LiveScan3D：依赖多个 Kinect v2 传感器实现快速廉价的 3D 数据重建系统

译者：王辰，网站: https://github.com/JohnnistWangc

摘要

LiveScan3D 是一个免费的开源系统，用于使用多个 Kinect Xbox One 传感器进行实时 3D 数据采集。

它允许用户在任何物理配置中放置任意数量的传感器，并开始以实时速度收集数据。

将传感器放置在任何配置中的自由度允许许多可能的采集方案，例如：从多个视点捕获单个对象或使用多个彼此靠近的设备创建 3D 全景图。

多亏了现成的 Kinect Xbox One 传感器，该系统既准确又便宜，从而使 3D 采集面向更多用户。

在本文中，我们使用该算法描述了我们的系统，并在包括头部形状重建和动态场景 3D 重建在内的多种情况下展示了其有效性。

1. 简介

为了从多个角度获取各种物体的完整、密集的 3D 形状，必须将来自所有 Kinect Xbox One 传感器采集的数据融合并获得整个物体。

完整 3D 形状的物体采集遇到的问题可以通过以下几种方式解决：

-1- 一是使用单个传感器在多个瞬间捕获对象，例如，在[7]中，将对象放置在旋转的桌子上并且相机固定；

-2- 或者在[8]或[9]中，传感器围绕固定的对象移动。

这种方法的明显缺点是，在持续几秒钟的整个扫描过程中，对象的形状必须保持相同。某些类型的对象和场景不能长时间保持恒定的形状。这可以通过使用多个同时捕获对象的传感器来解决。

近年来，包括学术界在内，诸如[14、4、6]之类的文章在商业界以及使用 Agisoft/PhotoScan 之类程序的应用界，都普遍使用多个 RGB 相机来实现这一目标。

但它们需要提供非常高分辨率的纹理和深度数据，基于 RGB 相机的方法在纹理很少的区域中工作困难，需要长时间的后期处理。

与此不同的是，RGBD 传感器虽然分辨率较低，但对对象纹理的依赖性较小，不需要后期处理即可获得 3D 数据。

通过使用多个 RGBD 传感器的多视图三维重建的方案需要的后期处理远远少于通过使用 RGB 传感器的方案，但这种情况也在改变。

最近几年的研究将 RGBD 传感器采集带入了一个新时代。引入了新的，廉价的 3D 深度相机，从而向广大群众开放了 3D 采集的研究方法。

诸如 Microsoft Kinect 开发者 、Creative Senz3D 之类的传感器可实现快速、实时的数据采集。

在本文中，我们介绍了使用多个 Kinect Xbox One 传感器进行数据采集的设置，算法和开源软件，我们将其命名为 LiveScan3D。

尽管以前已经提出过类似的系统，但它们要么使用了普通用户无法使用的定制设备[15]，要么使用了精度较低的 Kinect Xbox 360 [13，1]。

而且，最初为 Kinect Xbox 360 设计的软件无法轻松地与 Kinect Xbox One 一起工作，因为每个 Kinect Xbox One 都需要一台单独的计算机（2.1 中有更多详细信息）。

据我们所知，我们的系统是目前唯一的将多个 Kinect Xbox One 设备一起使用的全自动解决方案。以下各节包括：

- 系统的体系结构；
- 两步校准过程；
- 干扰问题的讨论；
- 开源软件应用程序的详细信息；
- 实验结果。

2. 采集系统

在本节中，我们将介绍 3D 采集系统。该系统使用经过校准的 Kinect Xbox One 传感器来捕获同时从多个视点观看到的场景并自动将它们融合。

本文的这一节结构如下：第 2.1 节介绍了采集系统的总体框架设计，第 2.2 节介绍了我们估算 3D 空间中相机姿态的方法，第 2.3 节讨论了多个 Kinect Xbox One 传感器设备之间相互干扰的问题。

2.1. 体系结构

在我们的系统中，每个 Kinect Xbox One 传感器都连接到单独的计算机上。每台计算机都连接到管理所有传感器数据的服务器。

C/S 设计是必须的，因为当前的 Kinect v2 SDK 只允许每台计算机运行一台 Kinect Xbox One 传感器。即使可以将多个 Kinect Xbox One 传感器连接到计算机，当前的 PCI-E 3.0 标准主板也可能仅具有用于两个传感器设备的足够带宽。

当记录帧顺序时，服务器通过同时向所有客户端计算机发送适当的信号来同步帧捕获。发送信号后，服务器将等待所有客户端确认已捕获帧，然后再发送下一个捕获信号。

在记录过程中，帧存储在客户端计算机的内存中。录制完成后，客户端将其帧发送到服务器以进行合并和保存。

由于记录期间计算机之间的网络通信量较低，因此该设计允许以较短的间隔捕获帧并具有良好的同步性。

每个 Kinect Xbox One 传感器都会产生 3D 数据并初始化自己的坐标系。

为了合并扫描，因此又必须知道所有设备的姿态并将 3D 数据从每个设备的独立坐标系转换为世界坐标系。每个客户端均应服务器的请求执行校准，从而将传感器定位在世界坐标系中。每台客户端的 Kinect Xbox One 传感器在捕获时将其框架所在的相机坐标系转化为世界坐标系。

和大多数类似的 toF 传感器设备一样，Kinect Xbox One 会产生带有槽点的数据。为了减少槽点，我们使用简单的过滤：对于每个给定点云，我们会找到它的 n 个最接近的邻居，如果到最远邻居的距离大于阈值 t，则认为该点是离群值（槽点）。

n 和 t 的值是根据场景手动设置的。类似帧转换一样，过滤是在每台客户端计算机上分别执行的。这种分布式的计算形式使整个系统更快，并允许实时过滤。图 1 显示了应用过滤前、过滤后重构的点云。

2.2. 传感器姿态估计（校准）

我们的姿态估计过程包括两个步骤。

Step1 提供了相机姿势的初始粗略估计，随后在 Step2 中进行了细化。

(a) (b) (c)

图 1：一把吉他放在一只办公室椅子上的场景：(a)该场景的照片,(b)不过滤时 3D 重建的结果,(c)过滤时 3D 重建的结果。

在本小节中，我们用到了三种不同的坐标系，分别是：

-1- 以传感器为中心的 Kinect Xbox One 传感器的坐标系

-2- 以给定标记为中心的标记坐标系，它的旋转由每个标记中的五个顶点和-3-真实世界坐标系定义，该坐标系的中心和旋转都由标记位置定义。标记位置是用户基于-3-真实世界坐标系中标记应该存放的位置手动校定。

-3- 上述-2-中提到多次的真实世界坐标系

2.2.1 *粗略估计*

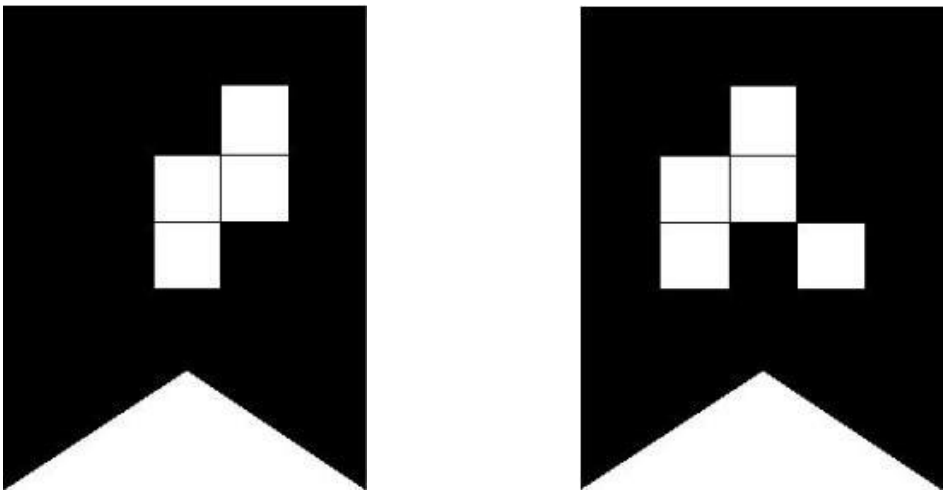
Step 1:使用此前在[11]中提出的 2D 视觉标记系统，图 2 中显示了一些标记。

这些标记都是由五个顶点构成，包含四个凸角和一个凹角的五边形。

每个标记中都包含 9 个黑白格子，代表 16 种可能的代码之一。

每个代码对应一个 4 位的二进制数字，这 9 个字段中的前 4 个显示 4 位，白色为 1，黑色为 0，随后的 4 个字段以相同的颜色显示相同的数字，最后的字段是奇偶校验位。

这种强大的标定板方法能使你轻松地验证被校准的标记是否为真。



(a) (b)

图 2：在校准程序的 Step1 中使用的两个标记。标记(a)代表的值是 2，而标记(b)代表的值是 5。请注意，标记的形状便于确定方向。

给定输入图像，使用阈值查找潜在的标记位置，这会产生许多假阳性结果。要丢弃它们，我们会将检测到的区域的顶点数量与标记形状的顶点数量进行比较。根

据每个标记应包含的九种领域模式来检查剩余的潜在标记。未丢弃的标记的顶点位置以亚像素精度精确定义。

知道彩色图像中的标记顶点位置后，即可使用 Kinect For Windows SDK 的功能在传感器坐标系中找到标记的 3D 位置。

知道标记的 3D 位置后，我们将找到一个转换，将任何点 xs 从传感器坐标系转换为标记坐标系中的点 xm：

$$x_m = R_s(x_s - t_s), \qquad (1)$$

其中 ts 是传感器坐标系中标记的中心，Rs 是使用正交 Procrustes 问题方法[12]找到的旋转矩阵。世界坐标系中每个标记的位置是已知的，因此可以将任何点从标记坐标系 xm 转换为世界坐标系 xc：

$$x_c = R_m x_m + t_m, \qquad (2)$$

其中 Rm 和 tm 是用户指定的标记的旋转和平移。注意，与所有其他变换相反，等式(2)中的平移跟随旋转。引入此功能是为了方便用户，以这种方式更容易手动输入标记姿态。公式(1)和(2)可以组合为：

$$x_c = R_m R_s (x_s - t_s) + t_m, \qquad (3)$$

它将传感器坐标系 xs 中的任何点转换为世界坐标系 xc 中的对应点。如果任何设备在其视野中检测到多个标记，则选择面积最大的标记进行校准。

2.2.2 精确度

最初的相机位置估计不准确，因为很难精确地测量标记相对于彼此的姿态。

这个手动测量的错误会在相机相对姿态中引入错误，从而导致合并过程中的点云未对准。

姿势估计的 Step2 使用迭代最近点（ICP）[2]完善初始估计。

每次在服务器上进行整理时，会同时使用所有客户端设备中传感器采集到的数据。

服务器首先会从每个设备中请求一帧的 3D 数据。当收到这些不同客户端程序抛回的帧数据之后，就用下述算法 1 进行优化。

Input: $F_{1..n}$ - 3D point cloud from each sensor
Data: $R_{1..n}, t_{1..n}$ - initialized to identity
Result: $R_{1..n}, t_{1..n}$ - refinement transform for each sensor

$v = 1$
$\tau = 0.01$
$firstPass = true$
$e = 0$
$\hat{e} = 0$
while $v > \tau$ **do**
  $e = 0$
  for $i = 1 : n$ **do**
    $K = \{1, 2, ..., n\} \setminus i$
    /* ICP aligns the i-th shape to all the other ones, updates R_i, t_i and returns the average error per point */
    $e = e + ICP(F_K, F_i, R_i, t_i)$
  end
  if $firstPass == true$ **then**
    $firstPass = false$
    $v = 1$
  else
    $v = \frac{|e - \hat{e}|}{e}$
  end
  $\hat{e} = e$
end

Algorithm 1: Camera pose refinement

该算法的结果用于更新将点从 Kinect 坐标系映射到世界坐标系的转换。

其中，i 是客户端的索引，而 Ri 和 ti 是算法 1 的结果。客户端依据(3)做出转换的定义如下：

$$x_c = R_r(R_m R_s(x_s - t_s) + t_m + t_r). \qquad (4)$$

2.3 干扰

当同时使用多台传感器设备时，传统的的 Kinect Xbox 360 传感器由于存在干扰问题而闻名[10]。

但是 Kinect Xbox One 基于不同的原理(Time of Flight/飞行时间)。

虽然多台基于 TOF 的设备一起使用时也会产生干扰，但据我们所知，还没有文章讨论 Kinect Xbox One 的这一问题。

我们的实验结果表明，仅当 Kinect Xbox One 面向物体表面时，干扰的负面影响才可见：由于反射令物体表面将一个设备的照明光束反射到另一设备的传感器中。

如果存在干扰，则可以很容易地识别出它，因为重建的表面在连续的 3D 帧之间快速移动。

在大多数情况下，精心计划的设备布置可以大大减少干扰。第 4.1 节中介绍了这种安排的示例。

3. LiveScan3D 软件

LiveScan3D 是我们创建的一种开源软件，用于实现本文所述的方法。

它由两个程序组成：LiveScanServer 和 LiveScanClient。

客户端利用 TCP/IP 协议与服务器实现同一局域网内的进程间通讯。

我们的框架允许以任何配置连接任何数量的 Kinect Xbox One 传感器设备。

为了运行该软件，用户需要为每个工作的传感器提供一台装有 Kinect v2 For Windows SDK 的计算机。

为了便于校准，用户还需要制作一个带有标记的校准对象。图 3 显示了一个非常简单且易于生产的校准对象的示例，该校准对象是由带有四个标记的纸板箱制成的。

3.1 LiveScanServer

服务器应用程序同时管理所有客户端。

它允许用户记录作为彩色点云保存到 ply 文件的队列，它也允许调整控制程序的某些设置，例如：被重建的物体所在空间的边界框，过滤参数，校准参数，标记位置等。

该应用程序还具有以下功能：一个“实时视图”窗口。用户可以通过这个窗口实时地查看所有设备上的重建场景。

但是，帧速率不如从记录的数据中获得的帧速率高，这是由于在“实时视图”窗口中的数据必须通过以太网实时发送。

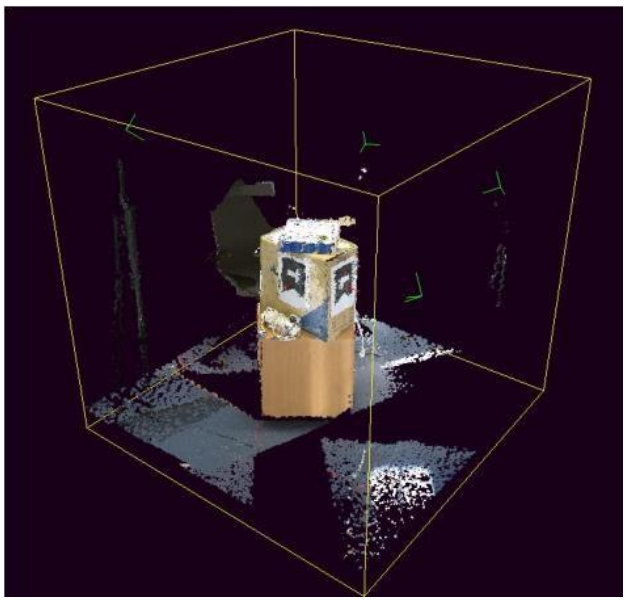


图 3:LiveScanServer 的实时视图窗口。

绿色标记表示 Kinectv2 传感器的摆放姿态，其位置如图 4(b)所示。

红色标记表示手动添加的标记姿态。

他们用黄色的框架定义了被重建对象所在的空间，任何超出它的数据都将被丢弃。

位于点云中心的是很容易校准的目标，该校准目标由带有四个标记的纸板箱制成。

图 3 显示了实时视图窗口输出的场景示例。

实时视图窗口对于设置视觉标记在世界坐标系中的位置非常有用。

这样可以不用手动测量各个传感器的姿态，直接在实时视图窗口内观察其大致姿态并对齐，再使用校准功能进行修正，从而获得该场景下最佳对齐的结果。

3.2 LiveScanClient

客户端的应用程序负责控制连接到计算机上的 Kinect Xbox One 传感器设备。

客户端接收到服务器的指令后执行校准，并将每帧捕获到的数据转换到世界坐标系。

在客户端窗口中，用户还可以检查给定传感器的原始图像和深度数据。

使用 OpenCV 库[5]实现 2.2.1 中描述的 2D 标记检测过程。

整个标记检测模块可以轻松地用于具有给定接口的类，从而实现的不同传感器设备的校准。

(a)

(b)

图 4：我们用于实验的 Kinect Xbox One 传感器

图(a)显示了第 4.1 和 4.3 中使用的传感器对内的设置，利用四个传感器针对固定空间内的聚焦重建

图(b)显示了 4.2 中使用的传感器视角向外的设置，利用三个相邻的传感器针对大场景空间的发散重建

图 5：实验 4.1 中不同表情的 3D 头部扫描。

从顶部的一个视点到底部的另一个视点显示了三个头部。

图 6：基于实验 4.1 扫描过的头部的 A3D 打印图。

4. 实验

为了进行实验，我们使用了四个 Kinect Xbox One 传感器设备连接到四台计算机。

每台计算机均配有 4 个内核和至少 8 GB 内存的 Intel Core i7 处理器。

我们测试了两种场景的设备配置，

-1- 配置 1：四个 Kinect Xbox One 传感器面向内（第 4.1 和 4.3 小节）

-2- 配置 2：三个 Kinect Xbox One 传感器彼此靠近且面向外（第 4.2 小节）

两种配置均如图 4 所示，每台计算机都在运行 LiveScanClient 客户端程序，其中一台计算机还在运行 LiveScanServer 服务器端进程。

使用四个设备进行的测试已达到每秒 25 帧的帧速率（使用 i5 处理器大约能达到 10 帧的 fps）。

4.1 3D 头部重建

精确的 3D 头部重建具有很多应用场景，包括从头部 3D 打印技术到生成非刚性 3D 头部模型，比如说[3]中提出的模型。

在 3D 头部扫描期间，通常用户必须做出特定的面部表情，以用于创建该表情的混合形状/变形器。

以一种现实的方式同步一些面部表情是非常困难的，因为它需要实时进行面部表情的 3D 扫描。

而我们的系统能够实时进行采集，所以我们认为它非常适合 3D 头部重建。

为了获得整个头部的形状，我们使用了一个面向内的传感器配置（见图 4）。

请注意，图 4 中的传感器朝下，我们的实验表明，这减少了 Kinect 之间的干扰量。

图 5 展示了我们在此实验中获得的一些结果。

图 6 展示了其中一个头部扫描的 3D 打印图。

4.2 3D 全景图像

在这个实验中，我们想证明我们的系统也可以用于设备朝外的配置中（见图 4）。

我们使用了三个传感器，这些传感器最初使用两个视觉标记进行了校准，之后使用 2.2.2 中描述的方法对校准进行了进一步的校准。

我们称此设置为全景设置，因为它类似于用于拍摄全景图像的标准 RGB 相机设置。

实验表明，我们的系统适用于这种情况。

如图 7 所示，来自各个传感器的帧已正确合并。

来自不同传感器的点之间唯一可见的差异是颜色，该颜色因自动设置每个传感器的曝光设置而有所不同。

4.3 3D 重建动态场景

此实验可以证明我们的系统能捕获动态场景，并获得良好的帧速率，而且来自不同传感器的数据之间的点云对齐也合理。

为了方便保存测试数据，我们将一个乒乓球扔进了从木柜弹起的重建场景中。图 8 显示了场景中球的轨迹以及测试数据中的单个帧。

我们还测试了另一个场景：人坐在椅子上弹吉他，如图 9 所示。

请注意，在图(b)和(c)中，吉他盒上的某些点是人身体上皮肤的颜色。在人的脸上有些白色的地方也是类似的问题。

这两个问题均源于 Kinect V2 For Windows SDK 在某些情况下无法正确匹配深度和色彩。这个问题在点云的边缘上最为明显，在该点云中，给定的前景点可能会被赋予背景颜色，反之亦然。

我们系统将来的版本会将贴有纹理的网格作为输出，而不是彩色点云。这很可能会解决问题，因为对于每个三角形，可以从摄像机中以最方便的角度看到该三角形并为其分配纹理。

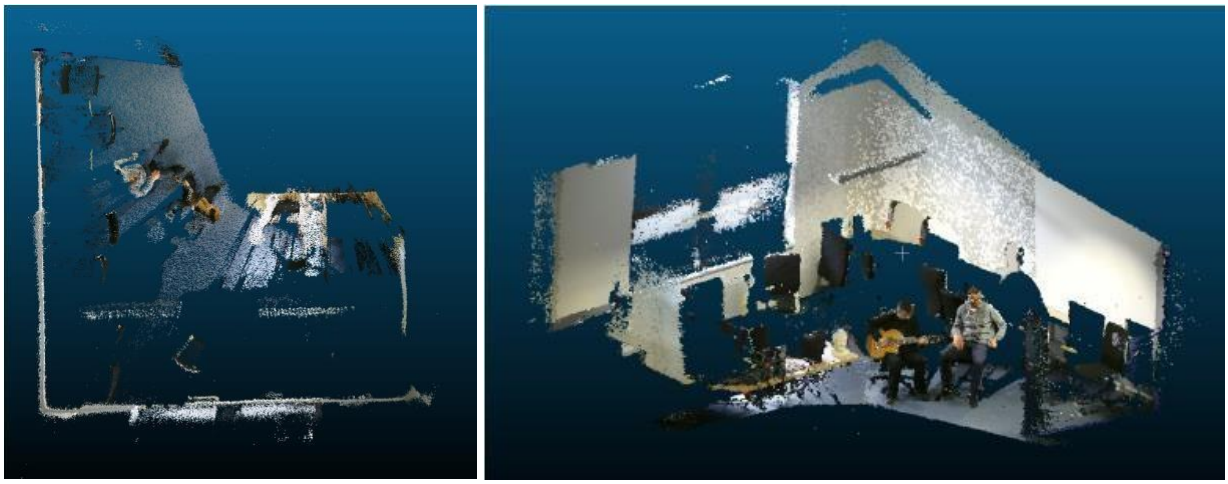
5. 结论

我们已经提出了使用多个 Kinect Xbox One 传感器设备进行实时 3D 重建的免费开源解决方案。

我们希望我们所创造的东西将有益于科学社区，激发利用多台 Kinect Xbox One 传感器设备的新研究。

我们计划接下来会继续改进和扩展之前提到的架构，其中一些新功能可能包括：

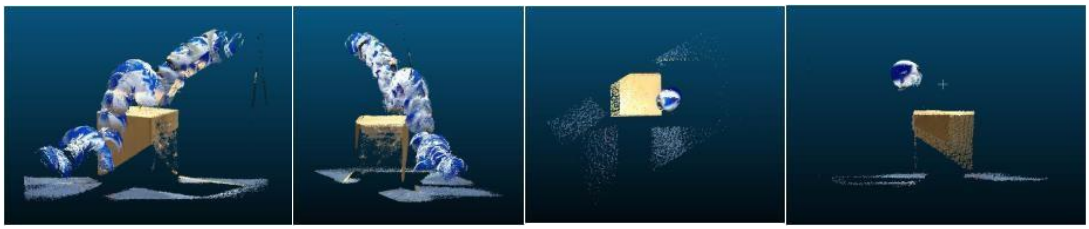
-1-无标记自动校准，-2-完善并将槽点消除，-3-输出点云的镶嵌和纹理化。



(a) (b)

图 7：由三个传感器生成的同一点云的两个视图。

注意：(a)左下角的重叠壁表面和(b)中相同的壁表面，都表明点云的良好对齐。



(a) (b) (c) (d)

图 8：一个动态队列，其中一个乒乓球被扔到一个木制橱柜上，然后球弹到了地板上。

图(a)和(b)展示了队列中所有点云的叠加（共 27 个），以演示球的路径以及拍摄的帧数。

在第一帧和最后一帧中，由于并不是所有传感器都能看到球，所以没有完全重建乒乓球。

图(b)和(c)展示了一个单独的框架，该框架描绘了乒乓在刚击中桌子之前的情况。 请注意：乒乓球由于其速度较快而变得模糊。

(a) (b) (c) (d)

图 9：一个人弹吉他的动态序列。

图(a)至(d)从顶部的一个视角转换到底部的另一个视角，显示了该队列里四个连续的帧。