QEM 网格简化算法

Figure 4: A sequence of approximations generated using our algorithm. The original model on the left has 5,804 faces. The approximations to the right have 994, 532, 248, and 64 faces respectively. Note that features such as horns and hooves continue to exist through many simplifications. Only at extremely low levels of detail do they begin to disappear.

网格简化的目的是在不过于损失模型精细度的情况下，尽可能减少使用的三角形个数，对于不同的需要，生成不同细节程度（Level Of Detail， LOD）的模型。

常用的网格简化操作包括

- Vertex Decimation
- Vertex Clustering
- Edge Contraction

然而，仅仅使用Edge Contraction会产生一系列问题，比如本来相邻很近的点会被拉远，此时可能会形成本不该有的空洞。

本文《Surface Simplification Using Quadric Error Metrics》提出的一种基于Pair Contraction的方法，可以很好的收缩本来没有边但是距离较近的点，从而避免这一问题，使用Pair Contraction和Edge Contraction的区别如下：

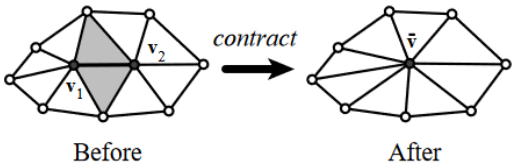


Figure 1: **Edge contraction.** The highlighted edge is contracted into a single point. The shaded triangles become degenerate and are removed during the contraction.

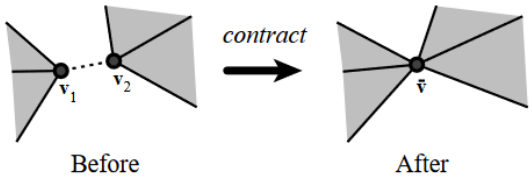


Figure 2: **Non-edge contraction.** When non-edge pairs are contracted, unconnected sections of the model are joined. The dashed line indicates the two vertices being contracted together.

论文中对算法的整体流程描述如下：

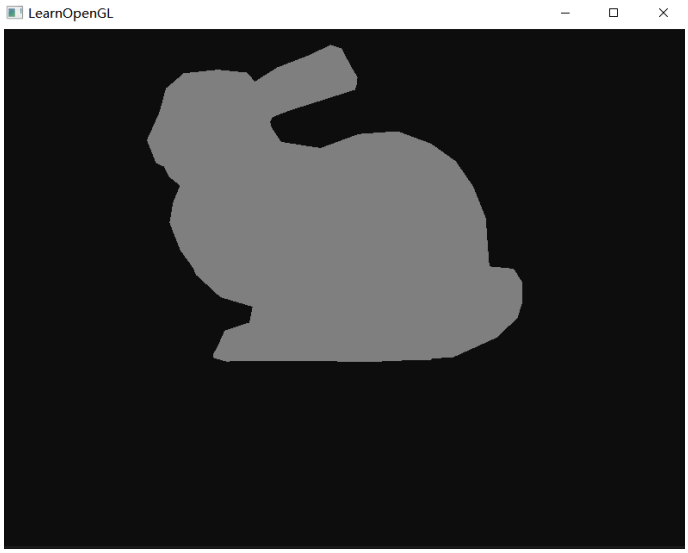
1. Compute the $\mathbf{Q}$ matrices for all the initial vertices.
2. Select all valid pairs.
3. Compute the optimal contraction target $\bar{\mathbf{v}}$ for each valid pair $(\mathbf{v}_1, \mathbf{v}_2)$. The error $\bar{\mathbf{v}}^T(\mathbf{Q}_1 + \mathbf{Q}_2)\bar{\mathbf{v}}$ of this target vertex becomes the *cost* of contracting that pair.
4. Place all the pairs in a heap keyed on cost with the minimum cost pair at the top.
5. Iteratively remove the pair $(\mathbf{v}_1, \mathbf{v}_2)$ of least cost from the heap, contract this pair, and update the costs of all valid pairs involving $\mathbf{v}_1$.

具体实现

在算法的具体实现上，因为需要保存到许多中间变量，数据结构的设计非常关键。

在我的实现中，使用了OpenMesh网格处理工具，使用其***半边数据结构 Half Edge data structure***对原始网格进行处理。

在***OpenGL***显示中，使用Assimp对网格进行处理，因为本文采用的模型没有使用到纹理坐标，因此在渲染的Shader中没有添加纹理映射，得到的图是单一的颜色



通过添加自定义属性来定义每个face的K值，每个Vertex的Q值以及每个顶点所构成的所有合法顶点对（Valid Pairs）。

```
struct MyTraits : public OpenMesh::DefaultTraits
{
    typedef Matrix<float, 4, 4> FaceK;//每个face对应于一个顶点属性
    typedef Matrix<float, 4, 4> VertexQ;//每个vertex对应于一个顶点属性
    typedef vector<VertexHandle> vector_pairs;//每个vertex对应一个vector 保存与当前顶点构成的pairs的vertex
};
```

1. 计算顶点的Q值

对于一次边收缩过程，两个三角顶点v1,v2收缩成一个顶点v，最小化如下的误差度量，即简化后的点与原表面相应局部的距离的平方。

$$\Delta(\mathbf{v}) = \Delta([v_x\ v_y\ v_z\ 1]^T) = \sum_{\mathbf{p}\in \text{planes}(\mathbf{v})} (\mathbf{p}^T\mathbf{v})^2$$

使用二次形式进行表示，其中a,b,c,d为构成平面P的参数，对顶点Q值的计算转化为某个顶点所有1邻域平面的K值求和。

$$\begin{aligned}\Delta(\mathbf{v}) &= \sum_{\mathbf{p}\in\text{planes}(\mathbf{v})} (\mathbf{v}^{\mathsf{T}}\mathbf{p})(\mathbf{p}^{\mathsf{T}}\mathbf{v}) \\ &= \sum_{\mathbf{p}\in\text{planes}(\mathbf{v})} \mathbf{v}^{\mathsf{T}}(\mathbf{p}\mathbf{p}^{\mathsf{T}})\mathbf{v} \\ &= \mathbf{v}^{\mathsf{T}}\left(\sum_{\mathbf{p}\in\text{planes}(\mathbf{v})} \mathbf{K}_{\mathbf{p}}\right)\mathbf{v}\end{aligned}$$

where $\mathbf{K}_{\mathbf{p}}$ is the matrix:

$$\mathbf{K}_{\mathbf{p}} = \mathbf{p}\mathbf{p}^{\mathsf{T}} = \begin{bmatrix} a^2 & ab & ac & ad \\ ab & b^2 & bc & bd \\ ac & bc & c^2 & cd \\ ad & bd & cd & d^2 \end{bmatrix}$$

在OpenMesh中，可以通过句柄访问点、边等属性的所有1邻域信息，

```
    for (MyMesh::VertexIter v_it = mesh.vertices_begin(); v_it !=
mesh.vertices_end(); v_it++)
        //把所有点的1-ring facek面相加
        for (MyMesh::VertexFaceIter vf_it = mesh.vf_begin(*v_it);
vf_it.is_valid(); vf_it++)
```

2.选择valid pairs

有效点对的选择需要满足如下条件之一：

- $ (v1-v2)$ 是一条边
- $||v1-v2||<t$

在本次实现中，使用了Vertex Handle构成的向量来存储每个顶点的所有valid pairs

```
typedef vector<VertexHandle> vector_pairs;//每个vertex对应一个vector 保存与当前顶点
构成的pairs的vertex
```

3.计算每个新顶点对的最优位置，并得到cost，加入优先队列

对于每一对合法顶点对(v1,v2)，通过求解矩阵方程，获取新顶点 $\bar{v} $ 的坐标，其中Q=Q1+Q2

$$\bar{\mathbf{v}} = \begin{bmatrix} q_{11} & q_{12} & q_{13} & q_{14} \\ q_{12} & q_{22} & q_{23} & q_{24} \\ q_{13} & q_{23} & q_{33} & q_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \qquad (1)$$

计算 $\bar{v} (Q1+Q2) v $ 得到每个顶点对的Cost，并将顶点对加入一个优先队列中

存储顶点对的数据结构为：

```
priority_queue<my_set_element,vector<my_set_element>, Cmp> Costs; //优先队列
```

4. 迭代化简

每个迭代过程中，选择堆顶的顶点对(v1,v2)弹出，在网格上进行收缩，更新堆中所有设计v1，v2的pairs，并重新计算

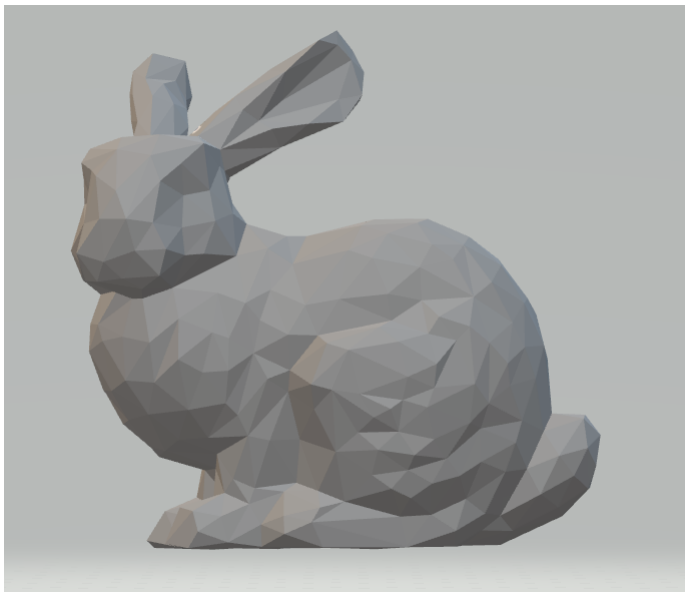
- $ \bar{v} $ 的Q矩阵，为Q1+Q2
- 所有涉及到v1,v2的顶点对，使用$ \bar{v} $ 代替，并修改对应的Costs和新的 $ \bar{v}_{new} $

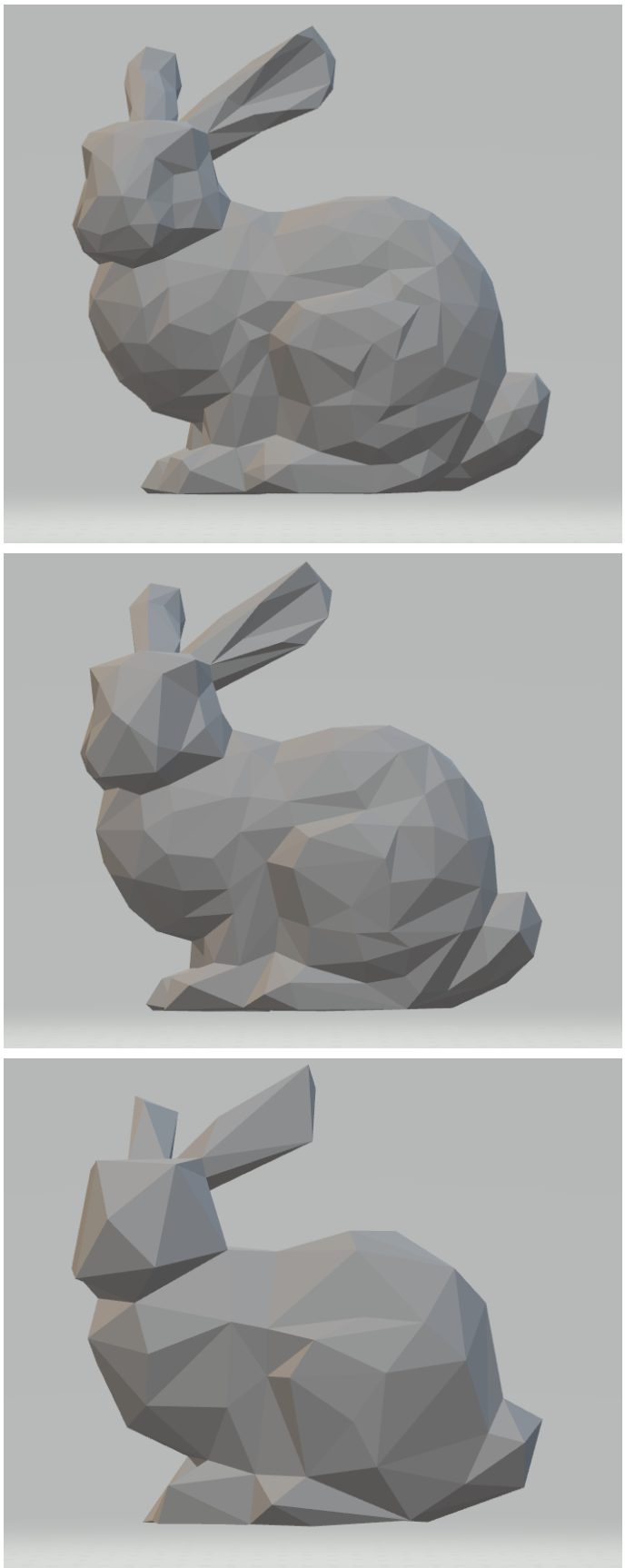
代码如下：

```
void QEM::Simplify()
{
    initQ();
    initPairs();
    calNewVertex();
    while (!Costs.empty() && mesh.n_faces() > simpilification_ratio * all_faces)
    {
        my_set_element temp= Costs.top();
        Costs.pop();
        Costs.pop();
        pairContraction(temp);
    }
}
```

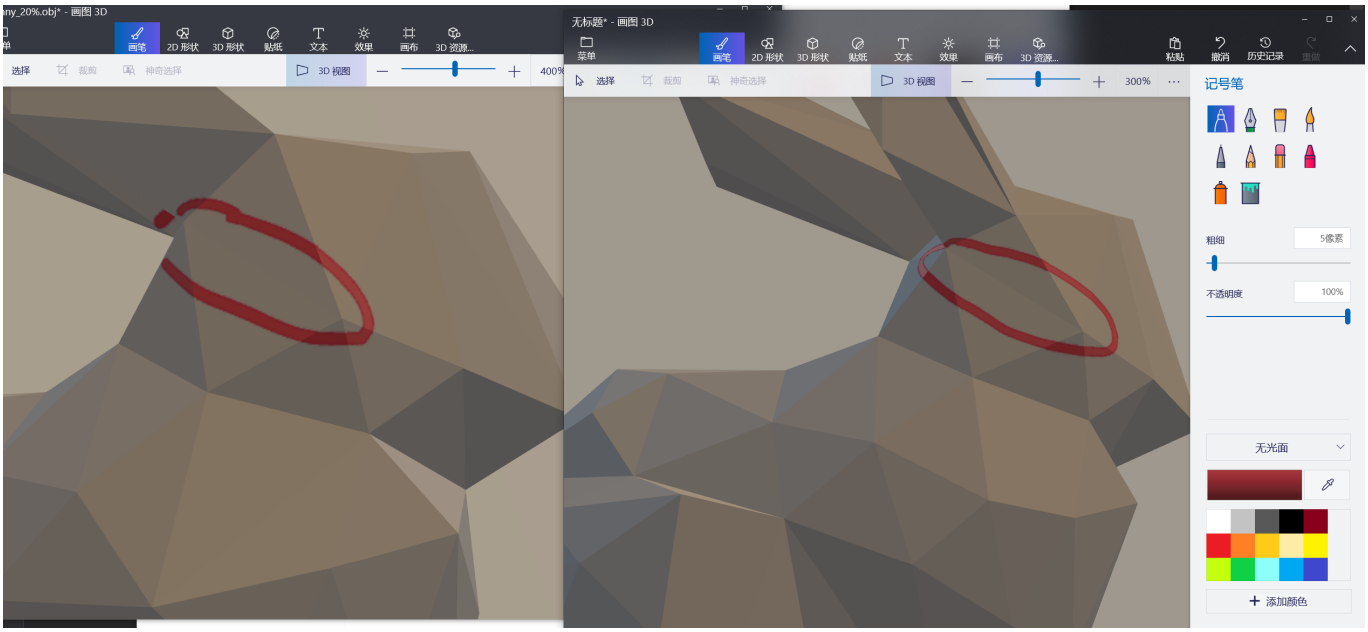
实验结果

使用QEM算法堆bunny.obj进行网格化简，简化比例依次为100%（原图），80%，50%，20%，得到的实验结果如下:



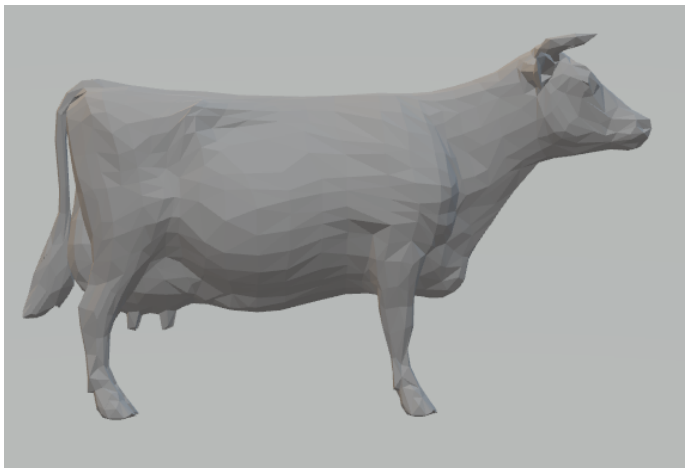


更改参数t，观察简化比例为20%的实验结果，可以发现参数t的设置堆网格的化简结果产生了一定的影响



不足与改进方法

- 本次实现采取了C++内置的priority_queue结构存储每个顶点对的cost，然而这也导致了在每次更新迭代的过程中，涉及到查找堆中所有与v1 v2相关顶点的操作的时候需要遍历堆中所有的元素，导致在涉及到较大网格模型时计算速度很慢，最好是再设计一下更加方便的查找和更新结构。



- 顶点属性typedef vector vector_pairs用于存储所有顶点对，也带来了存储空间的冗余，加大了程序的空间开销。