

第 5 讲

相机与图像

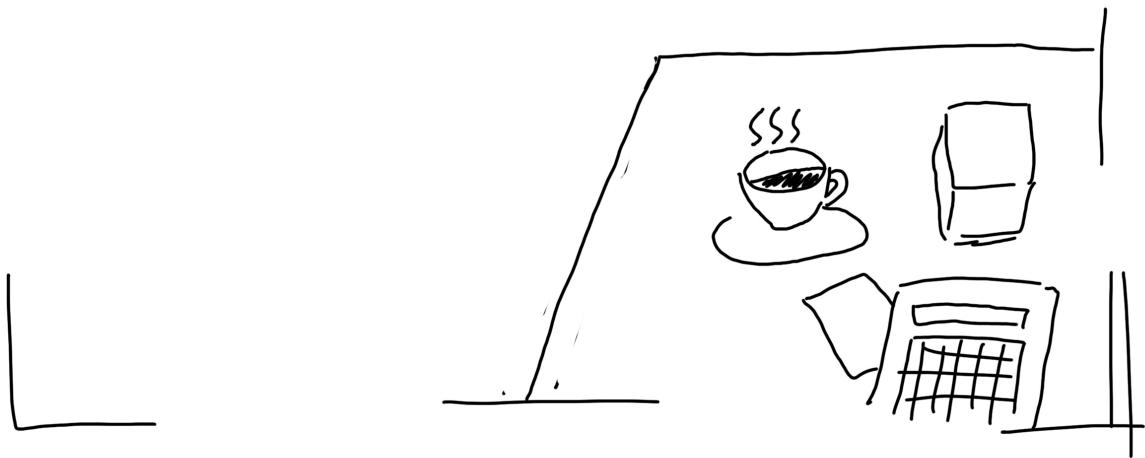
本节目标

1. 理解针孔相机的模型、内参与径向畸变参数。
2. 理解一个空间点是如何投影到相机成像平面的。
3. 掌握 OpenCV 的图像存储与表达方式。
4. 学会基本的摄像头标定方法。

前面两讲中，我们介绍了“机器人如何表示自身位姿”的问题，部分地解释了 SLAM 经典模型中变量的含义和运动方程部分。本讲，我们要讨论“机器人如何观测外部世界”，也就是观测方程部分。而在以相机为主的视觉 SLAM 中，观测主要是指相机成像的过程。

我们在现实生活中能看到大量的照片。在计算机中，一张照片由很多个像素组成，每个像素记录了色彩或亮度的信息。三维世界中的一个物体反射或发出的光线，穿过相机光心后，投影在相机的成像平面上。相机的感光器件接收到光线后，产生了测量值，就得到了像素，形成了我们见到的照片。这个过程能否用数学原理来描述呢？本讲，我们首先讨论相机模型，说明投影关系具体如何描述，相机的内参是什么。同时，简单介绍双目成像与 RGB-D 相机的原理。然后，介绍二维照片像素的基本操作。最后，我们根据内外参数的含义，演示一个点云拼接的实验。

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K(Rp + t)$$



5.1 相机模型

相机将三维世界中的坐标点（单位为米）映射到二维图像平面（单位为像素）的过程能够用一个几何模型进行描述。这个模型有很多种，其中最简单的称为针孔模型。针孔模型是很常用，而且有效的模型，它描述了一束光线通过针孔之后，在针孔背面投影成像的关系。在本书中我们用一个简单的针孔相机模型来对这种映射关系进行建模。同时，由于相机镜头上的透镜的存在，会使得光线投影到成像平面的过程中会产生畸变。因此，我们使用针孔和畸变两个模型来描述整个投影过程。

在本节我们先给出相机的针孔模型，再对透镜的畸变模型进行讲解。这两个模型能够把外部的三维点投影到相机内部成像平面，构成了相机的内参数。

5.1.1 针孔相机模型

在初中物理课堂上，我们可能都见过一个蜡烛投影实验：在一个暗箱的前方放着一支点燃的蜡烛，蜡烛的光透过暗箱上的一个小孔投影在暗箱的后方平面上，并在这个平面上形成了一个倒立的蜡烛图像。在这个过程中，小孔模型能够把三维世界中的蜡烛投影到一个二维成像平面。同理，我们可以用这个简单的模型来解释相机的成像过程。如图 5-1 所示。

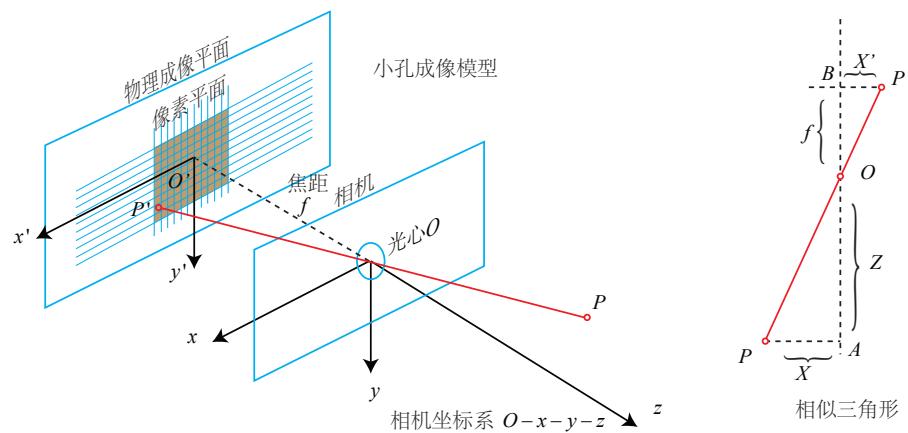


图 5-1 针孔相机模型

现在来对这个简单的针孔模型进行几何建模。设 $O - x - y - z$ 为相机坐标系，习惯上我们让 z 轴指向相机前方， x 向右， y 向下。 O 为摄像机的光心，也是针孔模型中的针孔。现实世界的空间点 P ，经过小孔 O 投影之后，落在物理成像平面 $O' - x' - y'$ 上，成

像点为 P' 。设 P 的坐标为 $[X, Y, Z]^T$, P' 为 $[X', Y', Z']^T$, 并且设物理成像平面到小孔的距离为 f (焦距)。那么, 根据三角形相似关系, 有:

$$\frac{Z}{f} = -\frac{X}{X'} = -\frac{Y}{Y'}. \quad (5.1)$$

其中负号表示成的像是倒立的。为了简化模型, 我们把可以成像平面对称到相机前方, 和三维空间点一起放在摄像机坐标系的同一侧, 如图 5-2 中间的样子所示。这样做可以把公式中的负号去掉, 使式子更加简洁:

$$\frac{Z}{f} = \frac{X}{X'} = \frac{Y}{Y'}. \quad (5.2)$$

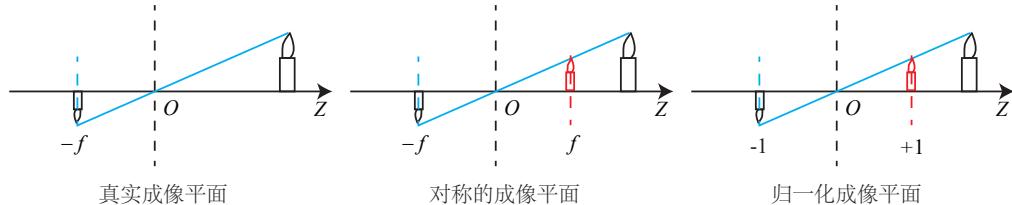


图 5-2 真实成像平面, 对称成像平面, 归一化成像平面的图示。

整理得:

$$\begin{aligned} X' &= f \frac{X}{Z} \\ Y' &= f \frac{Y}{Z} \end{aligned} \quad (5.3)$$

读者可能要问, 为什么我们可以看似随意地把成像平面挪到前方呢? 这只是我们处理真实世界与相机投影的数学手段, 并且, 大多数相机输出的图像并不是倒像——相机自身的软件会帮你翻转这张图像, 所以你看到的一般是正着的像, 也就是对称的成像平面上的像。所以, 尽管从物理原理来说, 小孔成像应该是倒像, 但由于我们对图像作了预处理, 所以理解成在对称平面上的像, 并不会带来什么坏处。于是, 在不引起歧义的情况下, 我们也不加限制地称后一种情况为针孔模型。

式 (5.3) 描述了点 P 和它的像之间的空间关系。不过, 在相机中, 我们最终获得的是一个个的像素, 这需要在成像平面上对像进行采样和量化。为了描述传感器将感受到的光线转换成图像像素的过程, 我们设在物理成像平面上固定着一个像素平面 $o-u-v$ 。我们在像素平面得到了 P' 的像素坐标: $[u, v]^T$ 。

像素坐标系^①通常的定义方式是：原点 o' 位于图像的左上角， u 轴向右与 x 轴平行， v 轴向下与 y 轴平行。像素坐标系与成像平面之间，相差了一个缩放和一个原点的平移。我们设像素坐标在 u 轴上缩放了 α 倍，在 v 上缩放了 β 倍。同时，原点平移了 $[c_x, c_y]^T$ 。那么， P' 的坐标与像素坐标 $[u, v]^T$ 的关系为：

$$\begin{cases} u = \alpha X' + c_x \\ v = \beta Y' + c_y \end{cases}. \quad (5.4)$$

代入式 (5.3) 并把 αf 合并成 f_x ，把 βf 合并成 f_y ，得：

$$\begin{cases} u = f_x \frac{X}{Z} + c_x \\ v = f_y \frac{Y}{Z} + c_y \end{cases}. \quad (5.5)$$

其中， f 的单位为米， α, β 的单位为像素每米，所以 f_x, f_y 的单位为像素。把该式写成矩阵形式，会更加简洁，不过左侧需要用到齐次坐标：

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \frac{1}{Z} \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \triangleq \frac{1}{Z} \mathbf{KP}. \quad (5.6)$$

我们按照传统的习惯，把 Z 挪到左侧：

$$Z \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \triangleq \mathbf{KP}. \quad (5.7)$$

该式中，我们把中间的量组成的矩阵称为相机的内参数矩阵（Camera Intrinsics） \mathbf{K} 。通常认为，相机的内参在出厂之后是固定的，不会在使用过程中发生变化。有的相机生产厂商会告诉你相机的内参，而有时需要你自己确定相机的内参，也就是所谓的标定。鉴于标定算法业已成熟，且网络上能找到大量的标定教学，我们在此就不介绍了。

除了内参之外，自然还有相对的外参。考虑到在式 (5.6) 中，我们使用的是 P 在相机坐标系下的坐标。由于相机在运动，所以 P 的相机坐标应该是它的世界坐标（记为 \mathbf{P}_w ），根据相机的当前位姿，变换到相机坐标系下的结果。相机的位姿由它的旋转矩阵 \mathbf{R} 和平

^①或图像坐标系，见本讲第二节。

移向量 \mathbf{t} 来描述。那么有：

$$Z\mathbf{P}_{uv} = Z \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \mathbf{K}(\mathbf{R}\mathbf{P}_w + \mathbf{t}) = \mathbf{K}\mathbf{T}\mathbf{P}_w. \quad (5.8)$$

注意后一个式子隐含了一次齐次坐标到非齐次坐标的转换（你能看出来吗？）。它描述了 P 的世界坐标到像素坐标的投影关系。其中，相机的位姿 \mathbf{R}, \mathbf{t} 又称为相机的外参数 (Camera Extrinsics)。相比于不变的内参，外参会随着相机运动发生改变，同时也是 SLAM 中待估计的目标，代表着机器人的轨迹。

上式两侧都是齐次坐标。因为齐次坐标乘上非零常数后表达同样的含义，所以可以简单地把 Z 去掉：

$$\mathbf{P}_{uv} = \mathbf{K}\mathbf{T}\mathbf{P}_w. \quad (5.9)$$

但这样等号意义就变了，成为在齐次坐标下相等的概念，相差了一个非零常数。为了避免麻烦，我们还是从传统意义下来定义书写等号。

我们还是提一下隐含着的齐次到非齐次的变换吧。可以看到，右侧的 $\mathbf{T}\mathbf{P}_w$ 表示把一个世界坐标系下的齐次坐标，变换到相机坐标系下。为了使它与 \mathbf{K} 相乘，需要取它的前三维组成向量——因为 $\mathbf{T}\mathbf{P}_w$ 最后一维为 1。此时，对于这个三维向量，我们还可以按照齐次坐标的方式，把最后一维进行归一化处理，得到了 P 在相机归一化平面上的投影：

$$\tilde{\mathbf{P}}_c = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = (\mathbf{T}\mathbf{P}_w)_{(1:3)}, \quad \mathbf{P}_c = \begin{bmatrix} X/Z \\ Y/Z \\ 1 \end{bmatrix}. \quad (5.10)$$

这时 \mathbf{P}_c 可以看成一个二维的齐次坐标，称为归一化坐标。它位于相机前方 $z = 1$ 处的平面上。该平面称为归一化平面。由于 \mathbf{P}_c 经过内参之后就得到了像素坐标，所以我们可以把像素坐标 $[u, v]^T$ ，看成对归一化平面上的点进行量化测量的结果。

至此，针孔相机的成像模型我们就讲清楚了。

5.1.2 畸变

为了获得好的成像效果，我们在相机的前方加了透镜。透镜的加入对成像过程中光线的传播会产生新的影响：一是透镜自身的形状对光线传播的影响，二是在机械组装过程中，透镜和成像平面不可能完全平行，这也会使得光线穿过透镜投影到成像面时的位置发生变

化。

由透镜形状引起的畸变称之为径向畸变。在针孔模型中，一条直线投影到像素平面上还是一条直线。可是，在实际拍摄的照片中，摄像机的透镜往往使得真实环境中的一条直线在图片中变成了曲线^①。越靠近图像的边缘，这种现象越明显。由于实际加工制作的透镜往往是中心对称的，这使得不规则的畸变通常径向对称。它们主要分为两大类，桶形畸变和枕形畸变，如图5-3所示。

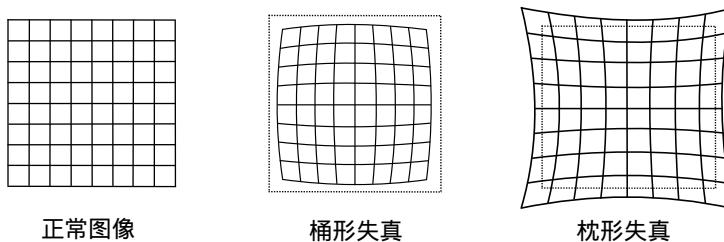


图 5-3 径向畸变的两种类型。

桶形畸变是由于图像放大率随着离光轴的距离增加而减小，而枕形畸变却恰好相反。在这两种畸变中，穿过图像中心和光轴有交点的直线还能保持形状不变。

除了透镜的形状会引入径向畸变外，在相机的组装过程中由于不能使得透镜和成像面严格平行也会引入切向畸变。如图 5-4 所示。

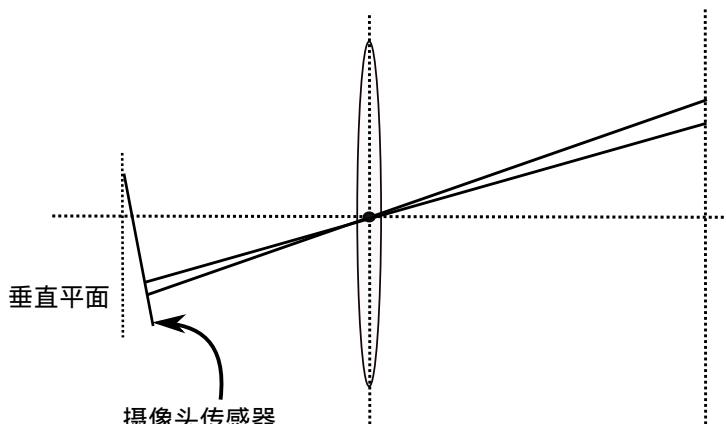


图 5-4 切向畸变来源示意图。

为更好地理解径向畸变和切向畸变，我们用更严格的数学形式对两者进行描述。我们知道平面上的任意一点 p 可以用笛卡尔坐标表示为 $[x, y]^T$ ，也可以把它写成极坐标的形式

^①是的，它不再直了，而是弯的。如果往里弯，称为桶形失真；往外弯则是枕形失真。

$[r, \theta]^T$, 其中 r 表示点 p 离坐标系原点的距离, θ 表示和水平轴的夹角。径向畸变可看成坐标点沿着长度方向发生了变化 δr , 也就是其距离原点的长度发生了变化。切向畸变可以看成坐标点沿着切线方向发生了变化, 也就是水平夹角发生了变化 $\delta\theta$ 。

对于径向畸变, 无论是桶形畸变还是枕形畸变, 由于它们都是随着离中心的距离增加而增加。我们可以用一个多项式函数来描述畸变前后的坐标变化: 这类畸变可以用和距中心距离有关的二次及高次多项式函数进行纠正:

$$\begin{aligned}x_{corrected} &= x(1 + k_1r^2 + k_2r^4 + k_3r^6) \\y_{corrected} &= y(1 + k_1r^2 + k_2r^4 + k_3r^6)\end{aligned}\quad (5.11)$$

其中 $[x, y]^T$ 是未纠正的点的坐标, $[x_{corrected}, y_{corrected}]^T$ 是纠正后的点的坐标, 注意它们都是归一化平面上的点, 而不是像素平面上的点。

在式 (5.11) 描述的纠正模型中, 对于畸变较小的图像中心区域, 畸变纠正主要是 k_1 起作用。而对于畸变较大的边缘区域主要是 k_2 起作用。普通摄像头用这两个系数就能很好的纠正径向畸变。对畸变很大的摄像头, 比如鱼眼镜头, 可以加入 k_3 畸变项对畸变进行纠正。

另一方面, 对于切向畸变, 可以使用另外的两个参数 p_1, p_2 来进行纠正:

$$\begin{aligned}x_{corrected} &= x + 2p_1xy + p_2(r^2 + 2x^2) \\y_{corrected} &= y + p_1(r^2 + 2y^2) + 2p_2xy\end{aligned}\quad (5.12)$$

因此, 联合式 (5.11) 和式 (5.12), 对于相机坐标系中的一点 $P(X, Y, Z)$, 我们能够通过五个畸变系数找到这个点在像素平面上的正确位置:

1. 将三维空间点投影到归一化图像平面。设它的归一化坐标为 $[x, y]^T$ 。
2. 对归一化平面上的点进行径向畸变和切向畸变纠正。

$$\begin{cases}x_{corrected} = x(1 + k_1r^2 + k_2r^4 + k_3r^6) + 2p_1xy + p_2(r^2 + 2x^2) \\y_{corrected} = y(1 + k_1r^2 + k_2r^4 + k_3r^6) + p_1(r^2 + 2y^2) + 2p_2xy\end{cases}\quad (5.13)$$

3. 将纠正后的点通过内参数矩阵投影到像素平面, 得到该点在图像上的正确位置。

$$\begin{cases}u = f_x x_{corrected} + c_x \\v = f_y y_{corrected} + c_y\end{cases}\quad (5.14)$$

在上面的纠正畸变的过程中, 我们使用了五个畸变项。实际应用中, 可以灵活选择纠

正模型，比如只选择 k_1, p_1, p_2 这三项等。

在这一节中，我们对相机的成像过程使用针孔模型进行了建模，也对透镜引起的径向畸变和切向畸变进行了描述。实际的图像系统中，学者们提出了有很多其他的模型，比如相机的仿射模型和透视模型等，同时也存在很多其他类型的畸变。考虑到视觉 SLAM 中，一般都使用普通的摄像头，针孔模型以及径向畸变和切向畸变模型已经足够。因此，我们不再对其它模型进行描述。

值得一提的是，存在两种去畸变处理（Undistort，或称畸变校正）做法。我们可以选择先对整张图像进行去畸变，得到去畸变后的图像，然后讨论此图像上的点的空间位置。或者，我们也可以先考虑图像中的某个点，然后按照去畸变方程，讨论它去畸变后的空间位置。二者都是可行的，不过前者在视觉 SLAM 中似乎更加常见一些。所以，当一个图像去畸变之后，我们就可以直接用针孔模型建立投影关系，而不用考虑畸变了。因此，在后文的讨论中，我们可以直接假设图像已经进行了去畸变处理。

最后，我们小结一下单目相机的成像过程：

1. 首先，世界坐标系下有一个固定的点 P ，世界坐标为 \mathbf{P}_w ；
2. 由于相机在运动，它的运动由 \mathbf{R}, \mathbf{t} 或变换矩阵 $\mathbf{T} \in SE(3)$ 描述。 P 的相机坐标为：

$$\tilde{\mathbf{P}}_c = \mathbf{R}\mathbf{P}_w + \mathbf{t}.$$
3. 这时的 $\tilde{\mathbf{P}}_c$ 仍有 X, Y, Z 三个量，把它们投影到归一化平面 $Z = 1$ 上，得到 P 的归一化相机坐标：

$$\mathbf{P}_c = [X/Z, Y/Z, 1]^T$$
^①。
4. 最后， P 的归一化坐标经过内参后，对应到它的像素坐标：

$$\mathbf{P}_{uv} = \mathbf{K}\mathbf{P}_c.$$

综上所述，我们一共谈到了四种坐标：世界、相机、归一化相机和像素坐标。请读者理清它们的关系，它反映了整个成像的过程。

5.1.3 双目相机模型

针孔相机模型描述了单个相机的成像模型。然而，仅根据一个像素，我们是无法确定这个空间点的具体位置的。这是因为，从相机光心到归一化平面连线上的所有点，都可以投影至该像素上。只有当 P 的深度确定时（比如通过双目或 RGB-D 相机），我们才能确切地知道它的空间位置。

测量像素距离（或深度）的方式有很多种，像人眼就可以根据左右眼看到的景物差异（或称视差）来判断物体与我们的距离。双目相机的原理亦是如此。通过同步采集左右相机

^①注意到 Z 可能小于 1，说明该点位于归一化平面后面，它可能不会在相机平面上成像，实践当中要检查一次。

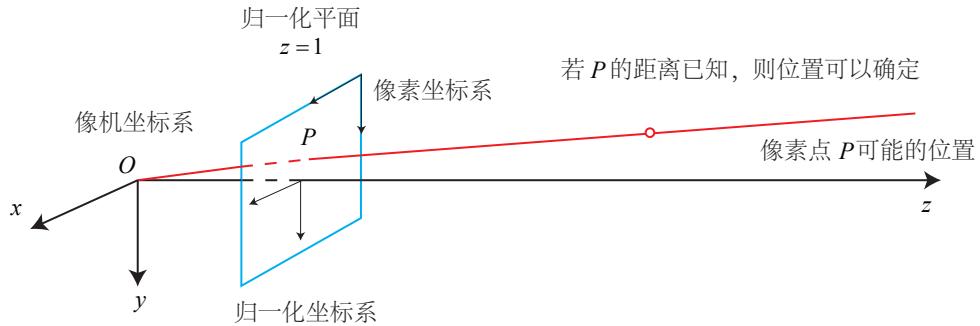
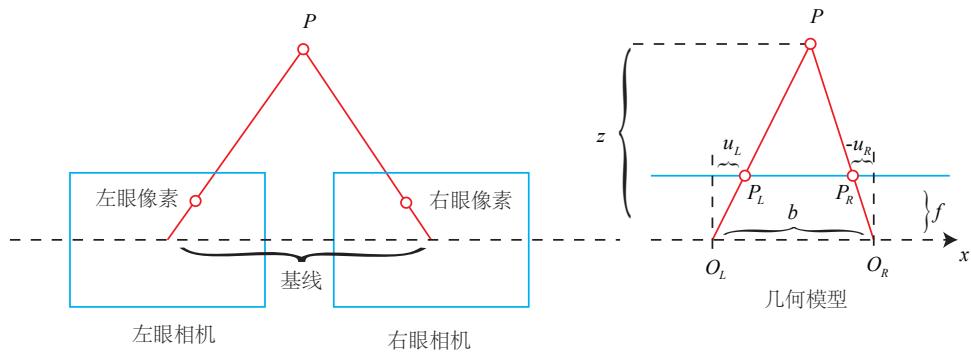


图 5-5 像素点可能存在的位置。

图 5-6 双目相机的成像模型。 O_L, O_R 为左右光圈中心, 蓝色框为成像平面, f 为焦距。 u_L 和 u_R 为成像平面的坐标。请注意按照图中坐标定义, u_R 应该是负数, 所以图中标出的距离为 $-u_R$ 。

的图像，计算图像间视差，来估计每一个像素的深度。下面我们简单讲讲双目相机的成像原理（图 5-6）。

双目相机一般由左眼和右眼两个水平放置的相机组成。当然也可以做成上下两个目^①，但我们见到的主流双目都是做成左右的。在左右双目的相机中，我们可以把两个相机都看作针孔相机。它们是水平放置的，意味两个相机的光圈中心都位于 x 轴上。它们的距离称为双目相机的基线（Baseline，记作 b ），是双目的重要参数。

现在，考虑一个空间点 P ，它在左眼和右眼各成一像，记作 P_L, P_R 。由于相机基线的存在，这两个成像位置是不同的。理想情况下，由于左右相机只有在 x 轴上有位移，因此 P 的像也只在 x 轴（对应图像的 u 轴）上有差异。我们记它在左侧的坐标为 u_L ，右侧坐标为 u_R 。那么，它们的几何关系如图 5-6 右侧所示。根据三角形 $P - P_L - P_R$ 和 $P - O_L - O_R$ 的相似关系，有：

$$\frac{z - f}{z} = \frac{b - u_L + u_R}{b}. \quad (5.15)$$

稍加整理，得：

$$z = \frac{fb}{d}, \quad d = u_L - u_R. \quad (5.16)$$

这里 d 为左右图的横坐标之差，称为视差（Disparity）。根据视差，我们可以估计一个像素离相机的距离。视差与距离成反比：视差越大，距离越近^②。同时，由于视差最小为一个像素，于是双目的深度存在一个理论上的最大值，由 fb 确定。我们看到，当基线越长时，双目最大能测到的距离就会变远；反之，小型双目器件则只能测量很近的距离。

虽然由视差计算深度的公式很简洁，但视差 d 本身的计算却比较困难。我们需要确切地知道左眼图像某个像素出现在右眼图像的哪一个位置（即对应关系），这件事亦属于“人类觉得容易而计算机觉得困难”的事务。当我们想计算每个像素的深度时，其计算量与精度都将成为问题，而且只有在图像纹理变化丰富的地方才能计算视差。由于计算量的原因，双目深度估计仍需要使用 GPU 或 FPGA 来计算。这将在十三章中提到。

5.1.4 RGB-D 相机模型

相比于双目相机通过视差计算深度的方式，RGB-D 相机的做法更为“主动”一些，它能够主动测量每个像素的深度。目前的 RGB-D 相机按原理可分为两大类：

1. 通过红外结构光（Structured Light）来测量像素距离的。例子有 Kinect 1 代、Project Tango 1 代、Intel RealSense 等；

^① 那样外观会有些奇特。

^② 读者可以自己用眼睛模拟一下。

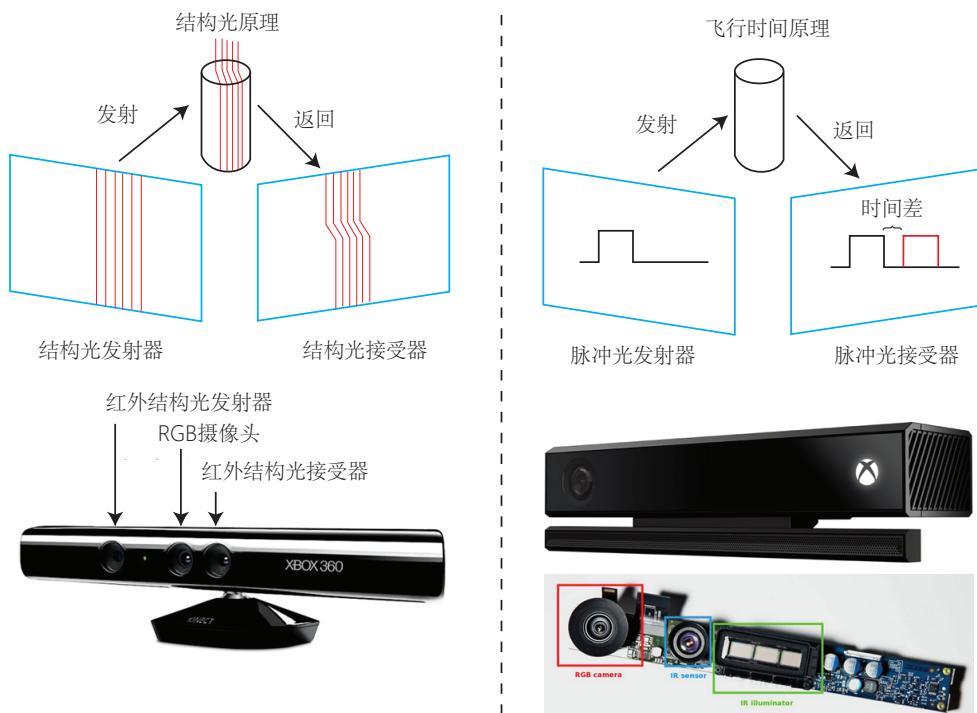


图 5-7 RGB-D 相机原理示意图

- 通过飞行时间法 (Time-of-flight, ToF) 原理测量像素距离的。例子有 Kinect 2 代和一些现有的 ToF 传感器等。

无论是结构光还是 ToF，RGB-D 相机都需要向探测目标发射一束光线（通常是红外光）。在结构光原理中，相机根据返回的结构光图案，计算物体离自身的距离。而在 ToF 中，相机向目标发射脉冲光，然后根据发送到返回之间的光束飞行时间，确定物体离自身的距离。ToF 原理和激光传感器十分相似，不过激光是通过逐点扫描来获取距离，而 ToF 相机则可以获得整个图像的像素深度，这也正是 RGB-D 相机的特点。所以，如果你把一个 RGB-D 相机拆开，通常会发现除了普通的摄像头之外，至少会有一个发射器和一个接收器。

在测量深度之后，RGB-D 相机通常按照生产时的各个相机摆放位置，自己完成深度与彩色图像素之间的配对，输出一一对应的彩色图和深度图。我们可以在同一个图像位置，读取到色彩信息和距离信息，计算像素的 3D 相机坐标，生成点云 (Point Cloud)。对

RGB-D 数据，既可以在图像层面进行处理，亦可在点云层面处理。本讲的第二个实验将演示 RGB-D 相机的点云构建过程。

RGB-D 相机能够实时地测量每个像素点的距离。但是，由于这种发射-接受的测量方式，使得它使用范围比较受限。用红外进行深度值测量的 RGB-D 相机，容易受到日光或其他传感器发射的红外光干扰，因此不能在室外使用，同时使用多个时也会相互干扰。对于透射材质的物体，因为接受不到反射光，所以无法测量这些点的位置。此外，RGB-D 相机在成本、功耗方面，都有一些劣势。

5.2 图像

相机加上镜头，把三维世界中的信息转换成了一个由像素组成的照片，随后存储在计算机中，作为后续处理的数据来源。在数学中，图像可以用一个矩阵来描述；而在计算机中，它们占据一段连续的磁盘或内存空间，可以用二维数组来表示。这样一来，程序就不必区别它们处理的是一个数值矩阵，还是有实际意义的图像了。

本节，我们将介绍计算机图像处理的一些基本操作。特别地，通过 OpenCV 中图像数据的处理，理解计算机中处理图像的常见步骤，为后续章节打下基础。

5.2.1 计算机中图像的表示

我们从最简单的图像——灰度图开始说起。在一张灰度图中，每个像素位置 (x, y) 对应到一个灰度值 I ，所以一张宽度为 w ，高度为 h 的图像，数学形式可以记成一个矩阵：

$$\mathbf{I}(x, y) \in \mathbb{R}^{w \times h}.$$

然而，计算机并不能表达整个实数空间，所以我们只能在某个范围内，对图像进行量化。例如常见的灰度图中，我们用 0-255 之间整数（即一个 `unsigned char`，一个字节）来表达图像的灰度大小。那么，一张宽度为 640，高度为 480 分辨率的灰图度就可以这样表示：

```
1 unsigned char image[480][640];
```

为什么这里的二维数组是 480×640 呢？因为在程序中，图像以一个二维数组形式存储。它的第一个下标则是指数组的行，而第二个下标是列。在图像中，数组的行数对应图像的高度，而列数对应图像的宽度。

下面我们来考察这个图像的内容。图像自然是由像素组成的。当我们访问某一个像素时，需要指明它所处的坐标，请看图 5-8。

图 5-8 左边显示了传统像素坐标系的定义方式。一个像素坐标系原点位于图像的左上角， X 轴向右， Y 轴向下（也就是前面所说的 u, v 坐标）。如果它还有第三个轴的话，根

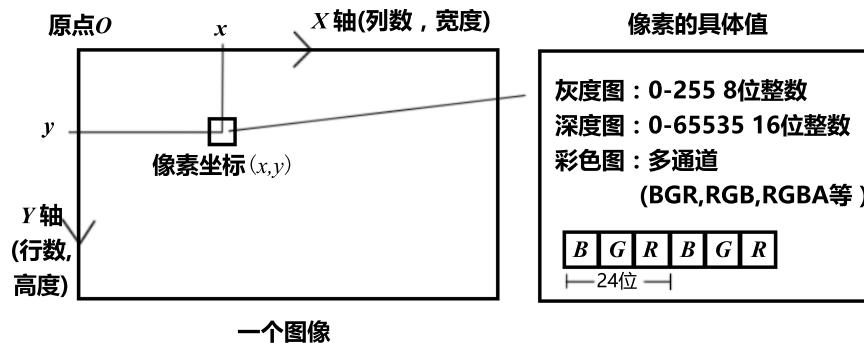


图 5-8 图像坐标示意图。

据右手法则， Z 轴应该是向前的。这种定义方式是与相机坐标系一致的。我们平时说的图像的宽度和列数，对应着 X 轴；而图像的行数或高度，则对应着它的 Y 轴。

根据这种定义方式，如果我们讨论一个位于 x, y 处的像素，那么它在程序中的访问方式应该是：

```
1 unsigned char pixel = image[y][x];
```

它对应着灰度值 $I(x, y)$ 的读数。请注意这里的 x 和 y 的顺序。虽然我们有些繁琐地向读者讨论坐标系的问题，但是像这种下标顺序的错误，会是新手在调试过程中经常碰到的，又具有一定隐蔽性的错误之一。如果你在写程序时不慎调换了 x, y 的坐标，编译器无法提供任何信息，而你能看到的只是程序运行中的一个越界错误而已。

一个灰度像素可以用八位整数记录，也就是一个 0-255 之间的值。当我们要记录的信息更多时，一个字节恐怕就不够了。例如，在 RGB-D 相机的深度图中，记录了各个像素离相机的距离。这个距离通常是毫米为单位，而 RGB-D 相机的量程通常在十几米范围左右，超过了 255 的最大值范围。这时，人们会采用十六位整数（C++ 中的 `unsigned short`）来记录一个深度图的信息，也就是位于 0 至 65536 之间的值。换算成毫米的话，最大可以表示 65 米，足够一个 RGB-D 相机使用了。

彩色图像的表示则需要通道（channel）的概念。在计算机中，我们用红色，绿色和蓝色这三种颜色的组合来表达任意一种色彩。于是对于每一个像素，就要记录它的 R,G,B 三个数值，每一个数值就称为一个通道。例如，最常见的彩色图像有三个通道，每个通道都由 8 位整数表示。在这种规定下，一个像素占据了 24 位空间。

通道的数量，顺序都是可以自由定义的。在 OpenCV 的彩色图像中，通道的默认顺序是 B,G,R。也就是说，当我们得到一个 24 位的像素时，前 8 位表示蓝色数值，中间 8 位

为绿色，最后 8 位为红色。同理，亦可使用 R,G,B 的顺序表示一个彩色图。如果我们还想表达图像的透明度时，就使用 R,G,B,A 四个通道来表示它。

5.3 实践：图像的存取与访问

下面我们通过一个演示程序，来理解在 OpenCV 中，图像是如何存取，我们又是如何访问其中的像素的。

5.3.1 安装 OpenCV

OpenCV^①提供了大量的开源图像算法，是计算机视觉中使用极广的图像处理算法库。本书也使用 OpenCV 做基本的图像处理。在使用之前，我建议你从源代码安装它。在 ubuntu 下，你可以选择从源代码安装和只安装库文件两种方式：

1. 从源代码安装，是指从 OpenCV 网站下载所有的 OpenCV 源代码。并在你的机器上编译安装，以便使用。好处是可以选择的版本比较丰富，而且能看到源代码，不过需要花费一些编译时间；
2. 只安装库文件，是指通过 Ubuntu 来安装由 Ubuntu 社区人员已经编译好的库文件，这样你就无需重新编译一遍。

由于我们使用较新版本的 OpenCV，所以你必须从源代码来安装它。一来，你可以调整一些编译选项，来匹配你的编程环境（例如需不需要 GPU 加速等）；再者，源代码安装可以使用一些额外的功能。OpenCV 目前维护了两个主要版本，分为 OpenCV 2.4 系列和 OpenCV 3 系列。本书使用 OpenCV 3 系列。

由于 OpenCV 工程比较大，我们就不放在本书的 3rdparty 下了。请读者从 <http://opencv.org/downloads.html> 中下载，选择 OpenCV for Linux 版本即可。你会获得一个像 opencv-3.1.0.zip 这样的压缩包。将它解压到任意目录下，我们发现 OpenCV 亦是一个 cmake 工程。

在编译之前，先来安装 OpenCV 的依赖项：

```
1 sudo apt-get install build-essential libgtk2.0-dev libvtk5-dev libjpeg-dev libtiff4-dev libjasper-dev  
libopenexr-dev libtbb-dev
```

事实上 OpenCV 的依赖项很多，缺少某些编译项会影响它的部分功能（不过我们也不会用到所有功能）。OpenCV 会在 cmake 阶段检查依赖项是否已安装，并调整自己的功能。如果你的电脑上有 GPU 并且安装了相关依赖项，OpenCV 也会把 GPU 加速打开。不过对于本书，上边那些依赖项就足够了。

^①官方主页：<http://opencv.org>

随后的编译安装和普通的 cmake 工程一样,请在 make 之后,调用 sudo make install 将 OpenCV 安装到你的机器上(而不是仅仅编译它)。视你的机器配置,这个编译过程大概需要二十分钟到一个小时不等。如果你的 CPU 比较强力,可以使用“make -j4”这样的命令,调用多个线程进行编译(-j 后边的参数就是使用的线程数量)。在安装之后,OpenCV 默认存储到你的/usr/local 目录下。你可以去寻找 opencv 头文件与库文件的安装位置,看看它们都在哪里。另外,如果你之前已经安装了 OpenCV 2 系列,我建议你把 OpenCV 3 安装到不同的地方——想想这应该如何操作。

5.3.2 操作 OpenCV 图像

接下来,我们通过一个例程熟悉一下 OpenCV 对图像的操作。

slambook/ch5/imageBasics.cpp :

```
1 #include <iostream>
2 #include <chrono>
3 using namespace std;
4
5 #include <opencv2/core/core.hpp>
6 #include <opencv2/highgui/highgui.hpp>
7
8 int main ( int argc, char** argv )
9 {
10     // 读取 argv[1] 指定的图像
11     cv::Mat image;
12     image = cv::imread ( argv[1] ); // cv::imread 函数读取指定路径下的图像
13     // 判断图像文件是否正确读取
14     if ( image.data == nullptr ) // 数据不存在, 可能是文件不存在
15     {
16         cerr<<"文件"<<argv[1]<<"不存在."<<endl;
17         return 0;
18     }
19
20     // 文件顺利读取, 首先输出一些基本信息
21     cout<<"图像宽为"<<image.cols<<, 高为"<<image.rows<<, 通道数为"<<image.channels()<<endl;
22     cv::imshow ( "image", image ); // 用 cv::imshow 显示图像
23     cv::waitKey ( 0 ); // 暂停程序, 等待一个按键输入
24     // 判断 image 的类型
25     if ( image.type() != CV_8UC1 && image.type() != CV_8UC3 )
26     {
27         // 图像类型不符合要求
28         cout<<"请输入一张彩色图或灰度图."<<endl;
29         return 0;
30     }
31 }
```

```
32 // 遍历图像，请注意以下遍历方式亦可使用于随机访问
33 // 使用 std::chrono 来给算法计时
34 chrono::steady_clock::time_point t1 = chrono::steady_clock::now();
35 for ( size_t y=0; y<image.rows; y++ )
36 {
37     for ( size_t x=0; x<image.cols; x++ )
38     {
39         // 访问位于 x,y 处的像素
40         // 用 cv::Mat::ptr 获得图像的行指针
41         unsigned char* row_ptr = image.ptr<unsigned char>( y ); // row_ptr 是第 y 行的头指针
42         unsigned char* data_ptr = &row_ptr[ x*image.channels() ]; // data_ptr 指向待访问的像素数据
43         // 输出该像素的每个通道，如果是灰度图就只有一个通道
44         for ( int c = 0; c != image.channels(); c++ )
45         {
46             unsigned char data = data_ptr[c]; // data 为 I(x,y) 第 c 个通道的值
47         }
48     }
49 }
50 chrono::steady_clock::time_point t2 = chrono::steady_clock::now();
51 chrono::duration<double> time_used = chrono::duration_cast<chrono::duration<double>>( t2-t1 );
52 cout<<"遍历图像用时："<<time_used.count()<<" 秒。"<<endl;
53
54 // 关于 cv::Mat 的拷贝
55 // 直接赋值并不会拷贝数据
56 cv::Mat image_another = image;
57 // 修改 image_another 会导致 image 发生变化
58 image_another( cv::Rect( 0,0,100,100 ) ).setTo( 0 ); // 将左上角 100*100 的块置零
59 cv::imshow( "image", image );
60 cv::waitKey( 0 );
61
62 // 使用 clone 函数来拷贝数据
63 cv::Mat image_clone = image.clone();
64 image_clone( cv::Rect( 0,0,100,100 ) ).setTo( 255 );
65 cv::imshow( "image", image );
66 cv::imshow( "image_clone", image_clone );
67 cv::waitKey( 0 );
68
69 // 其他图像操作请参见 OpenCV 官方文档，查询每个函数的调用方法。
70 cv::destroyAllWindows();
71 return 0;
72 }
```

在该例程中，我们演示了以下几个操作：图像读取、显示、像素遍历、拷贝、赋值等。大部分的注解已写在代码里面。编译该程序时，你需要在 CMakeLists.txt 添加 OpenCV 的头文件，然后把程序链接到库文件上。同时，由于我们使用了 C++ 11 标准（如 nullptr 和 chrono），还需要设置一下编译器：

```
1 # 添加 c++ 11 标准支持
2 set( CMAKE_CXX_FLAGS "-std=c++11" )
```

```
3 # 寻找 OpenCV 库
4 find_package( OpenCV REQUIRED )
5 # 添加头文件
6 include_directories( ${OpenCV_INCLUDE_DIRS} )
7
8
9 add_executable( imageBasics imageBasics.cpp )
10 # 链接 OpenCV 库
11 target_link_libraries( imageBasics ${OpenCV_LIBS} )
```

关于代码，我们给出几点注解：

1. 程序从 argv[1]，也就是命令行的第一个参数中读取图像位置。我们为读者准备了一张图像（ubuntu.png，一张 ubuntu 的壁纸，希望你喜欢）供测试使用。因此，编译之后，使用如下命令调用此程序：

```
1 build/image_basics ubuntu.png
```

如果在 Kdevelop 中调用此程序，请务必确保把参数同时给它。这可以在启动项中配置。

2. 程序的 10 到 17 行，使用 cv::imread 函数读取图像，并把图像和基本信息显示出来。
3. 在例程的 32 行至 52 行，我们遍历了程序当中的所有像素，并计算了整个循环的时间。请注意像素的遍历方式并不是唯一的，而且例程给出的方式也不是最高效的。OpenCV 提供了迭代器，你能够通过迭代器遍历图像的像素。或者，cv::Mat::data 提供了指向图像数据开头的指针，你可以直接通过该指针，自行计算偏移量，然后得到像素的实际内存位置。例程给出的方式是为了便于读者理解图像的结构。

在我的机器上（虚拟机），遍历这张图像用时大约 12.74 毫秒左右。你可以对比一下自己机器上的速度。不过，我们使用的是 cmake 默认的 debug 模式，如果使用 release 模式会快很多。

4. OpenCV 提供了许多对图像进行操作的函数，我们在此不一一列举，否则本书就会变成 OpenCV 操作手册了。例程给出了较为常见的读取、显示操作以及复制图像中可能陷入的深拷贝误区。在编程过程中，读者还会碰到图像的旋转、插值等操作，这时你应该自行查阅函数对应的文档，以了解它们的原理与使用方式。

应该指出，OpenCV 并不是唯一的图像库，它是许多图像库里，使用范围较广泛之一。不过，多数图像库对图像的表达是大同小异的。我们希望读者了解了 OpenCV 对图像的表示后，能够理解其他库中图像的表达，从而在需要数据格式时，能够自己处理。

另外，由于 cv::Mat 亦是矩阵类，除了表示图像之外，我们也可以用它来存储位姿等矩阵数据。只是一般认为 Eigen 对于固定大小的矩阵，使用起来效率更高一些。

5.4 实践：拼接点云

最后，我们来练习一下相机内外参的使用方法。本节程序提供了五张 RGB-D 图像，并且知道了每个图像的内参和外参。根据 RGB-D 图像和相机内参，我们可以计算任何一个像素在相机坐标系下的位置。同时，根据相机位姿，又能计算这些像素在世界坐标系下的位置。如果把所有像素的空间坐标都求出来，相当于构建一张类似于地图的东西。现在我们就来练习一下。

我们准备了五对图像，位于 `slambook/ch5/joinMap` 中。在 `color/` 下有 `1.png` 到 `5.png` 五张 RGB 图，而在 `depth/` 下有五张对应的深度图。同时，`pose.txt` 文件给出了五张图像的相机位姿（以 T_{wc} 形式）。位姿记录的形式是平移向量加旋转四元数：

$$[x, y, z, q_x, q_y, q_z, q_w],$$

其中 q_w 是四元数的实部。例如第一对图的外参为：

$$[-0.228993, 0.00645704, 0.0287837, -0.0004327, -0.113131, -0.0326832, 0.993042].$$

下面我们写一段程序，完成两件事：(1). 根据内参计算一对 RGB-D 图像对应的点云；(2). 根据各张图的相机位姿（也就是外参），把点云加起来，组成地图。

本书的点云库使用 PCL (Point Cloud Library)^①。PCL 的安装比较容易，输入以下命令即可^②：

```
1 sudo add-apt-repository ppa:v-launchpad-jochen-sprickerhof-de/pcl
2 sudo apt-get update
3 sudo apt-get install libpcl-all
```

安装完成后，PCL 的头文件将安装在 `/usr/include/pcl-1.7` 中，库文件位于 `/usr/lib/` 中。现在来写拼接部分的程序：

`slambook/ch5/joinMap/joinMap.cpp`

```
1 #include <iostream>
2 #include <fstream>
3 using namespace std;
4 #include <opencv2/core/core.hpp>
5 #include <opencv2/highgui/highgui.hpp>
6 #include <Eigen/Geometry>
```

^①官网：<http://pointclouds.org/>

^②在 Ubuntu 16.04 直接通过公共仓库的 `apt-get` 安装即可。

```
7 #include <boost/format.hpp> // for formating strings
8 #include <pcl/point_types.h>
9 #include <pcl/io/pcd_io.h>
10 #include <pcl/visualization/pcl_visualizer.h>
11
12 int main( int argc, char** argv )
13 {
14     vector<cv::Mat> colorImgs, depthImgs; // 彩色图和深度图
15     vector<Eigen::Isometry3d> poses; // 相机位姿
16
17     ifstream fin("./pose.txt");
18     if (!fin)
19     {
20         cerr<<"请在有 pose.txt 的目录下运行此程序"<<endl;
21         return 1;
22     }
23
24     for ( int i=0; i<5; i++ )
25     {
26         boost::format fmt( "./%s/%d.%s" ); //图像文件格式
27         colorImgs.push_back( cv::imread( (fmt%"color"%"(i+1)%"png").str() ) );
28         depthImgs.push_back( cv::imread( (fmt%"depth"%"(i+1)%"pgm").str(), -1 ) ); // 使用 -1 读取原始图像
29
30         double data[7] = {0};
31         for ( auto& d:data )
32             fin>>d;
33         Eigen::Quaterniond q( data[6], data[3], data[4], data[5] );
34         Eigen::Isometry3d T(q);
35         T.pretranslate( Eigen::Vector3d( data[0], data[1], data[2] ) );
36         poses.push_back( T );
37     }
38
39     // 计算点云并拼接
40     // 相机内参
41     double cx = 325.5;
42     double cy = 253.5;
43     double fx = 518.0;
44     double fy = 519.0;
45     double depthScale = 1000.0;
46
47     cout<<"正在将图像转换为点云..."<<endl;
48     // 定义点云使用的格式: 这里用的是 XYZRGB
49     typedef pcl::PointXYZRGB PointT;
50     typedef pcl::PointCloud<PointT> PointCloud;
51
52     // 新建一个点云
53     PointCloud::Ptr pointCloud( new PointCloud );
54     for ( int i=0; i<5; i++ )
55     {
56         cout<<"转换图像中: "<<i+1<<endl;
```

```

57     cv::Mat color = colorImg[i];
58     cv::Mat depth = depthImg[i];
59     Eigen::Isometry3d T = poses[i];
60     for ( int v=0; v<color.rows; v++ )
61         for ( int u=0; u<color.cols; u++ )
62         {
63             unsigned int d = depth.ptr<unsigned short>( v )[u]; // 深度值
64             if ( d==0 ) continue; // 为 0 表示没有测量到
65             Eigen::Vector3d point;
66             point[2] = double(d)/depthScale;
67             point[0] = (u-cx)*point[2]/fx;
68             point[1] = (v-cy)*point[2]/fy;
69             Eigen::Vector3d pointWorld = T*point;
70
71             PointT p ;
72             p.x = pointWorld[0];
73             p.y = pointWorld[1];
74             p.z = pointWorld[2];
75             p.b = color.data[ v*color.step+u*color.channels() ];
76             p.g = color.data[ v*color.step+u*color.channels()+1 ];
77             p.r = color.data[ v*color.step+u*color.channels()+2 ];
78             pointCloud->points.push_back( p );
79         }
80     }
81
82     pointCloud->is_dense = false;
83     cout<<"点云共有"<<pointCloud->size()<<"个点."<<endl;
84     pcl::io::savePCDFileBinary("map.pcd", *PointCloud );
85     return 0;
86 }
```

一点注解：

1. 14-39 行：读取彩色和深度图像对和位姿信息，并把位姿从四元数与平移向量转换为变换矩阵。注意程序里使用了 boost::format 进行字符串的格式化。
2. 65-80 行：计算位于 (u, v) ，深度为 d 的像素，在相机坐标系下的位置。并根据外参把它们变换到世界坐标。我们知道相机坐标 \mathbf{p}_c 到像素坐标 (u, v, d) 的关系为：

$$d \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \mathbf{K} \mathbf{p}_c. \quad (5.17)$$

反推 p_c 的形式亦非常简单。设 $p_c = [x, y, z]$, 那么:

$$\begin{aligned}z &= d \\x &= \frac{u - c_x}{f_x} z \\y &= \frac{v - c_y}{f_y} z.\end{aligned}$$

3. 为了编译此程序, 我们需三个库: Eigen、OpenCV 和 PCL。因此主程序的 CMakeLists.txt 应该是这样的:

```

1 # opencv
2 find_package( OpenCV REQUIRED )
3 include_directories( ${OpenCV_INCLUDE_DIRS} )

4
5 # eigen
6 include_directories( "/usr/include/eigen3/" )

7
8 # pcl
9 find_package( PCL REQUIRED COMPONENT common io )
10 include_directories( ${PCL_INCLUDE_DIRS} )
11 add_definitions( ${PCL_DEFINITIONS} )

12
13 add_executable( joinMap joinMap.cpp )
14 target_link_libraries( joinMap ${OpenCV_LIBS} ${PCL_LIBRARIES} )
```

最后, 我们把生成的点云以 pcd 格式存储在 map.pcd 中。用 PCL 提供的可视化程序打开这个文件:

```
1 pcl_viewer map.pcd
```

随后就可以看到拼合的点云地图了。你可以拖动鼠标, 查看此地图的样子。

在这个例程中, 我们使用相机内参和外参, 来计算一个像素在世界坐标系中的位置, 并把它们合并成一个点云。这是一个综合性的示例, 请读者仔细体会并掌握其内容。

习题

1. * 寻找一个相机 (你手机或笔记本的摄像头即可), 标定它的内参。你可能会用到标定板, 或者自己打印一张标定用的棋盘格。
2. 叙述相机内参的物理意义。如果一个相机的分辨率变成两倍而其他地方不变, 它的内参如何变化?
3. 搜索特殊的相机 (鱼眼或全景) 相机的标定方法。它们与普通的针孔模型有何不同?

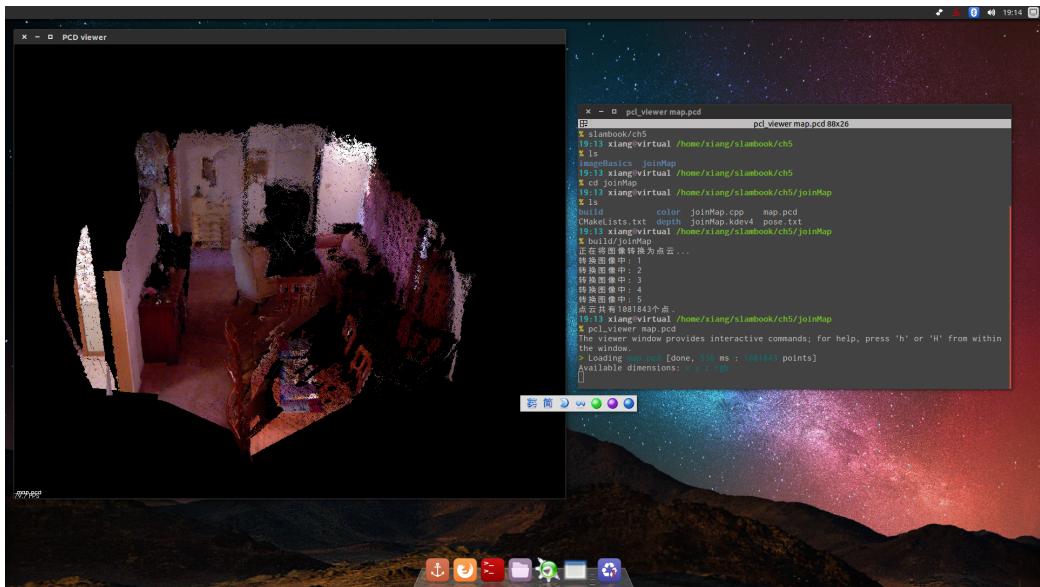


图 5-9 查看拼合的点云地图。

4. 调研全局快门相机（global shutter）和卷帘快门相机（rolling shutter）的异同。它们在 SLAM 中有何优缺点？
5. RGB-D 相机是如何标定的？以 Kinect 为例，需要标定哪些参数？（参照https://github.com/code-iai/iai_kinect2.）
6. 除了示例程序演示的遍历图像的方式，你还能举出哪些遍历图像的方法？
7. * 阅读 OpenCV 官方教程，学习它的基本用法。