BPA 点云重建算法

冯古豪
2000013175

摘要

在这次大作业中，我在原有 lab2 的框架上实现了 BPA 点云重建算法，并且以在原有的 lab2 素材的基础上加入了一些更加复杂的新素材，来测试我实现的 BPA 算法的效果。最后，我还与 Open3D 库中实现的 BPA 算法进行了比较。

1 导引

使用物体的点云重建出物体表面的三角网格结构是图形学中的一个重要问题。现在点云重建主流的算法主要有 Alpha Shapes，Ball-Pivoting Algorithm(BPA) 以及 Poisson surface reconstruction 三种。其中，Ball-Pivoting Algorithm 和 Poisson surface reconstruction 这两种算法除了要求点云中的点包含最基本的坐标信息，还要求提供每个点的法向量。在实际的应用场景中，通过激光扫描得到的点云一般都包含法向量的信息。在这次大作业中，我实现的是 BPA 算法，主要参考了助教提供的论文。

2 BPA 算法原理

BPA 算法的思想很简单，对于一团点云，我们从远处飞来一个直径大于点云格间隙的球，那么球一定会击中其中三个点，我们将这三个点建立为一个三角形，然后我们以这个三角形的边为轴来旋转这个球，这样球就会击中其他点，我们根据这条边和新的点来建立新的三角形。我们可以不断进行类似的操作就能建立点云对应的三角网格，这就是 BPA 算法的核心思想，如图1所示。根据这个思想，我们可以得到 BPA 的具体算法流程。

2.1 数据结构

首先我们定义一系列数据结构。BPA 算法的输入为一团点云和我们使用的旋转球的半径，点云中的每一个点包含点的位置和法向量，我们将点 i 的位置和法向量分别记为 p_i 和 n_i，球的半径 r。首先我们要建立空间查询数据结构，将空间划分为一系列正方体格子，格子的边长为 $2r$，然后将点云中的每一个点都分配到对应的格子中，点云和空间

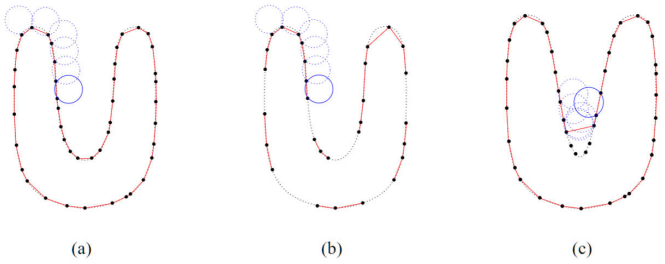

图 1: Ball Pivoting Algorithm in 2D

中球的交集就只需要去检验对应的格子中的点。为了实现的方便，我们在边的数据结构中，还包含了和边相对的顶点。每一条边有四种状态，分别是未使用，边界，激活，已处理。其中状态为未使用表示边还没有被算法处理，状态为边界的边表示这条边是三角网格的边界，状态为激活的边表示这条边可以用来作为轴进行旋转球操作，激活的边旋转完成后就标记为已处理。此外，我们建立一种数据结构 $front$，$front$ 中包含当前所有状态为激活的边。

2.2 基本操作

我们定义四种基本操作分别为 $Findseed$，$Pivotball$，$Join$，$Glue$。$Findseed$ 是为了找到旋转球开始的三角形，我们需要遍历所有目前没有处理过的点，找到能够分布在球的表面，且球内部没有其他的点的三个点，作为种子三角形来进行后续操作，并且将三角形的三条边标记为激活，加入到 $front$。$Pivotball$ 是对球进行旋转，输入为一条激活的边。我们需要找到在旋转过程中球所碰到的第一个点，并返回这个点。此外我们还要通过该点的法向量来判断此时球心是否在外侧，如果不在外侧，那么就舍弃这个点，继续寻找之后的点。$Join$ 操作输入是一条激活的边 e 包含点 p_1，p_2 和球旋转时碰到的第一个点 p，我们生成一个三角形 p_1pp_2，将边 e 标记为已处理，并从 $front$ 中删除，然后将边 p_1p，pp_2 标记为激活并加入 $front$。$glue$ 操作输入是一对反向边，它将这对反向边标记为已处理并从 $front$ 删除。

2.3 算法流程

BPA 算法的伪代码如图2下。我们找到种子三角形，不断进行旋转操作，直到 $front$ 中不存在激活的边。不断重复之前的流程，直到我们找不到种子三角形，这时候我们就能够得到重建的三角网格。

3 实验效果和分析

在这一部分中，我主要着重分析球的半径对 BPA 重建效果的影响，此外，我还比较了 Open3D 库函数中实现的 BPA 算法和我的实现的 BPA 算法。

```
Algorithm BPA(S, ρ)

1.  while (true)

2.      while (e_(i,j) = get_active_edge(F))
3.          if (σ_k = ball_pivot(e_(i,j)) &&
                    (not_used(σ_k) || on_front(σ_k)))
4.              output_triangle(σ_i, σ_k, σ_j)
5.              join(e_(i,j), σ_k, F)
6.              if (e_(k,i) ∈ F) glue(e_(i,k), e_(k,i), F)
7.              if (e_(j,k) ∈ F) glue(e_(k,j), e_(j,k), F)
8.          else
9.              mark_as_boundary(e_(i,j))

10.     if ((σ_i, σ_j, σ_k) = find_seed_triangle())
11.         output_triangle(σ_i, σ_j, σ_k)
12.         insert_edge(e_(i,j), F)
13.         insert_edge(e_(j,k), F)
14.         insert_edge(e_(k,i), F)
15.     else
16.         return
```

图 2: BPA 算法伪代码

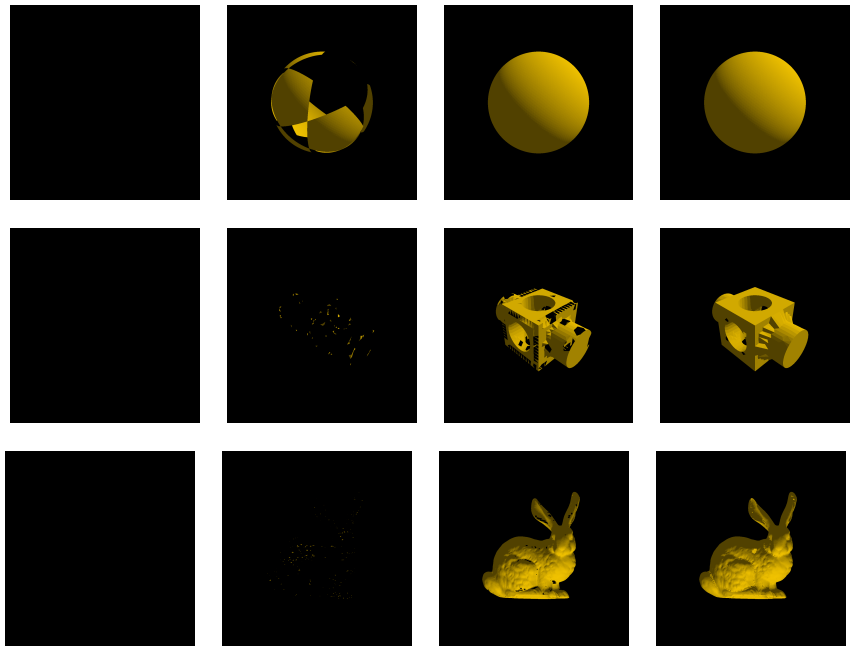

图 3: 从左到右球的半径依次为 0.375ρ，0.75ρ，1.5ρ，3ρ，其中 ρ 为所有顶点到最邻近点的平均距离

3.1 球的半径

首先，球的半径的选择对于算法的效果有很大影响，在此我比较了使用不同半径的球所得出效果。效果如下图3所示。可以看出球的半径很小时，它无法同时与三个点相交，也就无法形成三角形，当球的半径逐渐扩大，三角形的数量逐渐变多，物体的表面逐渐变得光滑。此外，当球的半径过大，我们建立的空间查询数据结构一个格子中的点数就会很多，导致 BPA 算法的效率下降。在助教提供的论文中，球的半径是作者手动选择的。在这个我的实现中，球的半径是根据所有顶点到最邻近点的平均距离 ρ 来选择。平衡重建效果和运行时间，我认为 1.5ρ 是一个比较不错的选择。

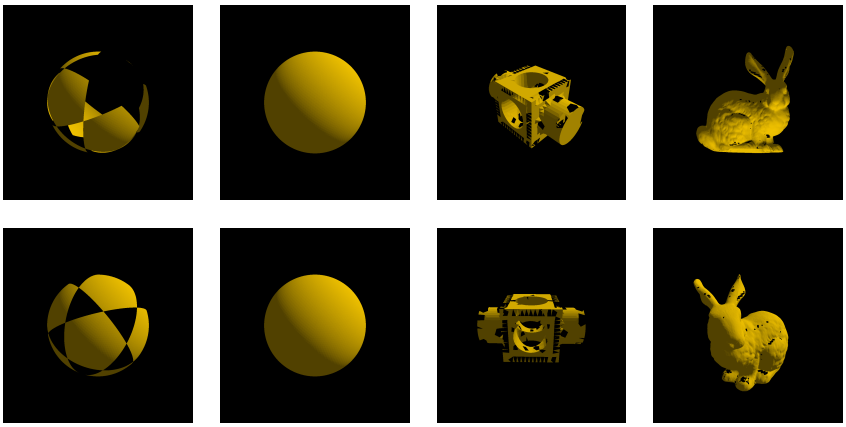

图 4: 双面一排是我实现的 BPA 的效果，下面一排是库函数的效果，两者使用的球的半径相同

3.2 与库函数的比较

为了检验我实现的算法的正确性，我与 Open3D 的库函数进行了比较，结果如图4所示。可以看出，两者实现的效果非常相近。在运行速度上，由于库函数一方面使用了更加高效的空间查询数据结构，另一方面可以并行计算，所以库函数的运行速度明显快于我实现的版本。

4 实现细节说明

在这个大作业中，我主要是按照助教提供的论文中描述的方式进行实现，整个算法的核心流程都没有使用第三方库提供的算法。为了和 Open3D 库中实现的 BPA 算法相比较，我统一使用了 Open3D 库中的点云数据结构，这个数据结构能够根据索引提供对应点的坐标和法向量。我使用的是 lab2 的代码框架，我在原有的基础上新加入了 1 个 task，图形界面的操作和编译的流程与 lab2 完全相同，其中 Radius 的选项可以调节使用的球的半径大小。与 BPA 算法相关代码在目录 $Point-Cloud$ 下，其中，$BPA.cpp$ 为我自己实现的 BPA 算法的核心代码，*utils.cpp* 和 *utils.h* 中为调用 Open3D 库函数的代码和一些为了实现方便而加入的宏定义，*pointcloud.h* 和 *pointcloud.cpp* 提供了和 lab2 框架的接口。由于 Open3D 库和 lab2 之前使用的一些库存在冲突，我无法调用 Open3D 中读取点云文件的函数，所以我直接从 lab2 中原有的三角网格中取出所有点的坐标和法向量作为点云，输入给 BPA 算法。另外，除了 lab2 中原有的素材，我还加入了 *bunny.obj*，*tyra.obj* 这些比较经典的素材，其中 *bunny.obj* 包含 35k 个顶点，*tyra.obj* 包含 100k 个顶点，顶点数都远远多于助教之前提供的素材。