深度与颜色信息融合的实时三维重建：基于 KinectFusion 的方法

金梁梁，蒋馨毅，孙雪晴，李庭安，李姝瑶，刘畅

摘要

三维重建技术的迅猛发展引发了对于更为高效、精准的处理方法的需求。本文主要探讨了一种基于 *KinectFusio*n 的实时三维重建方法，这种方法通过接受深度图像序列及其对应的 *RGB* 图像序列作为输入，利用顶点图像和法向量图像的生成以及图像配准，实现了准确的位姿估计和有效的 *Surface Reconstruction*。为了验证该方法的有效性，我们进行了一系列的实验。

首先，通过金字塔生成和双边滤波对深度图像进行预处理，降低图像分辨率并保留边缘特征。然后，采用深度相机的采样原理将深度图像转换为点云，存储在 *GpuMat* 中并计算每一点的法向量。接下来，我们设计了一个全局位姿估计的优化策略，通过匹配当前帧和上一帧中的点，以最小化点到平面的距离误差并优化位姿变换矩阵。此外，我们引入了截断带符号距离函数（*TSDF*），以提取物体表面信息并降低存储和计算负担。基于 *Ray Casting*，我们实现了 *Surface Prediction*，以确定物体表面的位置并计算法向量。

通过使用 *ICL-NUIM* 数据集中的 *Living Room* 作为测试对象，我们对该方法进行了评估。结果显示，该方法能在多数场景下实现准确的三维重建，并在有限的计算资源下实现实时重建。然而，当参照物稀缺时，可能导致重建效果出现错位。

尽管我们的方法在大多数场景下已经取得了相当不错的效果，但在参照物稀缺的情况下，我们的方法可能需要进一步改进。因此，未来的工作将探索如何改进该方法以更好地处理参照物稀缺的情况，并进一步优化其在低算力设备上的性能。总的来说，我们的工作为实时的三维重建技术提供了一种有效的优化策略。

目录

1 需求分析

1.1 背景

KinectFusion 是一种革新性的实时三维重建技术，它于 2011 年由微软研究院提出并由此引发了广泛的研究和应用。

微软发布的 *Kinect* 深度相机为我们提供了这样一个可能性，即获取和处理深度图像。*Kinect* 能够同时捕捉到 *RGB* 图像和深度图像，后者可以为每个像素点提供离相机的距离信息，使得三维扫描和定位成为可能。这一重要的技术突破使我们能够获取到深度图像数据，并在此基础上催生了许多深度图像处理算法，其中就包括 *KinectFusion*。

其次，在实时处理大量的三维数据时，*KinectFusion* 算法需要强大的计算资源支持。过去的计算环境无法支持这样的需求，但是近年来，*GPU* 的计算能力大幅度提升，并且还出现了许多专门的并行计算框架，例如 *CUDA* 和 *OpenCL*，这使得 *KinectFusion* 算法得以在普通的 *PC* 上实时运行，极大推动实时三维重建技术的发展。

总的来说，深度相机的普及和并行计算技术的发展构成了 *KinectFusion* 产生和发展的主要技术背景。这两项关键技术的进步不仅为实时三维重建技术的研究提供了基础，而且也为我们的研究带来了重要的启示和指引。

1.2 拟解决问题

通过连续的深度图像序列来构建出对应的三维模型，并通过对应的 RGB 图像序列来使其上色。

2 概要设计

2.1 相关工作

在过去的研究中，*Structure from Motion (SfM)* 和 *Multi-view stereo (MVS)* 方法成功提供了准确的相机跟踪、稀疏重建结果以及增量式的稠密表面重建结果，但这些方法并未针对实时应用场景优化。而 *Simultaneous Localization and Mapping (SLAM)* 技术，则基于单目相机，能对动态场景进行实时的相机跟踪以及地图构建，尽管其重建的地图

主要是稀疏点云，只能被视为初步的场景重建结果。对于基于单目相机的实时稠密三维重建来说，其挑战主要在于相机运动的估计以及对场景照明的要求。新的深度相机如 *Time-of-Flight (ToF)* 和结构光传感器虽然适用于此类任务，但现有的算法并未充分利用这类设备提供的数据和数据采集速度。

本文实现了一种新的方法，允许使用 *KinectFusion* 进行实时稠密的 *TSDF* 重建。这种方法只使用深度信息，可以连续跟踪相机的 6 自由度位姿，并将深度信息更新至 *TSDF* 模型，跟踪速率可高达 30FPS。为了实现这一目标，本文所有的设计核心都是希望使用 *GPU* 进行并行化计算，使得相机跟踪和建图都能实时进行。针对 Kinect 传感器存在的运动模糊和深度图孔洞等问题，本文对含有噪声的深度图进行了处理。此外，本文还引入了无漂移的 *SLAM*、基于深度相机的稠密跟踪建图、*Signed Distance Function (SDF)* 稠密场景表示等前沿技术，以进一步提高重建的质量和效率。此方法不仅可以对小物体进行高质量扫描，也可以针对更高速移动相机重建较大规模场景，具有很大的应用前景。

2.2 算法流程

这个过程可以分为以下四个主要步骤：

1.*Surface Measurement*：通过深度图像生成顶点图像和法向量图像。

2.*Pose Estimation*：将当前帧的顶点图像和法向量图像与通过 *Ray Casting* 推导的顶点图像和法向量图像进行配准。

3.*Surface Reconstruction*：将当前帧顶点图像的信息融合到 *TSDF Volume* 中。

4.*Surface Prediction*：在上一帧相机位姿处进行 *Ray Casting*，推导出对应的顶点图像和法向量图像。

上面的描述主要是关于深度信息，而对于颜色信息，则是在 *Surface Reconstruction* 的过程中，采用与深度值类似的处理方式，将其与顶点对应并融合到存储颜色用的

$TSDF\ Volume$中。

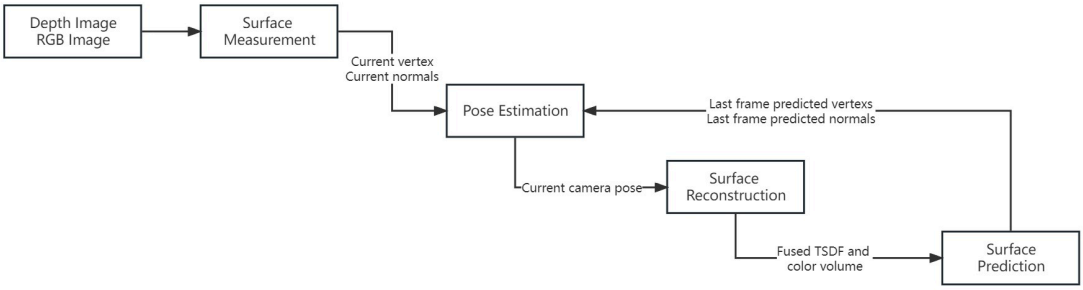

Figure 1: KinectFusion算法流程

3 实现方法

3.1 Surface measurement

```
// PreProcessor.cpp
FrameData PreProcessor::preProcess(const cv::Mat& depth_map){
    cv::cuda::Stream stream;
    //生成深度图图像金字塔
    for (int level = 1; level < numLevels; level++)
                        cv::cuda::pyrDown(data.depth_pyramid[level     -     1],
data.depth_pyramid[level], stream);
    //对图像金字塔中的每一张图像都进行双边滤波操作
    for (int level = 0; level < numLevels; level++) {
        cv::cuda::bilateralFilter(data.depth_pyramid[level],
                                            // source
                                  data.smoothed_depth_pyramid[level],
                                                // destination
                                  globalConfiguration.kernalSize,
                                    // 双边滤波器滤波的窗口大小
                                  globalConfiguration.colorSigma,
                                  globalConfiguration.spatialSigma,
                                  cv::BORDER_DEFAULT,
                                // 默认边缘的补充生成方案
                                  stream);
    }
}
```

深度图的预处理操作包括生成金字塔和双边滤波。为了实现这两个方法，我们可以直接调用$OpenCV$库函数 $pyrDown$ 和 $bilateralFilter$ 。

生成金字塔的过程涉及高斯核卷积，其中每个像素点的值通过对周围 8 个点的信息进行加权平均计算。在完成卷积操作后，我们按照规定的步长删除行和列，以减小图像的分辨率。金字塔的最底层即为原始深度图像，通过调整代码中的$numLevels$参数，我们可以生成不同层次的金字塔。在本示例中，我们采用了 3 层金字塔。

接下来，我们对金字塔中的三张深度图像进行双边滤波处理。需要注意的是，这里直接对深度图像进行滤波操作，而非直接对点云进行处理。

3.1.1图像金字塔--高斯金字塔

图像金字塔是一系列分辨率逐步降低的图像集合，它以金字塔的形状排列，源自同一张原始图像。下图展示了一个包含五层图像的金字塔示例。通过逐步向下采样的方式，可以获取图像金字塔，直到达到所需的金字塔层数为止。金字塔的层次越高，图像的分辨率越低。

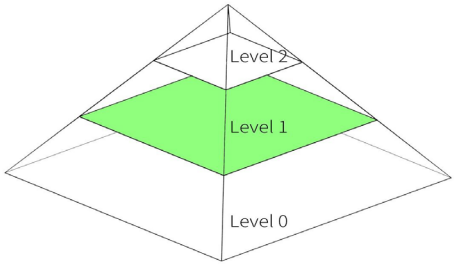

Figure 2: 生成金字塔

在金字塔中，高斯金字塔是一种常见的多尺度表示工具，通过连续应用高斯卷积和降采样操作来生成一系列逐渐降低分辨率的图像。高斯金字塔的底层是原始图像，而每一层的图像都比前一层的图像尺寸减小一半，从而形成金字塔结构。

生成高斯金字塔的过程首先对图像进行高斯卷积，该过程是一种加权平均的操作，每个像素的新值是它自身和周围像素值的加权平均。高斯核卷积使得邻近中心像素具有更高的权重，权重随着距离中心的增加而逐渐减小，其权重分布呈现高斯函数的形状。常见的 3 x 3 高斯核权重矩阵如下所示：

$$K(3,3) = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

通过加权平均操作，每个像素点都能融合周围 8 个像素点的信息，从而实现图像的平滑效果。随后，对卷积后的图像进行降采样操作，即删除所有偶数行和列，以减半图像的分辨率。通过重复这两个步骤，我们可以生成一系列图像，形成高斯金字塔结构。

在本示例中，我们生成了一个包含 3 层的高斯金字塔。底层是原始深度图像，而每一层的分辨率都比前一层减小一半。通过调整代码中的 $numLevels$ 参数，可以确定生成金字塔的层数。

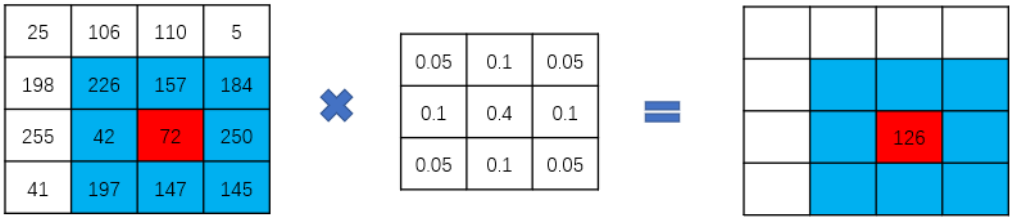

Figure 3: 卷积

3.1.2双边滤波

双边滤波器的核由两个函数生成: 空间域核和值域核，能够做到在平滑去噪的同时还能够很好的保存边缘。

空间域核: 由像素位置欧式距离决定的模板权值w_d

$$w_d(i,j,k,l) = exp\left(-\frac{(i-k)^2+(j-l)^2}{2\sigma_d^2}\right)$$

其中， i 和 j 代表了图像中的一个像素位置，而 k 和 l 则代表了图像中的另一个像素位置。这两对变量共同确定了图像中两个像素之间的空间关系。函数$w_d(i,j,k,l)$是空间距离权重函数，用于度量两个像素在空间位置上的相似性。公式中的σ_d^2是一个预先设定的参数，用于调整空间相似性的权重。一般来说，σ_d^2越大，相距较远的像素对权重函数的影响越大。

值域核：由像素值的差值决定的模板权值w_r。

$$w_r(i,j,k,l) = exp\left(-\frac{\| f(i,j) - f(k,l) \|^2}{2\sigma_r^2}\right)$$

其中，函数$w_r(i,j,k,l)$是值域相似性权重函数，用于度量两个像素在灰度强度上的相似性。其中，$f(i,j)$和$f(k,l)$分别代表了位置(i,j)和位置$(k,\ l)$的像素强度。公式中的σ_r^2是一个预先设定的参数，用于调整强度相似性的权重。一般来说，σ_r^2越大，相似性在强度上差距较大的像素对权重函数的影响越大。

将上述两个模板相乘就得到了双边滤波器的模板权值:

$$w(i,j,k,l) = w_d(i,j,k,l) \cdot w_r(i,j,k,l) = \exp\left(-\frac{(i-k)^2 + (j-l)^2}{2\sigma_d^2} - \frac{||f(i-j) - f(k,l)||^2}{2\sigma_r^2}\right)$$

在平坦区域，相邻像素的像素值差异较小，对应的值域权重w_r接近于 1。这意味着在这些区域，空间域权重w_d起主要作用，类似于对该区域进行高斯模糊处理。而在边缘区域，相邻像素的像素值差异较大，导致值域权重w_r接近于 0，进而导致核函数的值下降（因为$w = w_d \cdot w_r$）。因此，当前像素受到的影响较小，从而保留了原始图像中的边缘细节信息。

参数的效果如下图所示：

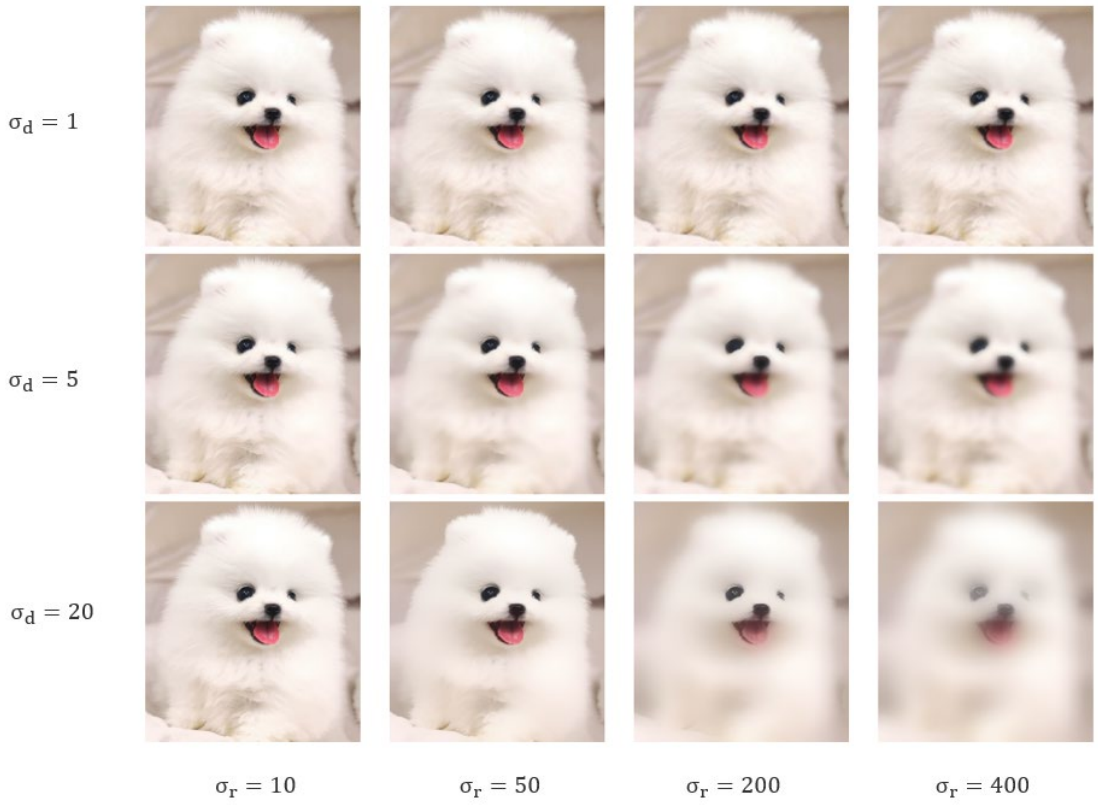

Figure 4: 双边滤波效果

3.1.3 vertex map生成

```
// SurfaceMeasurement.cu, 核函数，用于计算深度图像中每一个像素的3D点
__global__
void kernel_compute_vertex_map(
    const PtrStepSz<float> depth_map,       // 滤波之后的深度图像对象
    PtrStep<float3> vertex_map,             // 保存计算结果的顶点图
    const float depth_cutoff,               // 截断距离，不考虑的过远的点的距离
    const CameraParameters cam_params)      // 相机内参
{
    // step 1 计算对应的下标，并且进行区域有效性检查
    const int x = blockIdx.x * blockDim.x + threadIdx.x;
    const int y = blockIdx.y * blockDim.y + threadIdx.y;

    // 如果当前线程对应的图像像素并不真正地在图像中，那么当前就不需要计算了
    if (x >= depth_map.cols || y >= depth_map.rows)
        return;

    // step 2 获取深度并且进行背景判断
    float depth_value = depth_map.ptr(y)[x];
    if (depth_value > depth_cutoff) depth_value = 0.f; // Depth cutoff

            //      step      3      生成三维点，      根据相机内参进行反投影即可，
得到的是三维点在当前世界坐标系下的坐标
    Vec3f vertex(
            (x - cam_params.principal_x) * depth_value / cam_params.focal_x,
            (y - cam_params.principal_y) * depth_value / cam_params.focal_y,
            depth_value);

    // step 4 保存计算结果
    vertex_map.ptr(y)[x] = make_float3(vertex.x(), vertex.y(), vertex.z());
}
```

首先，我们来介绍深度相机采样生成深度图的基本原理，该原理基于相似三角形的概念。

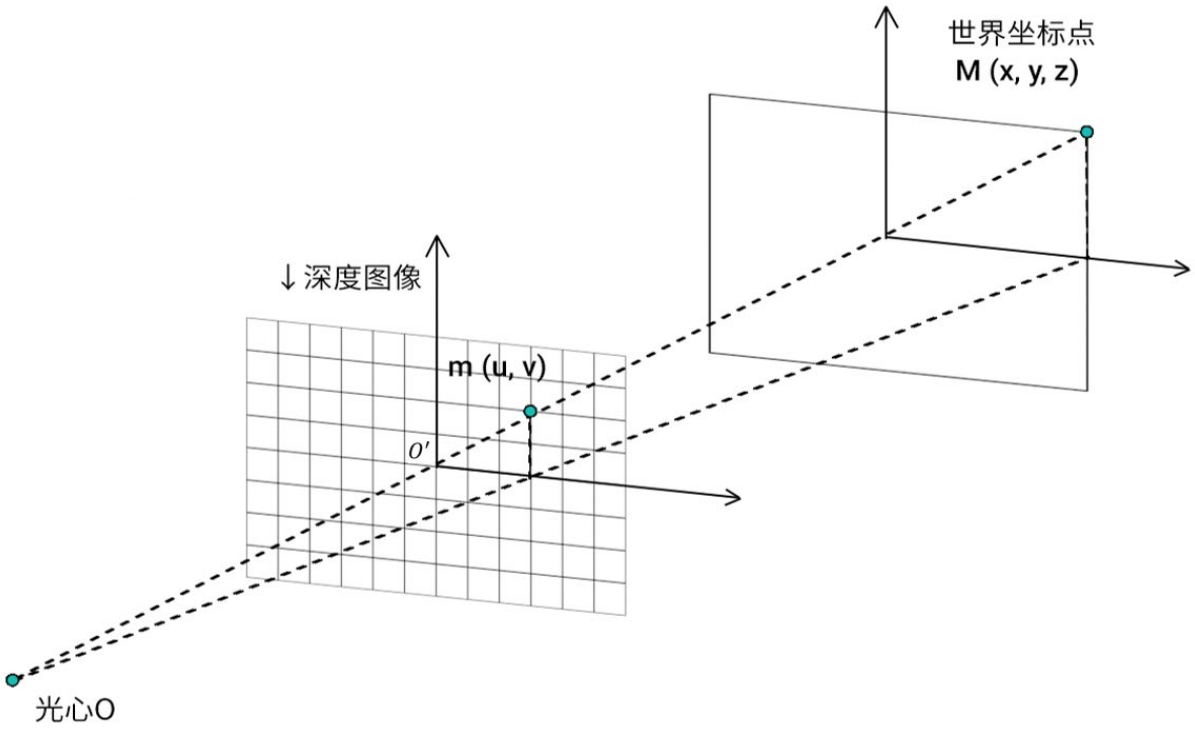

Figure 5: 深度相机采样示意图

如上图所示，假设深度图上的某一点 m 需要转换为世界坐标中的点 M，即深度图转点云的子过程。我们可以利用图中的三角形OmO'和OMA的相似性来推导转换关系。根据相似三角形的性质，我们可以得到以下关系式：$\frac{OO'}{OA} = \frac{mO'}{MA}$。

$$u = f_x \frac{X}{Z} + c_x$$
$$v = f_x \frac{Y}{Z} + c_y$$

将这个相似关系映射到坐标系中，可以表示为矩阵形式：

$$Z\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

其中，Z 代表深度值，即世界坐标系下的 Z 坐标。(u, v)表示深度图中点的坐标，而(X, Y, Z)表示世界坐标系下的坐标。右侧的 3 x 3 矩阵即为相机内参矩阵 K 。通过以上关系式，我们可以推导出在"kernel_compute_vertex_map"函数中第 3 步所使用的表达式。需要注意的是，代码中并没有使用矩阵求逆的操作，这可以理解为 K 矩阵将三维信息转换为深度图信息。而通过K^{-1}变换则是将深度图转换回三维信息。

3.1.4 normal map生成

```
 // SurfaceMeasurement.cu, 核函数, 用于根据顶点图计算法向图
__global__
void kernel_compute_normal_map(
    const PtrStepSz<float3> vertex_map,                 // 输入, 顶点图
    PtrStep<float3> normal_map)                         // 输出, 法向图
{
    // step 1 根据当前线程id得到要进行处理的像素, 并且进行区域有效性判断
    const int x = blockIdx.x * blockDim.x + threadIdx.x;
    const int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x < 1 || x >= vertex_map.cols - 1 || y < 1 || y >= vertex_map.rows -
1)
        return;

          //    step    2    获取以当前顶点为中心,    上下左右四个方向的顶点数据,
都是在当前帧相机坐标系下的坐标表示
    const Vec3f left(&vertex_map.ptr(y)[x - 1].x);
    const Vec3f right(&vertex_map.ptr(y)[x + 1].x);
    const Vec3f upper(&vertex_map.ptr(y - 1)[x].x);
    const Vec3f lower(&vertex_map.ptr(y + 1)[x].x);

    // step 3 计算当前顶点的法向
    Vec3f normal;
    // 当前的顶点的上下左右只要有一个顶点无数据, 那么当前的顶点就没有法向数据
    if (left.z() == 0 || right.z() == 0 || upper.z() == 0 || lower.z() == 0)
        normal = Vec3f(0.f, 0.f, 0.f);
    else {
        // 计算一个 right -> left 的向量
          Vec3f hor(left.x() - right.x(), left.y() - right.y(), left.z() -
right.z());
        // 计算一个 lower -> upper 的向量
         Vec3f ver(upper.x() - lower.x(), upper.y() - lower.y(), upper.z() -
lower.z());

        // 叉乘, 归一化, 确保法向的方向是"朝外的"(相对于相机光心来说)
        normal = hor.cross(ver);
        normal.normalize();

        if (normal.z() > 0)
            normal *= -1;
    }

    // 保存计算的法向量结果
```

```
    normal_map.ptr(y)[x] = make_float3(normal.x(), normal.y(), normal.z());
}
```

我们的顶点数据存储在 GpuMat 中，并且不考虑 RGB 等其他信息。可以简化地将其理解为一个维护着一个 640 x 480 的二维数组，其中每个数组元素都是一个三维向量 (x, y, z)。对于数组中的某一点，我们可以获取其上方、下方、左侧和右侧所存储的点的信息，并通过计算这四个点两对之间的差向量，再进行叉乘运算，以获得该点对应的法向量。

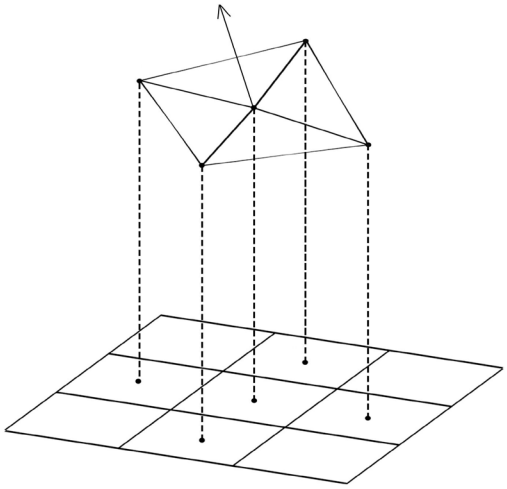

Figure 6: 法向量计算示意图

3.2 Pose Estimation

3.2.1 ICP(Iterative Closest Point)位姿估计模型

投影数据关联判断关系式：

$$\Omega(\mathbf{u}) \neq \text{null iff} \begin{cases} \mathbf{M}_k(\mathbf{u}) & = & 1, \\ \|\tilde{\mathbf{T}}^z_{g,k}\dot{\mathbf{V}}_k(\mathbf{u}) - \hat{\mathbf{V}}^g_{k-1}(\hat{\mathbf{u}})\|_2 & \leq & \varepsilon_d, \\ \langle \hat{\mathbf{R}}^z_{g,k}\mathbf{N}_k(\mathbf{u}), \hat{\mathbf{N}}^g_{k-1}(\hat{\mathbf{u}})\rangle & \leq & \varepsilon_\theta. \end{cases}$$

位姿估计的误差表达式：

$$\mathbf{E}(\mathbf{T}_{g,k}) = \sum_{u\in\Omega} \left\|(\mathbf{T}_{g,k}\dot{V}_k(\mathbf{u}) - \hat{V}_k)^\top N^g_{k-1}(\hat{\mathbf{u}})\right\|_2$$

其中，$V_k(u)$表示在第 K 帧中的测量的表面顶点位置。$N_k(u)$表示在第 k 帧中的测量的表面法向量。$\hat{V}^g_{k-1}(\hat{u})$表示在第$K-1$帧的模型预测中的表面顶点位置。$\hat{N}^g_{k-1}(\hat{u})$表

示在第$K-1$帧的模型预测中的表面法向量。$T_{g,k}$表示的是在第 K 帧的全局相机姿态估计。$\hat{u}$表示的是通过投影变换后的点的坐标。u表示的是原始图像中的点的坐标。$M_k(u)$这是一个判断函数，当其等于 1 时，表示点u在第 k 帧中被成功测量并且被认为是有效的。ε_d这是一个距离阈值，用来确定两个表面顶点的位置是否足够接近。ε_θ这是一个角度阈值，用来确定两个表面法向量的角度是否足够接近。$E(T_{g,k})$误差函数。

3.2.2 投影数据关联

由于第一帧不需要进行配准，因此我们在接下来的步骤中将对第 k-1 帧和第 k 帧（k＞2）进行配准。

```
// 投影数据关联，对数据进行初筛，不相关初始化为false
bool correspondence_found = false;

if (x < cols && y < rows) {
        // step 1，将当前帧的顶点坐标转换到世界坐标系下 Pw = Rwc * Pc + twc
          Matf31 vertex_current_global = rotation_current * vertex_current +
translation_current;

        // 这个顶点在上一帧相机坐标系下的坐标 Pc(k-1) = Rcw(k-1) * (Pw - twc(k-1))
        Matf31 vertex_current_camera =
                        rotation_previous_inv * (vertex_current_global -
translation_previous);

        // 得到(u, v)坐标
        point_x = __float2int_rd(
                        vertex_current_camera.x() * cam_params.focal_x /
vertex_current_camera.z() +
                cam_params.principal_x + 0.5f);
        point_y = __float2int_rd(
                        vertex_current_camera.y() * cam_params.focal_y /
vertex_current_camera.z() +
                cam_params.principal_y + 0.5f);

        // 取出对应的法向量和顶点
        Matf31 normal_previous_global;
                                        normal_previous_global.x()          =
normal_map_previous.ptr(point_y)[point_x].x;

        Matf31 vertex_previous_global;
                                        vertex_previous_global.x()          =
```

```
vertex_map_previous.ptr(point_y)[point_x].x;
                                        vertex_previous_global.y()          =
vertex_map_previous.ptr(point_y)[point_x].y;
                                        vertex_previous_global.z()          =
vertex_map_previous.ptr(point_y)[point_x].z;

        // 距离检查，如果顶点距离相差太多则认为不是正确的点
                const    float    distance    =    (vertex_previous_global    -
vertex_current_global).norm();
        if (distance <= distance_threshold) {
            // 由于 |axb|=|a|·|b|·sinα，  通过这个来判断sinα <= angle_threshold
                                            const       float       sine       =
normal_current_global.cross(normal_previous_global).norm();
            if (sine <= angle_threshold) {
                // 认为通过检查，保存关联结果和产生的数据
                n = normal_previous_global;
                d = vertex_previous_global;
                s = vertex_current_global;

                correspondence_found = true;
            }
        }
    }
}
```

首先，我们拥有第 K 帧经过 Surface Measurement 得到的 vertex map 和 normal map。我们取出 vertex map 中的一个点。在上一次迭代时，我们得到位姿转换矩阵。这里，我们直接拿过来对当前帧的点进行转换，将它转换到上一帧图像坐标系下观察到的 3D 点。接着，我们利用相机内参 K 矩阵逆变换，将它映射回(u, v)坐标。注意，我们对于保留了上一帧最终的结果 vertex map 和 normal map 二维数组，所以我们将(u, v)代入，取出这个坐标系下的 vertex 的 normal 值。到此，我们有了当前帧的一个点的信息和上一帧潜在关联点的信息，接下来，通过两个预设的参数 distance threshold 和 angle threshold 判断这两个点是否真的关联。具体而言，我们计算两个点的欧式距离是否小于 distance threshold, 而 normal 夹角是否小于 angle threshold，具体通过正弦值表达。注意，这两个预设参数值需要根据实际的模型手动进行适当调整。

我们使用 Surface Measurement 方法得到了第 K 帧的 vertex map 和 normal map。从 vertex map 中选取一个点，并利用上一次迭代得到的$T_{g,k-1}$将其从当前帧转换到上一帧的图像坐标系中观察到的点。然后，应用相机内参 K 矩阵的逆变换将该点映射回(u, v)

坐标。注意，我们在二维数组中保留了上一帧的最终结果 vertex map 和 normal map，因此可以使用坐标(u, v)来提取该坐标系下的 vertex 的 normal 值。这样，我们获取了当前帧中一个点的信息以及上一帧中可能关联的点的信息。接下来，我们使用两个预先设定的参数，即距离阈值和角度阈值，来判断这两个点是否真正相关。具体而言，我们计算两个点之间的欧氏距离是否小于距离阈值，并判断其法线的夹角是否小于角度阈值（通过正弦值来表示）。需要注意的是，这两个预先设定的参数值需要根据实际模型进行适当调整。

3.2.3 ICP 配准

```
float row[7];

// 只有对成功匹配的点才会进行的操作
if (correspondence_found) {
    // s是投影数据关联中的vertex_current_global
    Matf31 s_cross_n = s.cross(n);
    row[0] = s_cross_n.x00;
    row[1] = s_cross_n.x10;
    row[2] = s_cross_n.x20;
    row[3] = n.x00;
    row[4] = n.x10;
    row[5] = n.x20;
    // 矩阵b中当前点贡献的部分
    row[6] = n.dot(d - s);
} else
                //      如果没有找到匹配的点,      或者说是当前线程的id不在图像区域中,就全都给0。相当于这个点就废了
       row[0] = row[1] = row[2] = row[3] = row[4] = row[5] = row[6] = 0.f;
```

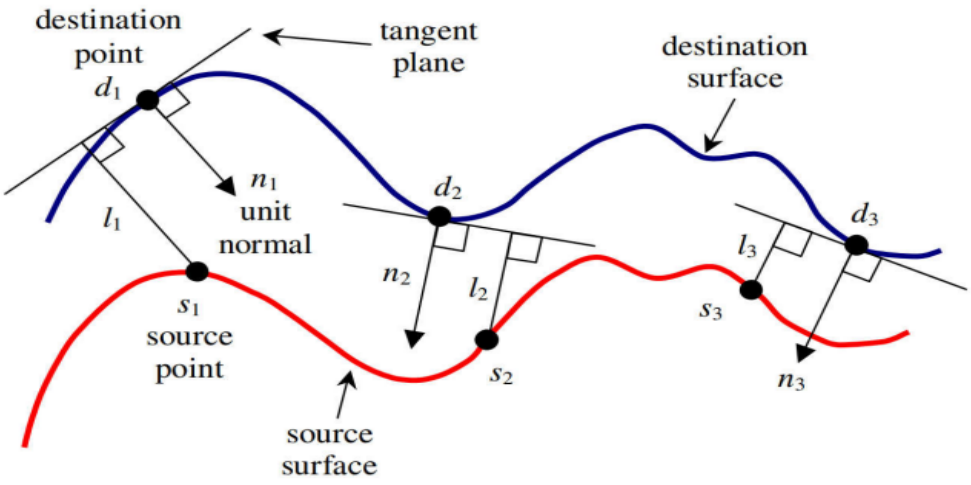

Figure 7: ICP配准示意图

$$\mathbf{E}(\mathbf{T}_{g,k}) = \sum_{u\in\Omega} \left\|(\mathbf{T}_{g,k}\dot{V}_k(\mathbf{u}) - \hat{V}_k)^{\mathrm{T}} N_{k-1}^{g}(\hat{\mathbf{u}})\right\|_2$$

其中，E表示所有投影关联点之间误差的累加和。$T_{g,k}$为将第k帧的点转移到全局坐标系下的位姿变换矩阵。V_k表示当前帧的相机坐标系下的点，$\widehat{V_k}$表示从上一帧最终得到的 vertex map 取出的关联点的全局坐标。$N_{k-1}^{g}(\hat{u})$表示从上一帧最终得到的 normal map 取出的法向量。表达式的含义即为将s_i和d_i两点间的距离向n_i投影得到的距离，这种方法被称为 point-to-plane。值得一提的是，point-to-plane 比 point-to-point（s_1和d_1之间的距离）具有更好的收敛效果，主要原因在于 point-to-plane 不仅具有两点之间距离的信息，同时 normal 是曲面在此处切平面的法线，故也包含曲面信息。

现在我们要做的就是找到使得E最小的$T_{g,k}$。

$$\begin{aligned}
\tilde{\mathbf{T}}_{qp} &= \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^{\mathrm{T}} & \tau \end{bmatrix} \\
&\approx \begin{bmatrix} 1 & \alpha\beta-\gamma & \alpha\gamma-\beta & t_x \\ \gamma & \alpha\beta\gamma+1 & \beta\gamma-\alpha & t_y \\ -\beta & \alpha & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
&\approx \begin{bmatrix} 1 & -\gamma & \beta & t_x \\ \gamma & 1 & -\alpha & t_y \\ -\beta & \alpha & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}
\end{aligned}$$

首先，我们将介绍位姿变换矩阵的表示方法。位姿变换矩阵 R 是一个 3x3 的矩阵，由三个变量 α、β 和 γ 组成，分别表示绕 x 轴、y 轴和 z 轴的旋转角度。

$$\mathcal{R}_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix}$$

$$\mathcal{R}_y(\beta) = \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix}$$

$$\mathcal{R}_z(\gamma)=\begin{bmatrix}\cos\gamma & -\sin\gamma & 0\\ \sin\gamma & \cos\gamma & 0\\ 0 & 0 & 1\end{bmatrix}$$

在 Kinect 相机中，例如 30FTS，由于旋转角度较小，我们可以通过将上面三个旋转矩阵相乘，并利用 sinx ≈ x、cosx ≈ 1 以及高阶无穷小近似来近似计算得到位姿变换矩阵 R。

矩阵 t = (tx, ty, tz)表示平移矩阵，而最后一行[0, 0, 0, 1]表示齐次坐标。将经过近似处理的位姿变换矩阵 T 代入以下表达式中进行展开，即可得到结果。

$$\begin{aligned}\left(\tilde{\mathbf{T}}_{qp}\cdot\mathbf{p}_i-\mathbf{q}_i\right)\cdot\mathbf{n}_i &= \left(\tilde{\mathbf{T}}_{qp}\cdot\begin{bmatrix}p_{ix}\\ p_{iy}\\ p_{iz}\\ 1\end{bmatrix}-\begin{bmatrix}q_{ix}\\ q_{iy}\\ q_{iz}\\ 1\end{bmatrix}\right)\cdot\begin{bmatrix}n_{ix}\\ n_{iy}\\ n_{iz}\\ 0\end{bmatrix}\\ &= \alpha\left(n_{iz}p_{iy}-n_{iy}p_{iz}\right)+\beta\left(n_{ix}p_{iz}-n_{iz}p_{ix}\right)+\gamma\left(n_{iy}p_{ix}-n_{ix}p_{iy}\right)+\\ &\quad t_x n_{ix}+t_y n_{iy}+t_z n_{iz}-\\ &\quad \left(n_{ix}q_{ix}+n_{iy}q_{iy}+n_{iz}q_{iz}-n_{ix}p_{ix}-n_{iy}p_{iy}-n_{iz}p_{iz}\right)\end{aligned}$$

令

$$\begin{cases}a_{i1}=n_{iz}p_{iy}-n_{iy}p_{iz}\\ a_{i2}=n_{ix}p_{iz}-n_{iz}p_{ix}\\ a_{i3}=n_{iy}p_{ix}-n_{ix}p_{iy}\\ b_i=n_{ix}q_{ix}+n_{iy}q_{iy}+n_{iz}q_{iz}-n_{ix}p_{ix}-n_{iy}p_{iy}-n_{iz}p_{iz}\\ \mathbf{x}=\left[\alpha,\beta,\gamma,t_x,t_y,t_z\right]^T\end{cases}$$

因此，将表达式转换为矩阵形式如下：

$$\begin{bmatrix}a_{11} & a_{12} & a_{13} & n_{1x} & n_{1y} & n_{1z}\\ a_{21} & a_{22} & a_{23} & n_{2x} & n_{2y} & n_{22}\\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots\\ a_{N1} & a_{N2} & a_{N3} & n_{Nx} & n_{Ny} & n_{Nz}\end{bmatrix}.\mathbf{x}-\begin{bmatrix}b_1\\ b_2\\ \vdots\\ b_N\end{bmatrix}=\mathbf{0}$$

记

$$A=\begin{bmatrix}a_{11} & a_{12} & a_{13} & n_{1x} & n_{1y} & n_{1z}\\ a_{21} & a_{22} & a_{23} & n_{2x} & n_{2y} & n_{2z}\\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots\\ a_{N1} & a_{N2} & a_{N3} & n_{Nx} & n_{Ny} & n_{Nz}\end{bmatrix}$$

$$b=[b_1 \quad b_2 \quad \cdots \quad b_N]^T$$

故最终表达式为

$$E(T_{g,k}) = \| Ax - b \|_2 = (Ax-b)^T(Ax-b) = x^TA^TAx - 2b^TAx + b^Tb$$

通过将上面的表达式对 x 求导，并令导数为 0 得到：

$$A^TAx = A^Tb$$

记$C = A^TA$，$d = A^Tb$，原式即为$Cx = d$。

需要特别指出的是，在一个大小为 640 x 480 的深度图像中，总共有 307,200 个点。因此，矩阵A的维度是$307,200$ x 6。传统的 SVD 分解方法用于求解非线性最小二乘问题时，需要对A^TA与AA^T进行正交相似对角化。由于AA^T的维度为 $307,200$ x $307,200$ 中，这对计算机而言计算量非常庞大。而在方程$A^TAx = A^Tb$中，A^TA是一个 6x6 的矩阵，A^Tb是一个 6x1 的矩阵，这为通过 LU 分解求解 x 提供了极大的便利。值得注意的是，计算A^TA和A^Tb的过程也是耗时的。因此，在编程实现时，可以通过对方程进行变形和观察，以简化计算过程。

注意到：

$$\mathbf{p}_i = \begin{bmatrix} p_{ix} \\ p_{iy} \\ p_{iz} \end{bmatrix} \quad \mathbf{n}_i = \begin{bmatrix} n_{ix} \\ n_{iy} \\ n_{iz} \end{bmatrix} \quad \mathbf{p}_i \times \mathbf{n}_i = \begin{bmatrix} n_{iz}p_{iy} - n_{iy}p_{iz} \\ n_{ix}p_{iz} - n_{iz}p_{ix} \\ n_{iy}p_{ix} - n_{ix}p_{iy} \end{bmatrix} = \begin{bmatrix} a_{i1} \\ a_{i2} \\ a_{i3} \end{bmatrix}$$

故 C 和 d 可化为

$$\begin{aligned} C &= A^TA \\ &= \begin{bmatrix} \mathbf{p}_1 \times \mathbf{n}_1 & \cdots & \mathbf{p}_N \times \mathbf{n}_N \\ \mathbf{n}_1 & \cdots & \mathbf{n}_N \end{bmatrix} \begin{bmatrix} (\mathbf{p}_1 \times \mathbf{n}_1)^{\mathrm{T}} & \mathbf{n}_1^{\mathrm{T}} \\ \vdots & \vdots \\ (\mathbf{p}_N \times \mathbf{n}_N)^{\mathrm{T}} & \mathbf{n}_N^{\mathrm{T}} \end{bmatrix} \\ &= \sum_{i=0}^{N} \begin{bmatrix} (\mathbf{p}_i \times \mathbf{n}_i)(\mathbf{p}_i \times \mathbf{n}_i)^c & (\mathbf{p}_i \times \mathbf{n}_i)\mathbf{n}_i^c \\ \mathbf{n}_i(\mathbf{p}_i \times \mathbf{n}_i)^{\mathrm{T}} & \mathbf{n}_i\mathbf{n}_i^{\mathrm{T}} \end{bmatrix} \\ &= \sum_{i=0}^{N} \begin{bmatrix} \mathbf{p}_i \times \mathbf{n}_i \\ \mathbf{n}_i \end{bmatrix} \begin{bmatrix} (\mathbf{p}_i \times \mathbf{n}_i)^{\mathrm{T}} & \mathbf{n}_i^{\mathrm{T}} \end{bmatrix} \end{aligned}$$

$$
\begin{aligned}
d &= A^T b \\
&= \begin{bmatrix} \mathbf{p}_1 \times \mathbf{n}_1 & \cdots & \mathbf{p}_N \times \mathbf{n}_N \\ \mathbf{n}_1 & \cdots & \mathbf{n}_N \end{bmatrix} \begin{bmatrix} (\mathbf{q}_1 - \mathbf{p}_1) \cdot \mathbf{n}_1 \\ \vdots \\ (\mathbf{q}_N - \mathbf{p}_N) \cdot \mathbf{n}_N \end{bmatrix} \\
&= \begin{bmatrix} \sum_{i=0}^{N} (\mathbf{p}_i \times \mathbf{n}_i) ((\mathbf{q}_i - \mathbf{p}_i) \cdot \mathbf{n}_i) \\ \sum_{i=0}^{N} \mathbf{n}_i ((\mathbf{q}_i - \mathbf{p}_i) \cdot \mathbf{n}_i) \end{bmatrix} \\
&= \sum_{i=0}^{N} \begin{bmatrix} (\mathbf{p}_i \times \mathbf{n}_i) ((\mathbf{q}_i - \mathbf{p}_i) \cdot \mathbf{n}_i) \\ \mathbf{n}_i ((\mathbf{q}_i - \mathbf{p}_i) \cdot \mathbf{n}_i) \end{bmatrix}
\end{aligned}
$$

经仔细观察，我们可以发现实际上只需要这三个变量：

$$
\begin{cases}
\mathbf{p}_i \times \mathbf{n}_i \\
\mathbf{n}_i \\
((\mathbf{q}_i - \mathbf{p}_i) \cdot \mathbf{n}_i)
\end{cases}
$$

而前两个是 3 维向量，第三个是一个标量，所以组装起来就是 7 个值，这也是代码中 float row[7]的由来。算出这三个变量值，我们就可以组装出C_i, d_i。同时，C_i, d_i的表达式只与p_i和q_i有关，所以每一项都可以并行化求出。

而前两个变量是 3 维向量，第三个变量是一个标量，因此总共有 7 个值。这也是代码中使用 float row[7]的原因。通过计算这三个变量的值，我们可以组装出C_i和d_i。值得注意的是，C_i和d_i的表达式仅与p_i和q_i有关，因此每个变量的计算都可以并行进行。

3.2.4 规约求和

```
// 在每个 Block 已经完成累加的基础上，进行全局的归约累加
__global__
void reduction_kernel(PtrStep<double> global_buffer, const int length,
PtrStep<double> output)
{
    double sum = 0.0;

    // 每个线程对应一个 block 的某项求和的结果，获取之
    for (int t = threadIdx.x; t < length; t += 512)
        sum += *(global_buffer.ptr(blockIdx.x) + t);
```

```
    __shared__ double smem[512];

    // 注意超过范围的线程也能够执行到这里，上面的循环不会执行，sum=0，因此保存到 smem 对后面的归约过程没有影响
    smem[threadIdx.x] = sum;

    // 512个线程都归约计算
    reduce<512>(smem);

    // 第0线程负责将每一项的最终求和结果进行转存
    if (threadIdx.x == 0)
        output.ptr(blockIdx.x)[0] = smem[0];
  };
```

$$C = \sum_{i=0}^{N} \begin{bmatrix} \mathbf{p}_i \times \mathbf{n}_i \\ \mathbf{n}_i \end{bmatrix} \begin{bmatrix} (\mathbf{p}_i \times \mathbf{n}_i)^{\mathrm{T}} & \mathbf{n}_i^{\mathrm{T}} \end{bmatrix}$$

$$d = \sum_{i=0}^{N} \begin{bmatrix} (\mathbf{p}_i \times \mathbf{n}_i)\big((\mathbf{q}_i - \mathbf{p}_i) \cdot \mathbf{n}_i\big) \\ \mathbf{n}_i\big((\mathbf{q}_i - \mathbf{p}_i) \cdot \mathbf{n}_i\big) \end{bmatrix}$$

在上述过程中，我们仅并行计算了单个点的C_i和D_i，而我们需要将它们加和起来。值得注意的是，加和的过程是相互独立的，因此我们可以采用规约和配对加和的方法，类似于排序算法中的合并（merge）思想。

以八个线程为例，图示如下：

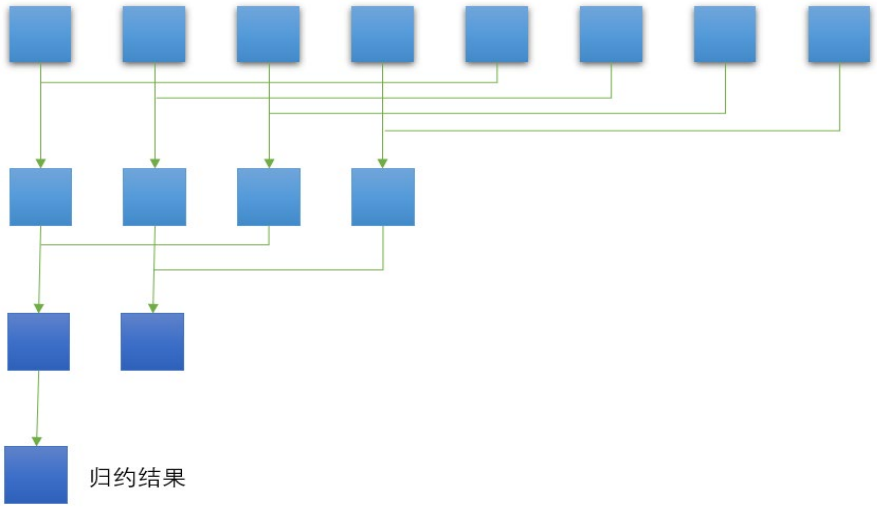

Figure 8: 规约求和示意图

到此，我们就可以得到C和d，接着我们利用 LU 分解求得x，即$[\alpha，\beta，\gamma，t_x, t_y, t_z]$，然后组装出相邻两帧的位姿变换矩阵。最后，将它左乘原来的矩阵T，即可得到新的$T_{g,k}$。

$$T_{g,k} = T \times T_{g,k-1}$$

3.3 Surface Reconstruction

3.3.1 TSDF 模型

基于截断的带符号距离函数，是一种常见的在 3D 重建中计算隐势面的方法。著名的 Kinfusion 就是才用$TSDF$来构建空间体素的，通过求每个体素的$TSDF$值，然后再使用之前提到的 Marching Cube 来提取表面的。

$TSDF$是在SDF进行改进的，是在SDF提出了截断距离，具体内容我们们在下面讲 ，很简单的。SDF 是在 2003 年由 S Osher 提出。在拥有大内存的显卡并行计算的情况下，使用$TSDF$可以做到实时的重建效果，获得了很多方面的落地使用。

下面我们一起来看一下，TSDF 的具体的算法思路。

用一个大的空间（Volume）作为要建立的三维模型，这个空间可以完全包括我们的模型，Volume 由许多个小的体素（voxel）组成，每个 voxel 对应空间中一个点，这个点我们用两个量来评价：

基于截断的带符号距离函数（Truncated Signed Distance Function，$TSDF$）是在 3D 重建中常用的方法之一。通过计算每个体素的$TSDF$值，再利用 Marching Cubes 算法提取表面。$TSDF$是在SDF的基础上进行改进的，其中SDF（Signed Distance Function）是由 S. Osher 于 2003 年提出的。在拥有大内存且支持并行计算的显卡上，使用$TSDF$能够实现实时重建效果，并在许多领域得到广泛应用。

下面我们将一起探讨$TSDF$算法的具体思路。

$TSDF$算法使用一个大的空间（Volume）作为要建立的三维模型，该空间完全包含我们的模型，并由许多小的体素（voxel）组成。每个体素对应空间中的一个点，并通过以下两个量来描述该点：

1.对于给定的 voxel，我们记$d(\vec{x})$，我们将该距离称为$SDF(x)$（$d(\vec{x})$），表示带符号的距离。

2.在进行体素更新时，我们引入权重 w 作为调整因子。

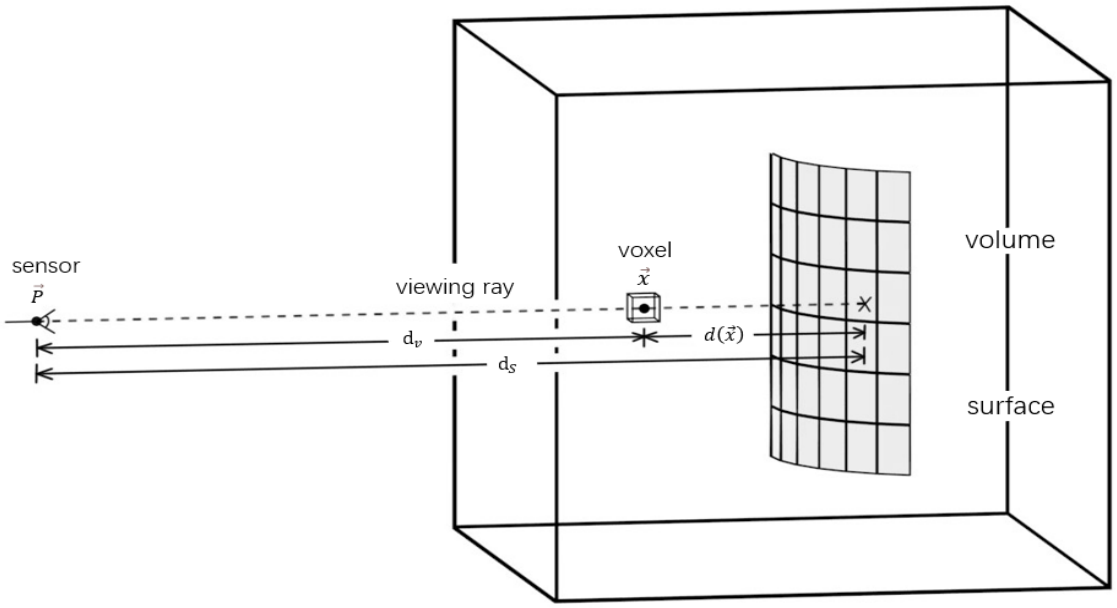

Figure 9: TSDF模型示意1

我们有真实表面与相机之间的深度值(d_s)，同时我们可以根据当前遍历到的 x 坐标计算出d_v值。因此，SDF值可以表示为 $d(\vec{x}) = d_s - d_v$。当$d(\vec{x})$大于零时，表示该体素位于真实表面之前；当$d(\vec{x})$小于零时，表示该体素位于真实表面之后。为了进行有效处理，我们对相机前后的距离进行限制，因为距离过远的情况下，真实表面的概率也较小，因此我们可以将其忽略。

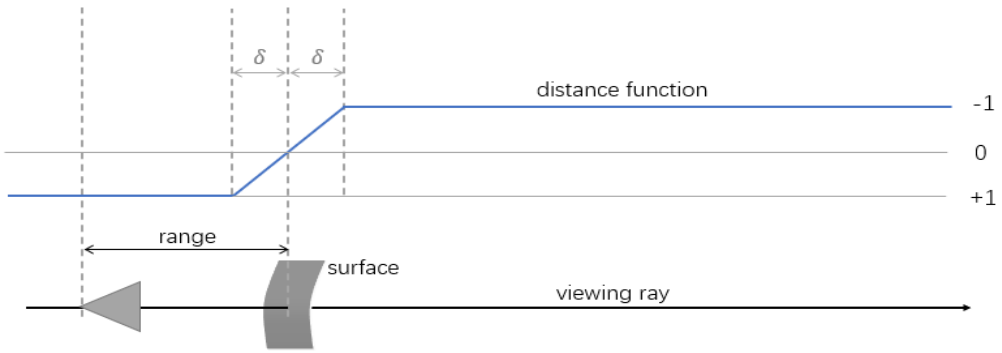

Figure 10: TSDF模型示意2

目前，我们已经获取了所有的SDF值，这些值存储和处理了每个空间点与物体表面的距离，即使是那些距离表面较远的点也包括在内。这导致了处理的数据量增加，同时也增加了存储和计算的负担。SDF对整个空间进行了均等处理，而没有专注于物体表面周围的区域。因此，相较于$TSDF$，SDF在物体表面附近的重建结果不够精确。在$TSDF$中，SDF值被截断到某个最大距离以内，超过这个距离的点的距离值被设置为该最大距离。这样一来，我们不再需要关注那些距离物体表面较远的区域，同时也节约了存储和计算资源。

3.3.2 计算SDF值

```
// 根据CUDA的block和thread ID计算体素在体素网格中的x坐标
const int x = blockIdx.x * blockDim.x + threadIdx.x;
// 根据CUDA的block和thread ID计算体素在体素网格中的y坐标
const int y = blockIdx.y * blockDim.y + threadIdx.y;

// 检查x或y坐标是否超出体素网格的范围
if (x >= volume_size.x || y >= volume_size.y)
    return;

// 遍历体素网格中的z坐标
for (int z = 0; z < volume_size.z; ++z) {
    // Step 1, 将体素坐标转换为世界坐标，计算3D位置
    const Matf31 position((static_cast<float>(x) + 0.5f) * voxel_scale,
                          (static_cast<float>(y) + 0.5f) * voxel_scale,
                          (static_cast<float>(z) + 0.5f) * voxel_scale);

    // Step 2,通过旋转和平移变换，将3D位置从世界坐标系转换到相机坐标系
    Matf31 camera_pos = rotation * position + translation;

    // 如果这个位置在相机的后面，则跳过这个位置
    if (camera_pos.z() <= 0)
        continue;

    // Step 3,将相机坐标系下的3D位置投影到图像平面，得到2D坐标(u, v)
      const int u = __float2int_rn(camera_pos.x() / camera_pos.z() *
cam_params.focal_x + cam_params.principal_x);
      const int v = __float2int_rn(camera_pos.y() / camera_pos.z() *
cam_params.focal_y + cam_params.principal_y);

    // 如果2D坐标超出了图像的范围，则跳过这个像素
    if (u < 0 || u >= depth_image.cols || v < 0 || v >= depth_image.rows)
```

```
        continue;

    // 从深度图像中获取这个像素的深度值
    const float depth = depth_image.ptr(v)[u];

    // 如果深度值非正，则跳过这个像素
    if (depth <= 0)
        continue;

    // Step 4, 用于计算投影，计算光线方向向量(x/z, y/z, 1)的模
    const Matf31 xylambda(
            (u - cam_params.principal_x) / cam_params.focal_x,
            (v - cam_params.principal_y) / cam_params.focal_y,
            1.f);

    const float lambda = xylambda.norm();

    // Step 5,计算这个位置的SDF值
    const float sdf = (-1.f) * ((1.f / lambda) * camera_pos.norm() - depth);
}
```

为了便于理解上述代码，我们将通过下图进行说明：

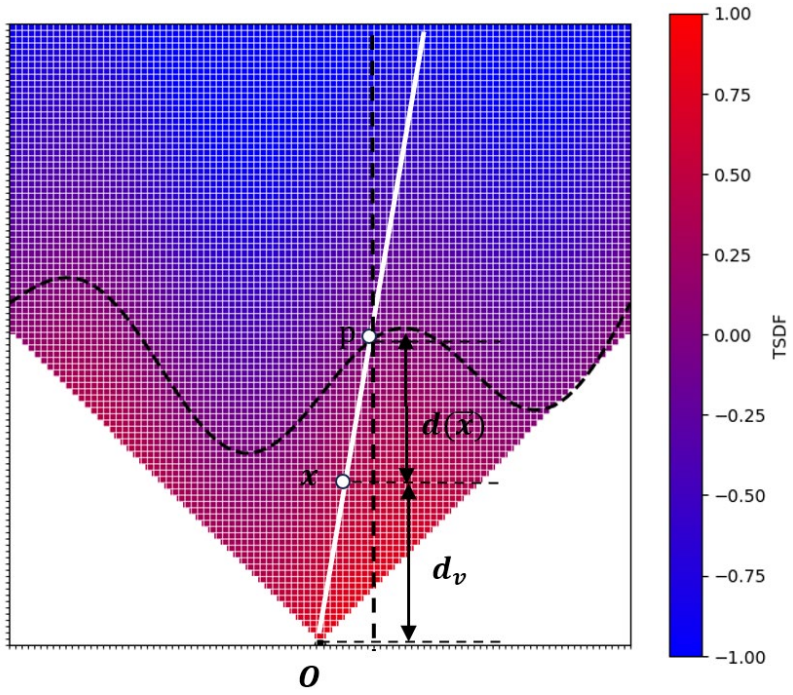

Figure 11: SDF值计算示意图

图中的黑色曲线表示物体的截面，O 表示相机光心，着色部分表示视场范围，$TSDF$值通过颜色渐变来表示。沿着相机光心 O 和体素 x 之间绘制了一条直线（白色粗

线），该直线与物体的截面在点 p 处相交。在代码中，Step 1 中的体素坐标已经转换为世界坐标系下的坐标，即 x 的世界坐标。Step 2 利用相机位姿的逆矩阵，将世界坐标系下的点转换到相机坐标系下。在 Step 3 中，我们将坐标使用相机内参进行反投影，以获得像素坐标(u, v)，然后读取深度值。之前提到过 Kinect 传感器的深度值实际上是 Z 轴上的投影。需要注意的是，实际操作中，相机、点 x 和点 p 并不一定共线，这个图示仅用于帮助理解。这个过程的思想本质上类似于位姿估计中的投影数据关联。由于 Camera_pos.norm()表示的是 Ox 向量，而我们需要求解的是点 x 到相机光心的深度d_v，因此我们需要在 Z 轴上进行投影。这也是为什么在 Step 4 中计算 lambda 的原因。最后，Step 5 中的代码用于计算图中的$d(\vec{x})$值，即 SDF 值。

3.3.3 计算当前TSDF并更新全局TSDF值

```
if (sdf >= -truncation_distance) {
    // step 1 计算当前次观测得到的 TSDF 值
    const float new_tsdf = fmin(1.f, sdf / truncation_distance);

    // step 2 更新 TSDF 值和权重值
    const int add_weight = 1;

     const float updated_tsdf = (current_weight * current_tsdf + add_weight *
new_tsdf) /
                                (current_weight + add_weight);

       const int new_weight = min(current_weight + add_weight, MAX_WEIGHT);
```

对SDF进行截断以获得$TSDF$的过程中，截断距离被视为体素 x 到对应截面点 P 的深度差值的阈值。当体素 x 与截面对应点 P 的距离非常远时，即SDF值大于阈值 t 或小于负阈值-t 时，$TSDF$数值被截断为 1 或-1。当体素 x 与截面对应点 P 的距离相对较近时，即SDF值位于[-t,t]范围内时，$TSDF$值计算为 SDF 值除以截断距离，使得$TSDF$值处于[-1,1]之间。

在构建$TSDF$时，权重用于衡量每个像素或体素对$TSDF$的贡献程度。每一帧的权重被设定为 1，$TSDF$的更新是通过将前 k-1 帧和当前帧进行融合来实现的，其中前一帧也是通过融合前 k-2 帧得到的，这是一个增量式的过程。从本质上讲，最终的$TSDF$是所有帧的$TSDF$值加权平均的结果。代码中的 MAX_WEIGHT 是预先设定的参数，旨

在防止随着帧数的累积，平均$TSDF$的权重变得越来越大，从而导致后续帧的$TSDF$值几乎没有对平均$TSDF$的更新效果。

然后，根据以下公式对 TSDF 进行全局更新融合，并更新权重：

$$F_k(p) = \frac{W_{k-1}(p)F_{k-1}(p) + W_{R_k}(p)F_{R_k}(p)}{W_{k-1}(p) + W_{R_k}(p)}$$
$$W_k(p) = W_{k-1}(p) + W_{R_k}(p)$$

3.3.4 更新color值

```
if (sdf <= truncation_distance / 2 && sdf >= -truncation_distance / 2) {

    uchar3& model_color = color_volume.ptr(z * volume_size.y + y)[x];

    const uchar3 image_color = color_image.ptr(v)[u];

    // Color也是同理加权平均
    model_color.x = static_cast<uchar>(
            (current_weight * model_color.x + add_weight * image_color.x) /
            (current_weight + add_weight));
    model_color.y = static_cast<uchar>(
            (current_weight * model_color.y + add_weight * image_color.y) /
            (current_weight + add_weight));
    model_color.z = static_cast<uchar>(
            (current_weight * model_color.z + add_weight * image_color.z) /
            (current_weight + add_weight));
}
```

Color 值的更新要求更加严格，截断距离被设置为原来的一半。随后的 Color 融合与之前的过程类似，也是通过加权平均来实现的。

3.4 Surface Prediction

3.4.1 计算光线方向

```
// step 1 计算 raycast 射线, 以及应该在何处开始, 在何处结束
// 计算 Volume 空间范围
const float3 volume_range = make_float3(volume_size.x * voxel_scale,
                                        volume_size.y * voxel_scale,
```

```
                                        volume_size.z * voxel_scale);
//          计算当前的点和相机光心的连线,          使用的是在当前相机坐标系下的坐标;
由于后面只是为了得到方向所以这里没有乘Z
const Matf31 pixel_position(
        (x - cam_parameters.principal_x) / cam_parameters.focal_x,      // X/Z
        (y - cam_parameters.principal_y) / cam_parameters.focal_y,      // Y/Z
        1.f);                                                           // Z/Z

Matf31 ray_direction = (rotation * pixel_position);
ray_direction.normalize();
```

Ray casting 方法指的是对于一个像素平面，通过光心与每个像素点相连形成的光线进行处理。以 Figure .x 中的点 x 为例，光心表示为 O，像素点表示为 m。通过连接 O 和 m，我们得到了光线的方向，从而确定了光线的位置。我们的目标是找到这条光线上$TSDF$为 0 的位置，即物体表面所在的位置。

3.4.2 计算光线在何时出入 Volume 空间

```
__device__ __forceinline__
//                                                          求射线为了射入Volume,
在给定步长下所需要的最少的前进次数(也可以理解为前进所需要的时间)
float get_min_time(
    const float3&   volume_max,     // 体素的范围(真实尺度)
    const Matf31&  origin,          // 出发点, 也就是相机当前的位置
    const Matf31&  direction)       // 射线方向
{
       //   分别计算三个轴上的次数,   并且返回其中最大;   当前进了这个最大的次数之后,
三个轴上射线的分量就都已经射入volume了
    float txmin = ((direction.x() > 0 ? 0.f : volume_max.x) - origin.x()) /
direction.x();
    float tymin = ((direction.y() > 0 ? 0.f : volume_max.y) - origin.y()) /
direction.y();
    float tzmin = ((direction.z() > 0 ? 0.f : volume_max.z) - origin.z()) /
direction.z();

    return fmax(fmax(txmin, tymin), tzmin);
}

__device__ __forceinline__
//                                                          求射线为了射出Volume,
在给定步长下所需要的最少的前进次数(也可以理解为前进所需要的时间)
float get_max_time(const float3& volume_max, const Matf31& origin, const Matf31&
direction)
```

```
{
        //    分别计算三个轴上的次数，    并且返回其中最小.    当前进了这个最小的次数后，
三个轴上的射线的分量中就有一个已经射出了volume了
    float txmax = ((direction.x() > 0 ? volume_max.x : 0.f) - origin.x()) /
direction.x();
    float tymax = ((direction.y() > 0 ? volume_max.y : 0.f) - origin.y()) /
direction.y();
    float tzmax = ((direction.z() > 0 ? volume_max.z : 0.f) - origin.z()) /
direction.z();

    return fmin(fmin(txmax, tymax), tzmax);
}
```

这段代码定义了两个函数：get_min_time 和 get_max_time。这两个函数用于计算从相机光心出发的射线进入和离开$TSDF$体素空间所需的长度。我们将体素空间记为[0, volume_max.x]、[0, volume_max.y]和[0, volume_max.z]。以 x 轴为例，get_min_time 函数假设从 0 到 volume_max.x 的方向为正方向。因此，当 direction.x()大于 0 时，表示方向为正，射线的入射位置应该从 0 开始，即为 0 - origin.x()；反之，当 direction.x()小于 0 时，表示方向为负，射线的入射位置应该从 volume_max.x 开始，即为 volume_max.x - origin.x()。然而，尽管 x 轴进入了体素空间，但 y 轴和 z 轴未必进入。为了确保射线在三个轴方向上都能进入体素空间，我们需要选择三个轴方向上的最大步长。get_max_time 函数的原理与此类似。通过这两个函数，我们可以确定射线在体素空间中的起点和终点。进入体素空间时的距离结果将被赋值给 ray_length。

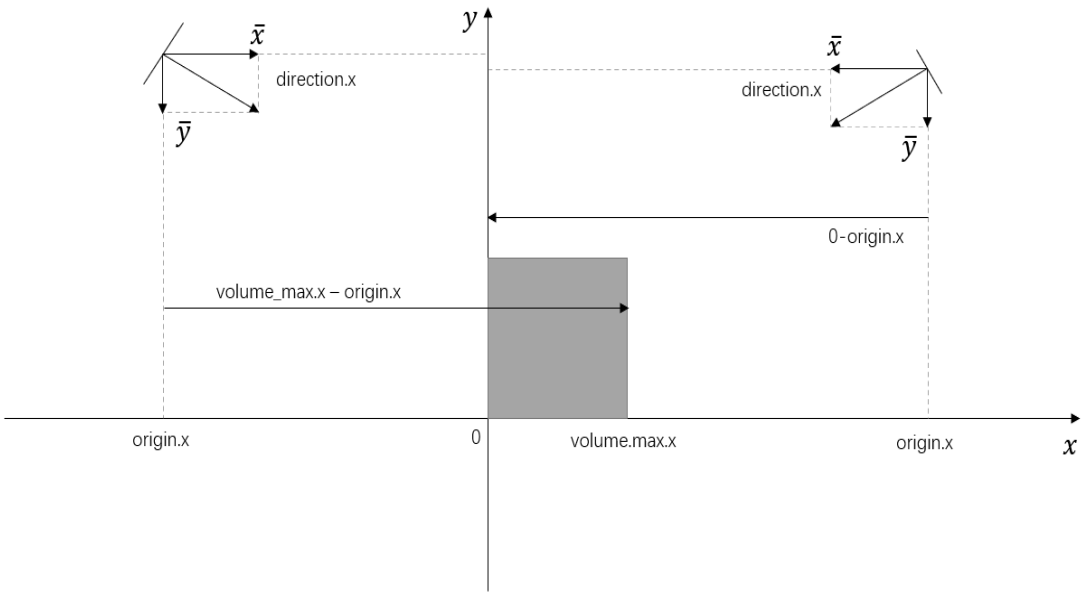

Figure 12: 表面估计示意图

3.4.3 寻找 TSDF 为 0 的 vertex

```
// 以步长为一半截断距离，迭代搜索，ray cast核心过程
for (; ray_length < max_search_length; ray_length += truncation_distance * 0.5f)
{

        // 计算当前次前进后，射线到达的体素id
              grid   =   ((translation   +   (ray_direction   *   (ray_length   +
truncation_distance * 0.5f))) / voxel_scale);


        // 保存上一次的 TSDF 值，用于进行下面的判断
        const float previous_tsdf = tsdf;
        // step 1 ***计算当前 Grid 处的 TSDF 值
        tsdf = static_cast<float>(tsdf_volume.ptr(
                                __float2int_rd(grid.at(2))   *   volume_size.y   +
__float2int_rd(grid.at(1)))[__float2int_rd(
                grid.at(0))].x) *
               DIVSHORTMAX;

        // step 3
        if (previous_tsdf < 0.f && tsdf > 0.f) //Zero-crossing from behind
            // 这种情况是从平面的后方穿出了
            break;
        if (previous_tsdf > 0.f && tsdf < 0.f) { //Zero-crossing
            // step 4
            const float t_star =
                    ray_length - truncation_distance * 0.5f * previous_tsdf /
(tsdf - previous_tsdf);
            // 计算射线和这个平面的交点. 下文简称平面顶点. vec3f 类型
            const auto vertex = translation + ray_direction * t_star;
        ...
  }
```

确定了起始和终止位置后，接下来使用 1/2 截断距离作为步长进行遍历。截断距离是指将SDF转换为$TSDF$时所使用的截断距离。对于射线上的每个步长点，在第一步中，我们使用最近邻插值法，即将其$TSDF$值设置为离其最近的体素中心的$TSDF$值。在遍历过程中，我们寻找$TSDF$发生变号的区间，即 Zero-crossing 区域。然而，我们不考虑$TSDF$由负值变为正值的情况，因为我们是从相机光心沿着射线向前遍历的，关注的是外部场景的成像效果。

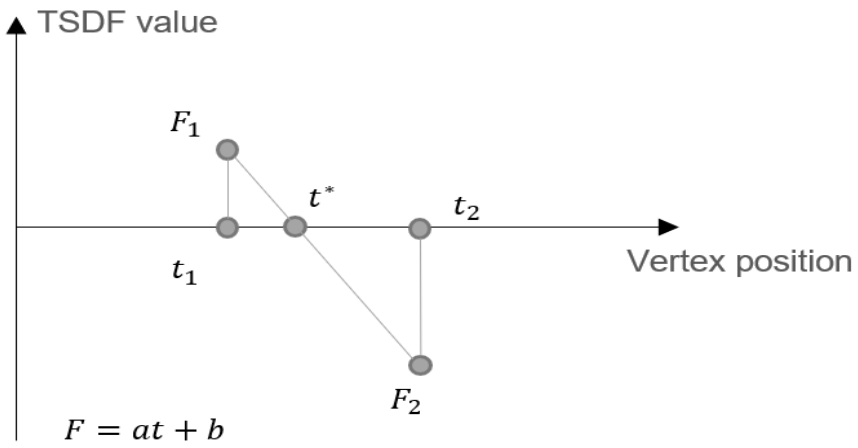

Figure 13: 寻找物体表面

在找到 Zero-crossing 的两个点后，假设$TSDF$值和坐标 t 之间存在线性关系，我们可以通过线性插值的思想计算t^*坐标。首先，我们知道直线经过(t_1, F_1)和(t_2, F_2)两个点，因此可以计算出直线的斜率 a 和截距 b：

$$a = \frac{F_2 - F_1}{t_2 - t_1}$$
$$b = F_1 - t_1 \frac{F_2 - F_1}{t_2 - t_1}$$

令$at^* + b = 0$，得

$$t^* = t_1 - F_1 \frac{t_2 - t_1}{F_2 - F_1}$$

换成步长Δt形式，即 step 4 中的表达式为

$$t^* = t - \frac{\Delta t F_t^+}{F_t^+ \Delta t - F_t^+}$$

3.4.4 计算法向量(以法向量 x 轴分量为例)

```
// step 1
shifted = location_in_grid;
shifted.x() += 1;
if (shifted.x() >= volume_size.x - 1)
    break;

const float Fx1 = interpolate_trilinearly(
```

```
    shifted,            // vertex 点在Volume的坐标滑动之后的点, Vec3fda
    tsdf_volume,        // TSDF Volume
    volume_size,        // Volume 的大小
    voxel_scale);       // 尺度信息

// step 2
shifted = location_in_grid;
shifted.x() -= 1;
if (shifted.x() < 1)
    break;
const float Fx2 = interpolate_trilinearly(shifted, tsdf_volume, volume_size,
voxel_scale);
normal.x() = (Fx1 - Fx2);
```

首先，让我们明确法向量的含义。$TSDF$与法向量存在直接关系，因为$TSDF$本质上是用于表示物体表面的数据结构。法向量用于确定$TSDF$值变化最大的方向，即计算$TSDF$函数的梯度即为法向量：

$$\nabla F = \left[\frac{\partial F}{\partial x}, \frac{\partial F}{\partial y}, \frac{\partial F}{\partial z}\right]^{\top}$$

然而，在体素空间中，我们并不知道 F 函数的具体表达式，我们只有每个离散体素块对应的$TSDF$值。因此，我们采用差分方法来估计梯度。差分方法的基本思想是通过在一个小的间隔上计算函数值的变化量来近似计算函数在该点的导数。由于dx、dy和dz相同，我们只关注梯度方向，所以只需将函数值之间的差异，而不需要除以dx、dy和dz。此外，我们还会对法向量进行标准化。对于x轴分量，我们通过在正向和负向移动一个体素的操作（即 Step 1 和 Step 2）来计算，如下图所示的情况：

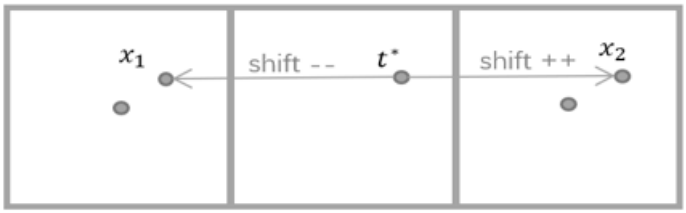

Figure 14: 差分方法示意

仔细观察可发现，偏移后的点并不恰好位于体素中心。对于法向量的精度要求而言，简单采用最近邻近似方法会引入较大的误差。因此，在这里我们采用三线性插值的方法，分别计算$TSDF(x_1)$和$TSDF(x_2)$。假设我们在三维空间中有一个点 P，它被八

个格点（即体素中心）所包围，我们知道这八个格点的函数值。我们的目标是通过这八个函数值的插值，得到点 P 对应的函数值。插值公式如下所示：

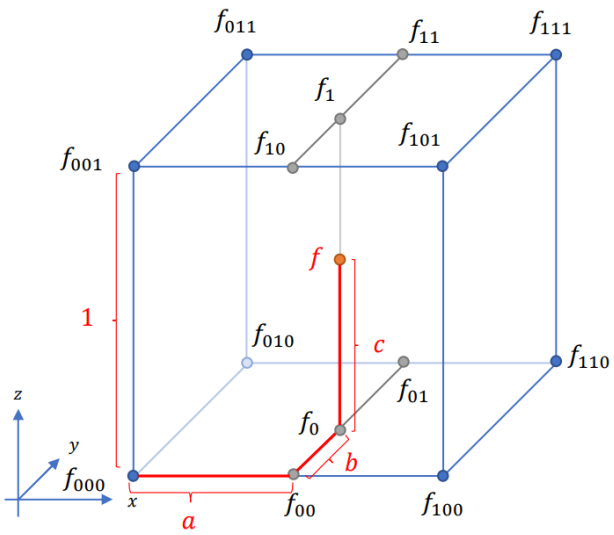

Figure 15: 三线性插值

$$f_{00} = (1-a)f_{000} + af_{100}$$

$$f_{01} = (1-a)f_{001} + af_{101}$$

$$f_{10} = (1-a)f_{010} + af_{110}$$

$$f_{11} = (1-a)f_{011} + af_{111}$$

$$f_0 = (1-b)f_{00} + bf_{01}$$

$$f_1 = (1-b)f_{10} + bf_{11}$$

$$f = (1-c)f_0 + cf_1$$

$TSDF(x_1)$ - $TSDF(x_2)$即为法向量在 x 轴上的分量。通过类似的方式处理 y 轴和 z 轴，我们可以得到该 t*点对应的完整法向量。到此为止，我们将所有射线得到的顶点和法向量存储在一个二维数组中，并将其返回给位姿估计模块。当新的一帧到来时，我们会重新进行位姿估计和$TSDF$更新，并进行循环迭代。通过这个过程，我们最终可以得到三维重建的结果。

4 功能测试

4.1 测试方法

在本研究中，我们主要通过将重建出的三维建模场景与原模型进行比较，来对重建效果进行评估。我们关注的主要是重建模型与原始模型中对应点之间的距离信息。

4.2 测试数据集、工具和参数

4.2.1 数据集

我们选用的测试数据集是 ICL-NUIM 数据集中的 Living Room。这个数据集提供了一个实际场景的三维模型，可以作为我们测试 KinectFusion 重建效果的基准。

4.2.2 工具与方法

评估过程中，我们使用了 CloudCompare 工具。首先，我们运用了 CloudCompare 的粗配准（manual registration）功能，这一功能使我们能够手动选择两个模型对应的几个点，从而实现大致的匹配。在进行粗配准之后，我们使用了 CloudCompare 的精配准（fine registration）功能，以 ICP（Iterative Closest Point）算法为基础，通过优化源点云的旋转和平移参数，实现了更精确的匹配。

在模型匹配完成后，我们利用 CloudCompare 的误差分析工具对重建模型和原始模型之间的对应点进行了距离计算，从而得到了详细的误差信息。在 CloudCompare 中采用的点云间距离计算方法默认采用“最近邻距离”算法。这种方法在比较点云中对每个点进行操作，通过在参考点云中搜索最近的点并计算它们的欧氏距离来确定距离。

通过上述步骤，我们对 KinectFusion 的重建效果进行了全面的测试和评估，从而得出了一些关于其效果的具体指标。这些信息对于我们深入理解 KinectFusion 的性能，以及如何改进算法有着重要的参考价值。

4.2.3 参数

表3.1　测试参数

参数名称	参数值	单位
宽度	640	像素
高度	480	像素
深度到实际世界的比例	5000	无
焦距x	481.2	无
焦距y	480.0	无
主点x	319.5	像素
主点y	239.5	像素
金字塔层数	3	层
双边滤波窗口大小	5	像素
深度截断距离	10000	毫米
ICP距离阈值	40	毫米
ICP角度阈值	10	度
ICP迭代次数（对于金字塔的每一层）	{10,5,4}	次
TSDF体素尺寸	10	毫米
TSDF体素数量	512*512*512	个
TSDF截断距离	25	毫米

4.3 测试结果

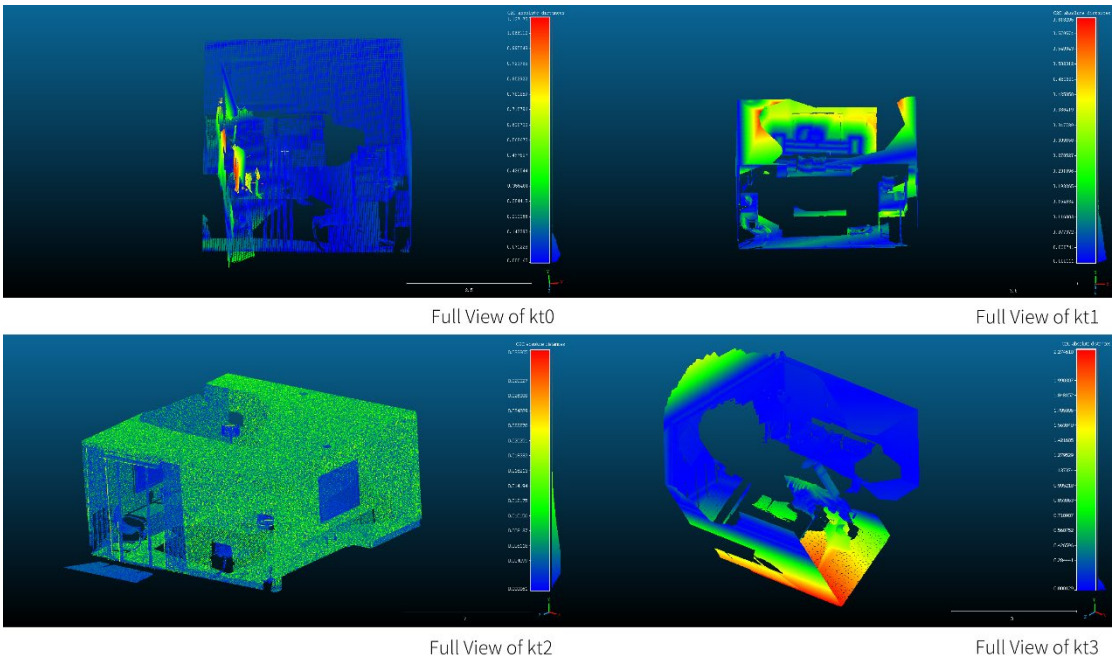

Figure 16： 重建效果概览

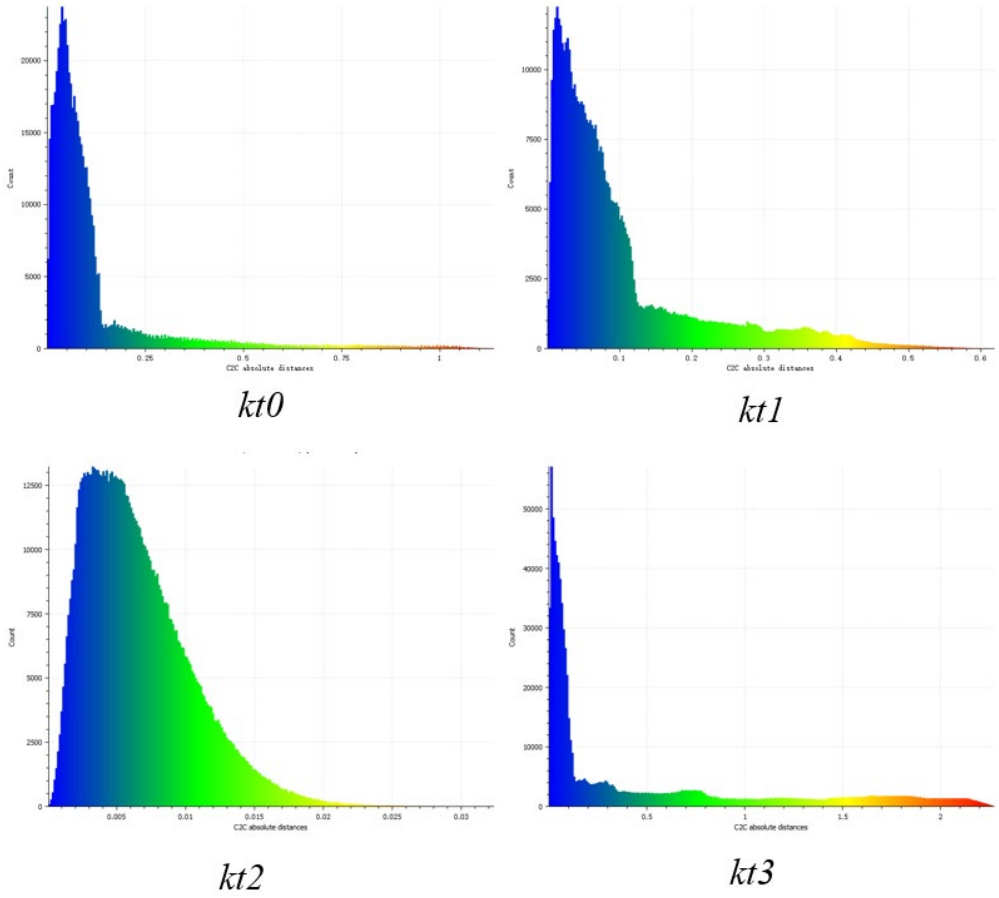

Figure 17： 误差分布直方图

表3.2　准确度测试结果

Error (m)	kt0	kt1	kt2	kt3
平均值	0.115926	0.101383	0.006561	0.506075
标准差	0.162509	0.10584	0.003837	0.642397
最小值	0	0	0	0
最大值	1.12381	0.632141	0.033698	2.24871

在新增的数据中，我们发现在 kt2 这一场景下，KinectFusion 的重建效果最好。在这个场景中，平均误差为 0.006561 米，误差分布的标准差为 0.003837 米，误差最大值为 0.033698 米。然后，kt0 和 kt1 的重建效果相对接近。平均误差分别为 0.115926 米和 0.101383 米，误差分布的标准差分别为 0.162509 米和 0.10584 米，误差最大值分别为 1.12381 米和 0.632141 米。从这些数据来看，整体来说，在 kt0、kt1、kt2 三个场景下，我们的 KinectFusion 实现的重建效果大部分能保持在较小的误差范围内，同时误差分布呈现一定的离散型。

相较而言，kt3 场景重建效果明显不如其他三个场景。在该场景中，重建结果的平均误差为 0.506075 米，标准偏差为 0.642397 米，这显示出该场景的误差分布的离散型较大。kt3 的误差最大值也达到了 2.24871 米。表现明显劣于其他三个场景。

综上所述，我们实现的 KinectFusion 在 kt0，kt1，kt2 三个场景下实现了展现了较为稳定的重建效果，然而在 kt3 场景下，重建效果不甚理想。在后面的问题分析部分，我们会对这一现象的原因进行深入分析。

5　问题分析

5.1　CUDA 占用率不高

原因在于 KinectFusion 为流水线类型的算法，当前帧的处理依赖于之前帧的处理结果。因此相比于使用高算力设备在短时间内处理大量图像序列，其更适合于使用低算力设备在长时间内处理实时图像信息。

在以下参数的情况下进行测试

表3.3 性能测试结果

参数	值	单位
显卡	GTX1650	-
图片数据集分辨率	640x480	像素
TSDF体素尺寸	10	毫米
TSDF体素数量	512x512x512	个
测试时间	184	秒
平均占用率	55.44%	-
平均每帧花费时间	33.9827	毫秒
每秒平均生成帧	29.4267	帧

在 TSDF 体素尺寸为 10 毫米，TSDF 体素数量为 512x512x512 的情况下，测试结果在 184 秒的样本时间内得出。可以看到，对于低算力设备如 GTX1650，在处理 640x480 分辨率的图片数据集时，使用 KinectFusion 算法的平均 CUDA 占用率约为 55.44%，每帧的平均处理时间约为 33.98 毫秒，这使得 KinectFusion 算法能够生成约 29.43 帧/秒的速度。这个帧数非常接近 KinectFusion 相机的 30 帧/秒的频率。同时表明该算法适合在长时间内处理实时图像信息，而非在短时间内处理大量图像序列。

与 Nerf 等算法相比，KinectFusion 算法的特性可能更适合于某些特定的应用场景。例如，在资源有限或处理能力较低的设备上，KinectFusion 的流水线类型处理方式可能更具优势。具体的选择应根据实际的应用需求和设备能力进行评估。

5.2 kt3 场景的重建效果较差

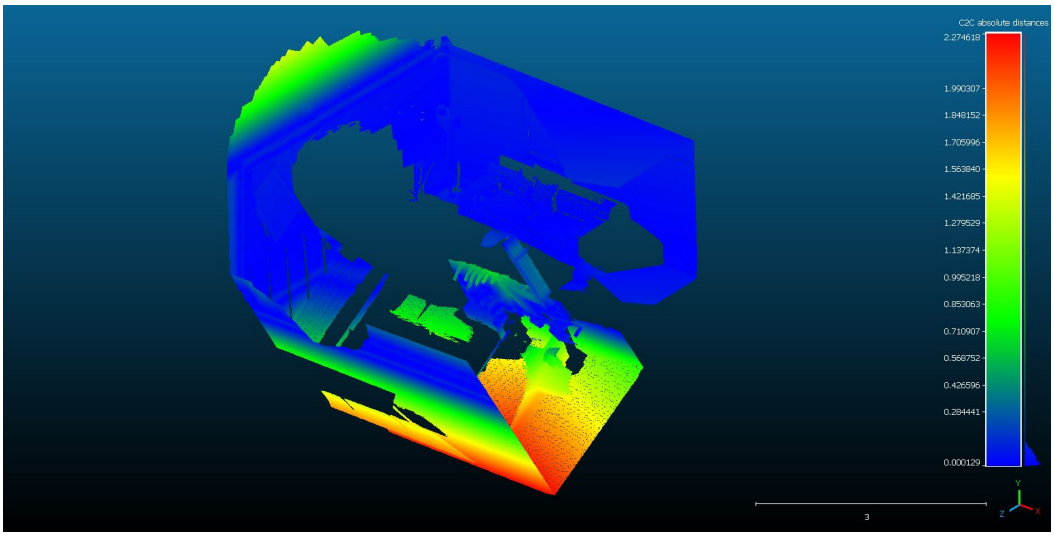

Figure 18： kt3 场景重建概览

在分析 kt3 场景的重建效果时，我们观察到许多墙壁和物体存在明显的错位情况。这种现象可能是由于场景内参照物的缺失导致的。

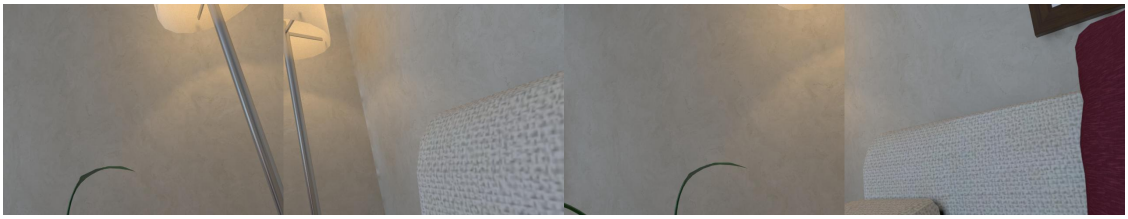

Figure 19： kt3 场景部分帧

在详细检查数据集后，我们发现在 kt3 场景的许多帧中，参照物相对较少，这在上图中可以清晰地看到。在这种参照物稀缺的情况下，KinectFusion 算法的配准过程可能会受到影响，从而无法准确进行。因为配准过程依赖于充足的参照物来确定摄像机的位置和方向，当参照物较少时，算法估计出的位姿可能会存在误差，进而导致重建结果出现错位。

基于上述发现，我们推测在使用 KinectFusion 算法进行场景重建时，应尽可能地确保每一帧中都包含足够的参照物。这不仅可以提高配准的准确性，也有助于减少错位等问题的发生。此外，这种观察也提醒我们在设计和采集数据集时，应尽可能选择或构造具有丰富参照物的场景，以便更好地支持场景的 3D 重建。

6 参考文献

[1] Izadi, Shahram, et al. "Kinectfusion: real-time 3d reconstruction and interaction using a moving depth camera." Proceedings of the 24th annual ACM symposium on User interface software and technology. 2011.

[2] Newcombe, Richard A., et al. "Kinectfusion: Real-time dense surface mapping and tracking." 2011 10th IEEE international symposium on mixed and augmented reality. Ieee, 2011.

[3] Izadi, Shahram, et al. "Kinectfusion: real-time dynamic 3d surface reconstruction and interaction." ACM SIGGRAPH 2011 Talks. 2011. 1-1.

[4] Whelan, Thomas, et al. "Kintinuous: Spatially extended kinectfusion." (2012).

[5] Tomasi, Carlo, and Roberto Manduchi. "Bilateral filtering for gray and color images." Sixth international conference on computer vision (IEEE Cat. No. 98CH36271). IEEE, 1998.

[6] Paris, Sylvain, et al. "Bilateral filtering: Theory and applications." Foundations and Trends® in Computer Graphics and Vision 4.1 (2009): 1-73.

[7] Adelson, Edward H., et al. "Pyramid methods in image processing." RCA engineer 29.6 (1984): 33-41.

[8] Low, Kok-Lim. "Linear least-squares optimization for point-to-plane icp surface registration." Chapel Hill, University of North Carolina 4.10 (2004): 1-3.

[9] Segal, Aleksandr, Dirk Haehnel, and Sebastian Thrun. "Generalized-icp." Robotics: science and systems. Vol. 2. No. 4. 2009.

[10] Ceradini, Daniel J., et al. "Progenitor cell trafficking is regulated by hypoxic gradients through HIF-1 induction of SDF-1." Nature medicine 10.8 (2004): 858-864.

[11] Kucia, Magda, et al. "Trafficking of normal stem cells and metastasis of cancer stem cells involve similar mechanisms: pivotal role of the SDF‐1‐CXCR4 axis." Stem cells 23.7 (2005): 879-894.

[12] Nießner, Matthias, et al. "Real-time 3D reconstruction at scale using voxel hashing." ACM Transactions on Graphics (ToG) 32.6 (2013): 1-11.

[13] Roth, Scott D. "Ray casting for modeling solids." Computer graphics and image processing 18.2 (1982): 109-144.

[14] Pfister, Hanspeter, et al. "The volumepro real-time ray-casting system." Proceedings of the 26th annual conference on Computer graphics and interactive techniques. 1999.

[15] Zhang, Zhengyou. "Microsoft kinect sensor and its effect." IEEE multimedia 19.2 (2012): 4-10.

[16] Han, Jungong, et al. "Enhanced computer vision with microsoft kinect sensor: A review." IEEE transactions on cybernetics 43.5 (2013): 1318-1334.