

Homework 2: Mesh Simplification

姓名: 方嘉聪 学号: 2200017849

1 项目整体介绍

以 C++ 实现了 增量式网格简化算法 [1] (Incremental Mesh Simplification Algorithm).

1.1 文件结构

项目结构如下:

- `src/`: 源代码, 包括 `main.cpp` 和 `model.hpp` 文件, 其中主要的算法实现在头文件中.
- `result/`: mesh 简化结果, 使用提供的 `simplification.obj` 作为输入, 分别生成了简化比例为 $\{0.9, 0.75, 0.5, 0.25, 0.1, 0.05\}$ 的结果, 以 `simplified_mesh_xxx.obj` 格式存储.
- `CMakeLists.txt`: CMake 配置文件, 用于编译项目.
- `doc/`: 报告 LaTeX 源文件及结果 MeshLab 可视化截图 (`doc/visual_imgs/`).
- `MeshSimplifier`: 编译完成的可执行文件, 使用方法见下.

1.2 编译和运行

第三方依赖库.

1. OpenMesh: 提供半边法数据结构和输入输出接口.
2. Eigen3: 线性代数库, 用于矩阵运算与方程求解.
3. Boost: 主要使用了 `boost/heap/fibonacci_heap.hpp`, 基于此构建了一个 fibonacci 堆, 以实现较为高效的修改元素的优先队列.

编译流程. 在安装好上述依赖库后, 进入项目根目录, 执行以下命令即可编译:

```
mkdir build
cd build && cmake .. && ninja
ninja
```

而后应该能够在 `build` 目录下看到可执行文件 `MeshSimplifier`¹.

运行方法. 使用如下命令运行, 要求 $\text{scalar} \in (0, 1)$.

`MeshSimplifier <dir_to_input_obj> <dir_to_output_obj> <scalar>`

算法的效率没有进行特别优化, 请耐心等待几秒钟:)

2 算法实现细节

2.1 数据结构

主要数据结构如下 (实现在 `model.hpp` 中):

¹这里由于本地一些不可知的 bugs, 我将 Eigen 路径在 `CMakeLists.txt` 中显示指定了, 如有问题需要修改一下.

MeshSimplifier Class 维护了一个 `OpenMesh::TriMesh` 的网格对象, 使用 `OpenMesh` property manager 的方法在 vertex, edge, face 上维护了后续算法所需的属性, 主要包括:

- `face_normal`: 面的单位法向量;
- `face_Q`: 面的 quadric matrix;
- `vertex_Q`: 顶点的 quadric matrix;
- `best_pos`: 每条边 collapse 后, 顶点的最优位置;

EdgeCost 结构体, 包含边及其对应的 cost, 重要用于下面的优先队列操作.

```
struct EdgeCost
{
    double cost;
    TriMesh::EdgeHandle eh;
    bool operator<(const EdgeCost &other) const
        return cost > other.cost;
};
```

Fibonacci Heap 利用 `boost` 库实现了一个 `Fibonacci Heap` 与哈希表, 以支持高效的优先队列操作, 主要用于维护当前所有边的 cost, 以及在每次 collapse 后更新相邻边的 cost. 具体实现如下:

```
using Heap = boost::heap::fibonacci_heap<EdgeCost>;
Heap priority_queue_;
std::unordered_map<TriMesh::EdgeHandle, Heap::handle_type> handles;
// Basic operations:
void push_element(EdgeCost ec);           // Constant complexity.
bool remove_element(TriMesh::EdgeHandle eh); // Logarithmic complexity.
EdgeCost pop_min();                       // Logarithmic (amortized). Linear (worst case).
```

2.2 算法流程

基本流程如下 (参考课程 slides, 原始论文 [1] 及该博客). 给定一个 mesh 输入 M , 对于每一个三角形 F_i , 记顶点为 v_0, v_1, v_2 , 那么该三角形的单位法向量为:

$$\vec{n} = \frac{(v_1 - v_0) \times (v_2 - v_0)}{\|(v_1 - v_0) \times (v_2 - v_0)\|}$$

那么空间中任意一点 v 到 F_i 的距离平方为 (这里省略细节推导)

$$d^2(v, F_i) = h^T Q_i h \text{ where } Q_{4 \times 4} = \begin{pmatrix} \vec{n}^T \vec{n} & \vec{n}^T v_0 \\ v_0^T \vec{n} & v_0^T v_0 \end{pmatrix} \text{ and } h = \begin{pmatrix} v \\ 1 \end{pmatrix}$$

那么对于任意一个三角形 F_i 可以定义出一个二次型:

$$Q_{F_i}(v) = h^T Q_i h$$

而对于一个顶点 v_i , 其 quadric matrix 定义为:

$$Q_{v_i} = \sum_{F_j \in \text{neigh}(v_i)} Q_{F_j}$$

具体算法如下:

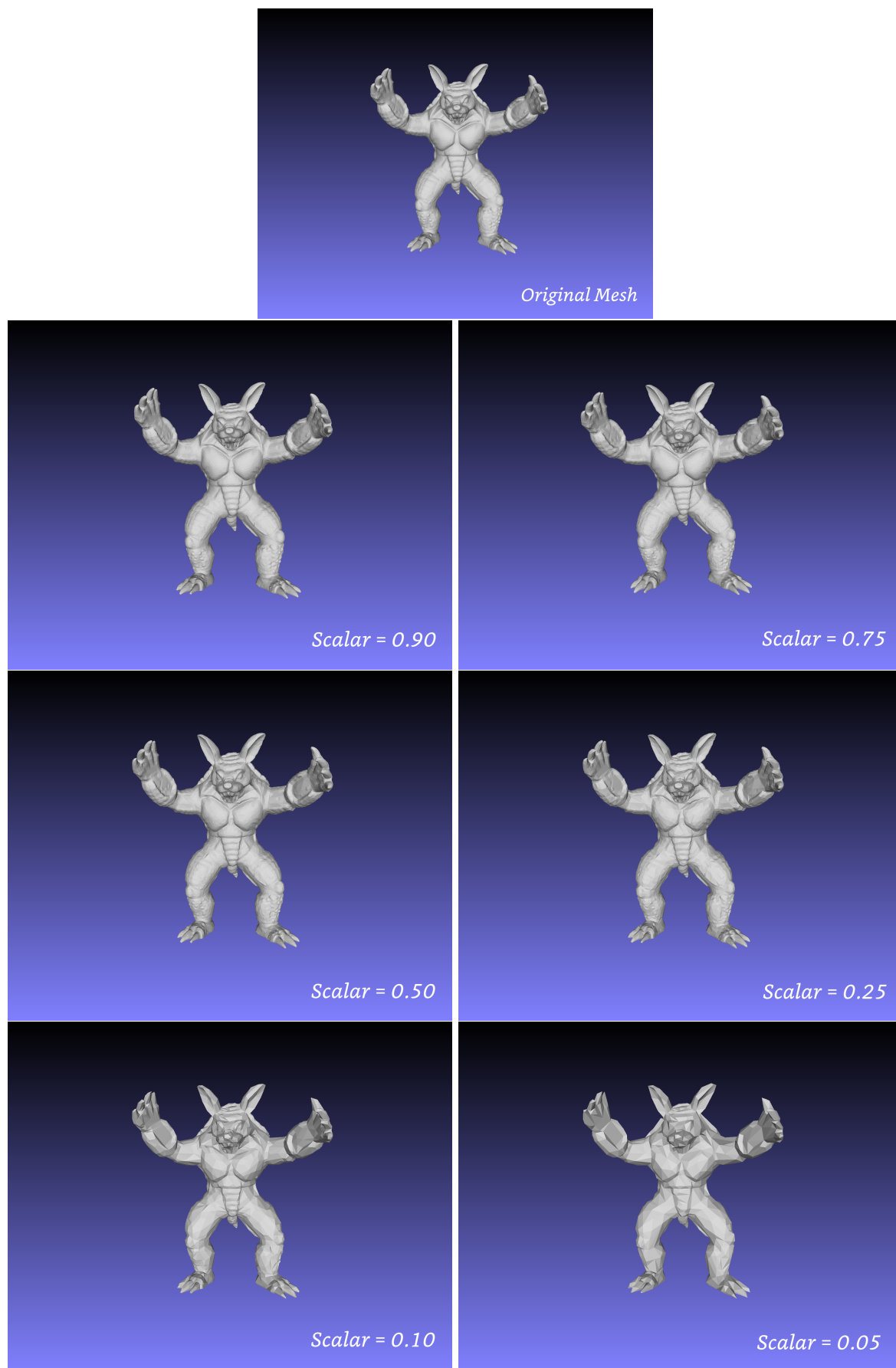


Figure 1: 最上方为原始网格, 其他为简化比例 0.9, 0.75, 0.5, 0.25, 0.1, 0.05 的结果, 放大查看差异.

Algorithm 1 Incremental Mesh Simplification Algorithm**Require:** mesh and simplification scalar $s \in (0, 1)$ **Ensure:** simplified mesh

- 1: 对于每个顶点 v_i , 维护一个 quadric matrix Q_{v_i} .
- 2: 对于每个边 (v_i, v_j) , 维护一个 cost $Q(v') = (Q_{v_i} + Q_{v_j})(v')$. 其中 v' 为最小化 QEM 的位置.
 - $A = n^T n$ 可逆时直接求解.
 - $A = n^T n$ 不可逆时, 取中点和两个端点中 cost 最小的点作为 v' .
- 3: 将所有边 (v_i, v_j) 及其对应的 cost 放入优先队列.
- 4: **while** 目前面数 > 输入 mesh 面数 $\times s$ 且队列非空 **do**
- 5: 从优先队列中取出 cost 最小的边 (v_i, v_j) .
- 6: collapse (v_i, v_j) 到 v' .
- 7: 更新 v' 的 quadric matrix $Q_{v'} = Q_{v_i} + Q_{v_j}$.
- 8: 更新相邻元素的 normal, quadric matrix, cost.
- 9: 将简化后的 mesh 输出到文件.

2.3 具体实现

在这一小节简要介绍一下上述算法的实现细节.

- 在类的构造函数中, 初始化了 `OpenMesh::TriMesh` 对象, 以及 quadric matrix, normal 等属性. 分别实现在 `compute_face_normals()`, `init_vertex_quadrics()`, `build_priority_queue()`.
- `init_vertex_quadrics()`: 利用 `OpenMesh` 的 `Iterator` 遍历所有面, 计算每个面的 quadric matrix 后再计算每个顶点的 quadric matrix. 这里的向量我使用了 `Eigen` 库来表示.
- `build_priority_queue()`: 遍历所有边, 计算 cost 和最优合并点, 并将其放入 `Fibonacci Heap` 中. 最优合并点的计算见下:

```
Eigen::Vector3d solveQuadraticCost(A, b, c, p1, p2){
    Eigen::Vector3d x;
    Eigen::ColPivHouseholderQR<Eigen::Matrix3d> qr(A);
    if (qr.isInvertible()) // A 可逆
        x = qr.solve(-b);
    else
    {
        Eigen::Vector3d x_mid = (p1 + p2) * 0.5;
        // ... 选择 p1, p2 中 cost 最小的点作为 x
    }
    return x;
}
```

- `simplify(size_t target_face_count)`

对外的主要接口, 实现了 mesh 简化的具体流程. 由于 `OpenMesh` 中的 `collapse()` 只会标记要删除的对象, 而不会真正删除, 而调用 `garbage_collection()` 时间成本较高且会重排顶点编号. 因此我维护了一个 `current_face_count` 变量, 每次减去被删除的面数.

`collapse()` 分别遍历两种半边收缩的情况, 选择能够通过 `mesh.is_collapse_ok()` 的一种, 先将 `vh_to` 坐标设置为最优合并点, 然后调用 `collapse()` 函数.

每次成功收缩后, 依次更新受影响的面属性, 受影响的边属性, 更新优先队列 (插入新边, 删

除旧边)。具体代码比较的冗长, 更具体的实现可以参考源代码。在循环结束后, 调用垃圾清理函数, 删除被收缩的元素。

- `main.cpp` 中实现了命令行参数的解析, 读取输入文件, 调用 `MeshSimplifier` 类的接口, 以及输出结果到文件。这里使用了 `OpenMesh` 的 IO 模块,

3 实现效果

在 MeshLab 中可视化的效果见 **Figure 1**。可以看到随着简化比例的降低, 网格的细节逐渐消失, 但整体的几何形状保持较好, 无翻转, 重叠等现象。

参考文献

- [1] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, 1997.