代表鸟瞰图的总宽高，在我们这个项目中标定布宽 6m 高 10m，于是鸟瞰图中地面的范围为 `(600 + 2 * shiftWidth, 1000 + 2 * shiftHeight)`。为方便计我们让每个像素对应 1 厘米，于是鸟瞰图的总宽高为

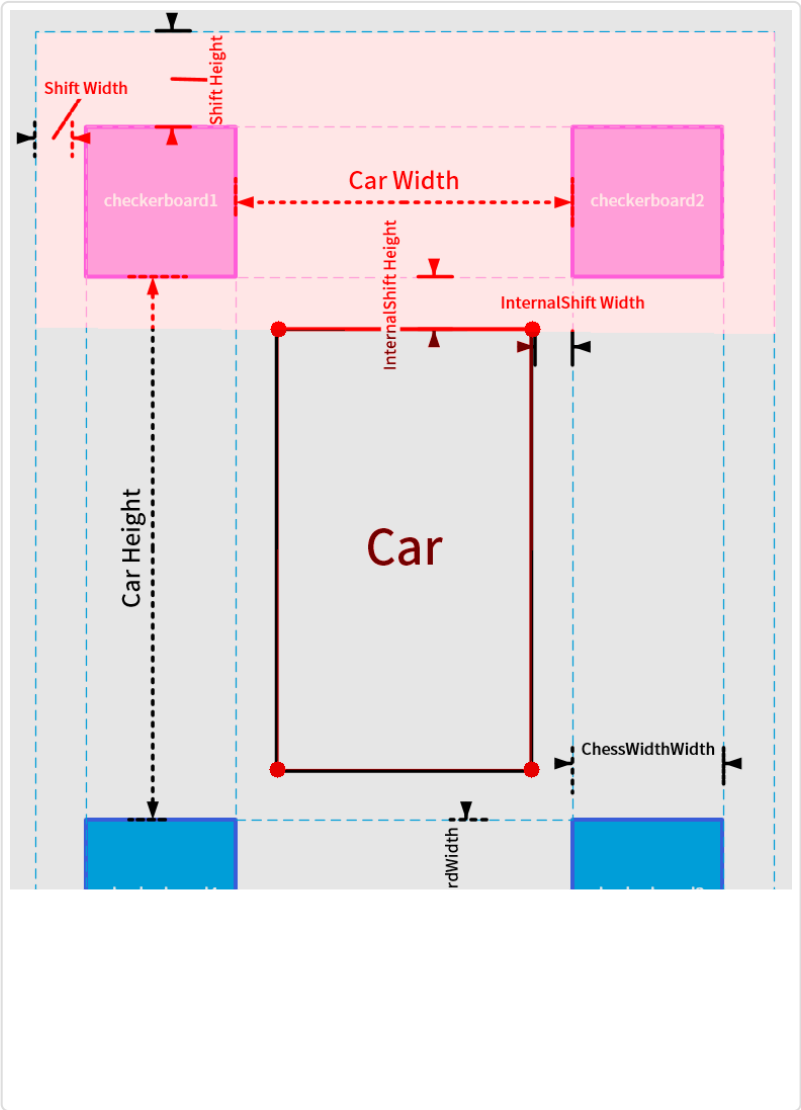
```
totalWidth = 600 + 2 * shiftWidth
totalHeight = 1000 + 2 * shiftHeight
```

- 车辆所在矩形区域的四角 (图中标注的红色圆点)，这四个角点的坐标分别为 `(xl, yt)`，`(xr, yt)`，`(xl, yb)`，`(xr, yb)`（`l` 表示 left，`r` 表示 right，`t` 表示 top，`b` 表示 bottom)。这个矩形区域相机是看不到的，我们会用一张车辆的图标来覆盖此处。

注意这个车辆区域四边的延长线将整个鸟瞰图分为前左 (FL)、前中 (F)、前右 (FR)、左 (L)、右 (R)、后左 (BL)、后中 (B)、后右 (BR) 八个部分，其中 FL (区域 I)、FR (区域 II)、BL (区域 III)、BR (区域 IV) 是相邻相机视野的重合区域，也是我们重点需要进行融合处理的部分。F、R、L、R 四个区域属于每个相机单独的视野，不需要进行融合处理。

以上参数存放在 param_settings.py 中。

设置好参数以后，每个相机的投影区域也就确定了，比如前方相机对应的投影区域如下：



接下来我们需要通过手动选取标志点来获取到地面的投影矩阵。

手动标定获取投影矩阵

首先运行项目中 run_get_projection_maps.py 这个脚本，这个脚本需要你输入如下的参数：

- `-camera`：指定是哪个相机。
- `-scale`：校正后画面的横向和纵向放缩比。
- `-shift`：校正后画面中心的横向和纵向平移距离。

为什么需要 `scale` 和 `shift` 这两个参数呢？这是因为默认的 OpenCV 的校正方式是在鱼眼相机校正后的图像中裁剪出一个 OpenCV "认为" 合适的区域并将其返回，这必然会丢失一部分像素，特别地可能会把我们希望选择的特征点给裁掉。幸运的是 cv2.fisheye.initUndistortRectifyMap 这个函数允许我们再传入一个新的内参矩阵，对校正后

但是裁剪前的画面作一次放缩和平移。你可以尝试调整并选择合适的横向、纵向压缩比和图像中心的位置使得地面上的标志点出现在画面中舒服的位置上，以方便进行标定。

运行

```
python run_get_projection_maps.py -camera front -scale 0.7 0.8 -shift -150
```

后显示前方相机校正后的画面如下：

然后依次点击事先确定好的四个标志点 (顺序不能错！)，得到的效果如下：

注意标志点的设置代码在这里。

这四个点是可以自由设置的，但是你需要在程序中手动修改它们在鸟瞰图中的像素坐标。当你在校正图中点击这四个点时，OpenCV 会根据它们在校正图中的像素坐标和在鸟瞰图中的像素坐标的对应关系计算一个射影矩阵。这里用到的原理就是四点对应确定一个射影变换 (四点对应可以给出八个方程，从而求解出射影矩阵的八个未知量。注意射影矩阵的最后一个分量总是固定为 1)。

如果你不小心点歪了的话可以按 `d` 键删除上一个错误的点。选择好以后点回车，就会显示投影后的效果图:

觉得效果可以的话敲回车，就会将投影矩阵写入 `front.yaml` 中，这个矩阵的名字为 `project_matrix` 。失败的话就按 `q` 退出再来一次。

再比如后面相机的标定如下图所示：

对应的投影图为

对四个相机分别采用此操作，我们就得到了四个相机的鸟瞰图，以及对应的四个投影矩阵。下一步我们的任务是把这四个鸟瞰图拼起来。

鸟瞰图的拼接与平滑

如果你前面操作一切正常的话，运行 run_get_weight_matrices.py 后应该会显示如下的拼接图：

我来逐步介绍它是怎么做到的：

1. 由于相邻相机之间有重叠的区域，所以这部分的融合是关键。如果直接采取两幅图像加权平均 (权重各自为 1/2) 的方式融合的话你会得到类似下面的结果：

   你可以看到由于校正和投影的误差，相邻相机在重合区域的投影结果并不能完全吻合，导致拼接的结果出现乱码和重影。这里的关键在于权重系数应该是随像素变化而变化的，并且是随着像素连续变化。

2. 以左上角区域为例，这个区域是 `front`，`left` 两个相机视野的重叠区域。我们首先将投影图中的重叠部分取出来：

   灰度化并二值化：

   注意这里面有噪点，可以用形态学操作去掉 (不必特别精细，大致去掉即可)：

   至此我们就得到了重叠区域的一个完整 mask。

3. 将 `front`，`left` 图像各自位于重叠区域外部的边界检测出来，这一步是通过先调用 `cv2.findContours` 求出最外围的边界，再用 `cv2.approxPolyDP` 获得逼近的多边形轮廓：

我们把 front 相机减去重叠区域后的轮廓记作 polyA (左上图中蓝色边界)，left 相机减去重叠区域后的轮廓记作 polyB (右上图中绿色边界)。

4. 对重叠区域中的每个像素，利用 cv2.pointPolygonTest 计算其到这两个多边形 polyA 和 polyB 的距离 d_A, d_B，则该像素对应的权值为 $w = d_B^2/(d_A^2 + d_B^2)$，即如果这个像素落在 front 画面内，则它与 polyB 的距离就更远，从而权值更大。

5. 对不在重叠区域内的像素，若其属于 front 相机的范围则其权值为 1，否则权值为 0。于是我们得到了一个连续变化的，取值范围在 0~1 之间的矩阵 G，其灰度图如下：

   将 G 作为权值可得融合后的图像为 `front * G + (1- G) * left` 。

6. 注意由于重叠区域中的像素值是来自两幅图像的加权平均，所以出现在这个区域内的物体会不可避免出现虚影的现象，所以我们需要尽量压缩重叠区域的范围，尽可能只对拼接缝周围的像素计算权值，拼接缝上方的像素尽量使用来自 front 的原像素，拼接缝下方的像素尽量使用来自 back 的原像素。这一步可以通过控制 d_B 的值得到。

7. 我们还漏掉了重要的一步：由于不同相机的曝光度不同，导致不同的区域会出现明暗的亮度差，影响美观。我们需要调整每个区域的亮度，使得整个拼接图像的亮度趋于一致。这一步做法不唯一，自由发挥的空间很大。我查阅了一下网上提到的方法，发现它们要么过于复杂，几乎不可能是实时的；要么过于简单，无法达到理想的效果。特别在上面第二个视频的例子中，由于前方相机的视野被车标遮挡导致感光范围不足，导致其与其它三个相机的画面亮度差异很大，调整起来很费劲。

   一个基本的想法是这样的：每个相机传回的画面有 BGR 三个通道，四个相机传回的画面总共有 12 个通道。我们要计算 12 个系数，将这 12 个系数分别乘到这 12 个通道上，然后再合并起来形成调整后的画面。过亮的通道要调暗一些所以乘的系数小于 1，过暗的通道要调亮一些所以乘的系数大于 1。这些系数可以通过四个画面在四个重合区域内的亮度比值得出，你可以自由设计计算系数的方式，只要满足这个基本原理即可。

   我的实现见这里。感觉就像一段 shader 代码。

   还有一种偷懒的办法是事先计算一个 tone mapping 函数 (比如逐段线性的，或者 AES tone mapping function)，然后强制把所有像素进行转换，这个方法最省力，但是得到的画面色调会与真实场景有较大差距。似乎有的市面产品就是采用的这种方法。

8. 最后由于有些情况下摄像头不同通道的强度不同，还需要进行一次色彩平衡，见下图：

在第二个视频的例子中，画面的颜色偏红，加入色彩平衡后画面恢复了正常。

具体实现的注意事项

1. 多线程与线程同步。本文的两个例子中四个摄像头都不是硬件触发保证同步的，而且即便是硬件同步的，四个画面的处理线程也未必同步，所以需要有一个线程同步机制。这个项目的实现采用的是比较原始的一种，其核心代码如下：

```
class MultiBufferManager(object):

...

def sync(self, device_id):
    # only perform sync if enabled for specified device/stream
    self.mutex.lock()
    if device_id in self.sync_devices:
        # increment arrived count
        self.arrived += 1
        # we are the last to arrive: wake all waiting threads
        if self.do_sync and self.arrived == len(self.sync_devices):
            self.wc.wakeAll()
        # still waiting for other streams to arrive: wait
        else:
            self.wc.wait(self.mutex)
        # decrement arrived count
        self.arrived -= 1
    self.mutex.unlock()
```

这里使用了一个 `MultiBufferManager` 对象来管理所有的线程，每个摄像头的线程在每次循环时会调用它的 `sync` 方法，并通过将计数器加 1 的方法来通知这个对象 "报告，我已做完上次的任务，请将我加入休眠池等待下次任务"。一旦计数器达到 4 就会触发唤醒所有线程进入下一轮的任务循环。

2. 建立查找表 (lookup table) 以加快运算速度。鱼眼镜头的画面需要经过校正、投影、翻转以后才能用于拼接，这三步涉及频繁的图像内存分配和销毁，非常费时间。在我的测试中抓取线程始终稳定在 30fps 多一点左右，但是每个画面的处理线程只有 20 fps 左右。这一步最好是通过预计算一个查找表来加速。你还记得 `cv2.fisheye.initUndistortRectifyMap` 这个函数吗? 它返回的 `mapx, mapy` 就是两个查找表。比如当你指定它返回的矩阵类型为 `cv2.CV_16SC2` 的时候，它返回的 `mapx` 就是一个逐像素的查找表，`mapy` 是一个用于插值平滑的一维数组 (可以扔掉不要)。同理对于 `project_matrix` 也不难获得一个查找表，两个合起来就可以得到一个直接从原始画面到投影画面的查找表 (当然损失了用于插值的信息)。在这个项目中由于采用的是 Python 实现，而 Python 的 `for` 循环效率不高，所以没有采用这种查找表的方式。

3. 四个权重矩阵可以作为 `RGBA` 四个通道压缩到一张图片中，这样存储和读取都很方便。四个重叠区域对应的 mask 矩阵也是如此：

实车运行

你可以在实车上运行 run_live_demo.py 来验证最终的效果。

你需要注意修改相机设备号，以及 OpenCV 打开摄像头的方式。usb 相机可以直接用 `cv2.VideoCapture(i)`（`i` 是 usb 设备号）的方式打开，csi 相机则需要调用 `gstreamer` 打开，对应的代码在这里和这里。

附录：项目各脚本一览

项目中目前的脚本根据执行顺序排列如下：

1. `run_calibrate_camera.py`：用于相机内参标定。
2. `param_settings.py`：用于设置投影区域的各参数。
3. `run_get_projection_maps.py`：用于手动标定获取到地面的投影矩阵。
4. `run_get_weight_matrices.py`：用于计算四个重叠区域对应的权重矩阵以及 mask 矩阵，并显示拼接效果。
5. `run_live_demo.py`：用于在实车上运行的最终版本。