网格简化

陈之杨 计 76 2017011377

2019.6

本次作业实现了基于边坍塌的网格简化算法（*Surface Simplification Using Quadric Error Metrics*）。

1 使用方法

测试环境为 `Windows`。

使用 `g++ -o mesh_simp mesh_simp.cpp -std=c++11 -O3` 编译代码。

使用 `mesh_simp <input model> <output model> <simp ratio> <distance threshold>` 运行代码。其中第三个参数为简化比，第四个参数为算法中设定的距离阈值 t。

2 算法实现

2.1 算法流程

算法流程如论文中所述：

1. 对每个顶点计算 $\boldsymbol{Q}$ 矩阵。
2. 找出所有合法点对（通过边相连或距离小于 t）。
3. 对每个合法点对，计算最小坍塌误差，以及坍塌后的新点位置。
4. 将所有的合法点对以误差为键值存入小根堆。
5. 重复如下操作直至达到简化比要求：找出误差最小的一个点对，合并为一个点，并更新相邻点的 $\boldsymbol{Q}$ 矩阵和对应点对的代价。

2.2 合法点对快速查找

算法时间复杂度的一大瓶颈在于第二步中找出所有距离小于 t 的点对。如果暴力实现，时间复杂度为 $O(|V|^2)$。事实上，如果 t 足够大，那么任意一个点对都是合法点对，那么时间复杂度不可能优于 $\Theta(|V|^2)$。但是考虑到有实际意义的 t 不会很大，每个顶点周围 t 距离内的点数不会很多，因此我们希望设计一个时间复杂度与点对数相关的算法，这在 t 较小时有较大的时间效率优势。

注意到这个问题等价于：给定一个三维点集，列举以每个点为球心，t 为半径中的所有点。那么不难想到使用 KD-Tree 来加速查找。但是，建立一棵 KD-Tree 需要分配大量的内存，时间常数较大，且球内查询的 KD-Tree 并没有明确的时间复杂度保障。注意到本问题中只需要对点集本身的每一个点依次查询一次，将 KD-Tree 显式地建立出来并没有利用到这个问题的特殊性。

我们考虑 KD-Tree 是如何起到加速效果的。KD-Tree 作为一个分治结构，每一轮按照某一个维度将所有点均分为两份，在子树内维护所有点的包围盒以实现剪枝的效果。考虑到网格模型的点一般不会太过密集（总体上较为均匀），因此剪枝能起到较好的效果。

利用这个思想，我们设计一个分治算法解决这个问题。对每一个分治结构，将所有点按照某一维**排序**，并选出中点将点集平分。假设中点的这一维坐标为 x，那么显然只有坐标在 $[x-t, x+t]$ 中的点才可能形成合法的点对，这样的点数是较少的。我们只需要在这个范围内使用平方时间的暴力算法找出所有点对，之后递归处理两个子问题即可。稍微思考一下就可以发现这个算法实质上是把 KD-Tree 的建树和查询操作放在了一起，且没有显式地建出树结构，而是利用 t 较小的性质，进一步优化时间效率。显然，如果点集分布足够均匀，坐标在 $[x-t, x+t]$ 中的点的数量是常数的话，总的时间复杂度应为 $T(n) = 2T(n/2) + O(n \log n) = O(n \log^2 n)$（我们令 $n = |V|$）。

2.3 堆的懒标记维护

我们可以使用 `C++ STL` 中的 `priority_queue` 来实现小根堆。然而，一个难以处理的地方在于，算法中我们需要修改点对的键值，而小根堆只支持插入元素与删除堆顶元素。事实上，可以使用懒标记的思想实现堆的任意元素删除或修改：在我们需要删除一个元素时，我们并不立刻将它从堆中除去，而是给它打上 “删除” 的标记。每次取堆顶元素时，如果遇到被打了标记的元素，则将之删除，直至堆顶元素没有标记为止。修改操作则等价于删除再插入。容易发现这样的实现是合理的，且没有改变时间复杂度。

具体到本算法中，我们使用时间戳来实现懒标记。注意到在任意时刻，每一个点对在堆中只有一个版本是合法的（最后插入的版本）。因此，每当我们修改一个点对时，将其的时间戳加一。当我们取出堆顶元素时，我们只须查验该元素的时间戳与对应点对当前的时间是否吻合即可。由于点对至多有 $O(|V|^2)$ 种，我们可以使用 hash 表或 `STL` 中的 `map/unordered_map` 来维护。

2.4 面片翻转

图 1: 面片翻转：图中的黑色部分并非是空洞，而是一个折入模型内部的面片。

如果直接运行此算法，会出现面片翻转的情况（见图 1）。为了避免这种情况，在计算一个点对的坍缩代价时，须额外判断这种现象是否发生。具体地，对于每一个相邻面片，判断原法向量与坍缩后的法向量的夹角是否大于 180°。若夹角大于 180°（即内积小于 0），则出现了翻转现象，必须对此现象进行惩罚。笔者在测试时最初采用的惩罚方法是每出现一个翻转面片，则代价乘 2。这样翻转面片越多，代价则指数增长。但笔者发现即使这样在简化比较小时仍然难以避免局部翻转现象的发生。故笔者索性将翻转现象发生的点对的代价直接设为无穷大，经测试，算法仍然能正常运行，且有效避免了翻转现象。

2.5 边界保持

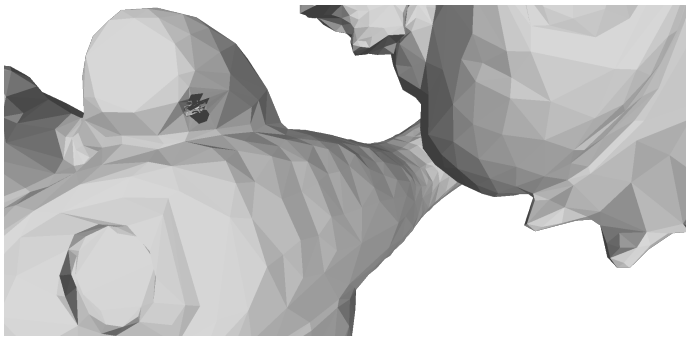

图 2: 边界保持：可以发现模型原有的孔洞形状没有变化，但周围的网格明显被简化了。

有时为了保持模型的拓扑结构，必须维持原有的边界。因此，在选择坍缩点对时，如果其中一个顶点为边界点，我们应当取消之。判断边界的方法是：由于网格模型一般是流形，局部具备欧式空间的性质，因此如果一条边只作为一个面片的边界，我们可以认为这是边界的一部分。

效果图如图 2 所示。

3 实验结果

一些实验结果如下：

模型	原面片数	简化比	距离阈值 t	误差	运行时间
bunny	70580	0.01	0	0.003356	1.516s
bunny	70580	0.01	0.001	0.003356	1.526s
bunny	70580	0.01	0.01	0.003385	14.144s
dragon	209227	0.01	0	0.560887	4.917s
dragon	209227	0.01	0.0001	0.560887	4.907s
dragon	209227	0.01	0.001	0.561980	4.953s
Arma	46398	0.1	0.001	0.248483	0.971s
Arma	46398	0.05	0.001	0.430478	0.999s