南 京 师 范 大 学

毕 业 设 计（论 文）

（2022 届）

题　　目：	基于二次度量误差的曲面重构的实现
学　　院：	计算机与电子信息学院/人工智能学院
专　　业：	软件工程
姓　　名：	叶顺龙
学　　号：	19180306
指导教师：	王琼

南京师范大学教务处　　制

摘　要

现有与曲面处理相关的工作中，大多力求在曲面细分和曲面简化领域获得更好的效果或者适应更特殊的的数据，很少有工作关注使用曲面细分和曲面简化算法来更好的重构曲面。但事实上曲面细分和曲面简化工作中的长处可以完成互补，以完成一个全新的目标。

基于这一点，本工作提出了一种基于二次度量误差的曲面重构算法，通过将经典二次度量误差在衡量边坍缩代价上的优势，在曲面细分过程中进行整合，最终达到了使得曲面顶点分布更合理的效果。

为了更好的展现效果，并将提出的曲面重构算法嵌入到整个渲染管线中，本工作设计了完整的重构流程，并在一项开源渲染器上进行了实验。结果表明，该算法具有良好的效果和优异的处理性能，能够在不影响整体渲染流程的条件下，完成高质量的曲面重构。

关键词：几何模型，曲面细分，曲面简化，曲面重构

Abstract

Most of the existing work on surface processing has sought to achieve better results in the area of surface subdivision and surface simplification or to adapt to more specific data, while few work has focused on using surface subdivision and surface simplification algorithms to reconstruct surfaces. However, the fact is that the strengths of surface subdivision and surface simplification work can combine with each other to accomplish a completely new goal.

Based on this observation, a quadratic error-based surface reconstruction algorithm is proposed to integrate the advantages of classical quadratic error in measuring the cost of edge collapse in the process of surface subdivision, ultimately achieving the effect of making a more rational distribution of surface vertices.

In order to better present the results and embed the proposed surface reconstruction algorithm into the entire rendering pipeline, a complete reconstruction process is designed and experimented on an open source renderer. The results show that the algorithm provides good results and excellent processing performance, and is able to perform high-quality surface reconstruction without slowing down the overall rendering progress.

Keywords: geometric mesh, mesh subdivision, mesh simplification, mesh reconstruction

目　　录

第 1 章 绪论

这一章主要描述本工作的研究目的及意义，同时结合对国内外研究现状的观察，确定本工作的研究内容。

1.1 本课题的研究目的及意义

计算机图形学中的几何部分，一直以来都是一个值得探讨研究的工作。因为它并不是一个遥远抽象的问题，而是自我们一出生起，就出现在了我们的生活之中。

我们平常看到的玻璃杯，有着各种曲线的轮廓，让人思考如何去生成这样的曲线；看到的轿车，有着各种光滑的曲面，让人思考如何去定义这样的曲面；看到的发动机，有着各种精密连接的零部件，让人思考如何在几何体之间进行拼接；看到的丝绸布料，有着独特的质感和透明度，让人思考如何构造合适的几何体还原这些细节；看到的水滴，有着不规则的形态，让人思考如何在几何体中模拟这样的流体张力；看到的城市鸟瞰图，有着复杂的结构和表面，让人思考如何构建、存储、渲染如此复杂的几何模型；看到的显微镜下的生物，有着微观的表征形态，让人思考如何这些细节在宏观场景和微观场景中是如何切换的等等。

在计算机图形学中，将几何划归为了两类，第一类名为隐式（Implicit）几何，第二类名为显式（Explicit）几何。隐式几何不直接给出模型顶点位置等要素信息，而是给出模型要素信息满足的关系，例如使用一个球面通过方程$x^2 + y^2 + z^2 = 1$定义；显示几何则会直接给出要素信息。隐式几何能够很方便的判断诸如“给定顶点是否位于指定平面上”等问题，但对于处理诸如“哪些顶点位于指定平面上”等问题时则会显得捉襟见肘。

隐式几何中的常见表示方法包括代数表示法（Algebraic）、集合布尔运算表示法（Boolean）、距离函数表示法（Distance Function）、水平集表示法（Level Set）、分形表示法（Fractals）等。这些表示法根据其表示方式有着不同的应用，但与本工作关系较小，这里不进行详述。

显示几何中常见的表示方法包括点云表示法（Point Cloud）、多边形面表示法（Polygen Mesh）等。其中点云表示法通常来自于三维扫描结果，并会在三维扫描后，

被转换为多边形表示法对应的曲面。因此本工作将重心放在了显示几何中的多边形面表示法上，并选择多边形面表示法中最典型的三角面片表示法进行研究。

在完成了对模型曲面的表示后，对于曲面上顶点坐标、纹理、法向量等要素信息的分布便成了一个值得研究的问题。无论是将顶点变得更稠密，使得它们能够表示更细节的信息；还是将顶点变得更稀疏，使得它们占用更小的渲染负荷；抑或是将顶点重新布局，使得它们能够组成更合理的结构，都在从不同方面契合着人们对于曲面处理操作的需求。

因此，针对“将顶点重新布局，使得它们能够组成更合理的结构”这一需求，本工作提出了一种基于二次度量误差的曲面重构算法。从经典的曲面细分和曲面简化算法中汲取长处，并将它们创造性的结合，最终达到新的目的。

1.2 国内外研究现状

目前，对于曲面相关算法的研究包括曲面细分、曲面简化和曲面正则化。其中曲面正则化用于让曲面中的“每个三角形更像正三角形”，因此与本工作关联很小，这里不再展开。这里将主要介绍曲面细分研究和曲面简化研究。

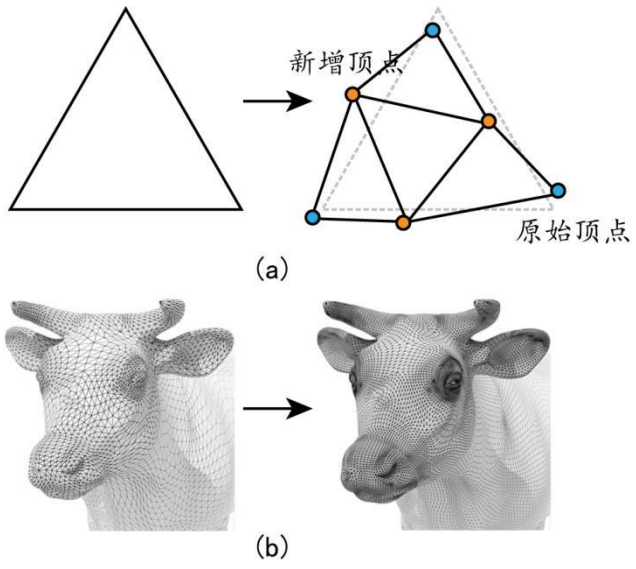

图 1-1 曲面细分算法。（a）曲面细分过程拓扑图；（b）曲面细分过程效果图。

如图 1-1 所示，曲面细分算法的核心在于使得曲面更加平滑，并可以基于此实现 LOD（Levels of Details）算法，即通过物体覆盖屏幕的大小来确定它的绘制粒度。计算机无法直接渲染生成曲线，自然更无法直接渲染生成曲面。我们在计算机中所看到

的曲线曲面，实际上是由若干个极小的线段和多边形组成。随着线段和多边形的数量越多，粒度越小，曲线和曲面就会显得越加真实。曲面细分算法中经典的基于三角面片的曲面细分算法 Loop 细分过程如图 1-1（a）所示，其基本思想为：把一个三角面片分成四个三角面片，并根据新增顶点和原始顶点分别进行坐标的调整。它的细分效果如图 1-1（b）所示。

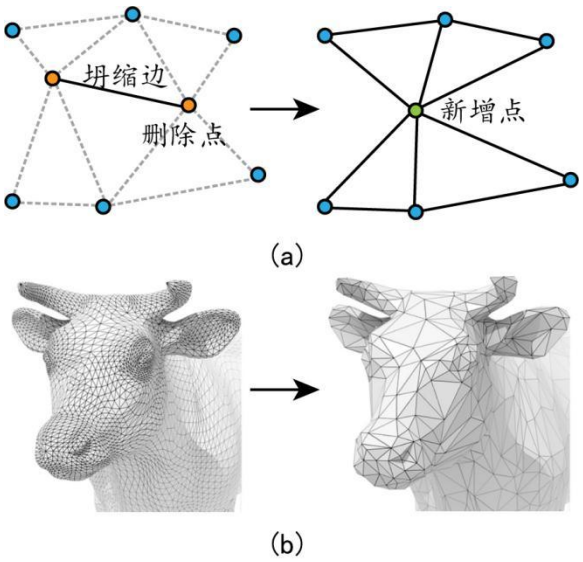

图 1-2 曲面简化算法。（a）曲面简化过程拓扑图；（b）曲面简化过程效果图。

如图 1-2 所示，曲面简化算法的核心在于使得曲面在保留基本轮廓的前提下，使用尽量少的顶点，从而降低模型精度，同样也是 LOD 技术的组成部分之一。曲面简化算法中经典的基于边坍缩的曲面简化算法简化过程如图 1-2（a）所示，其基本思想为：将一条边的两个顶点合并为一个顶点，在这其中需要有效的衡量确定坍缩哪一条边可以对曲面形状带来的影响最小。它的简化效果如图 1-2（b）所示。

1.3 研究内容

本工作在经典曲面简化和曲面细分算法上做出拓展，而不再受限于原始算法所面对的时代困境。通过融合这两者的优势，以应对更新的曲面设计目标。

更具体地描述，即原始的曲面简化和曲面细分算法，分别专注于计算机性能和视觉效果。而基于这两者融合产生的新的曲面重构算法会在尽可能考虑计算机性能的前提下，达到更好的视觉效果。虽然这样的描述可能与 Xia 等人提出的动态且视角独立

的曲面简化算法[12]有相似之处，但需要强调的是，本工作最终设计完成的算法，与该算法有如下几点主要区别。

（一）本工作实现的算法在曲面简化的基础上，通过结合经典的曲面细分算法获得了更合理的曲面顶点分布。而 Xia 等人提出的算法并没有考虑曲面细分问题，即最终获得的效果必然带来视觉效果上的滑落，而本工作会使得视觉效果变好。

（二）本工作实现的算法针对整个曲面，而非视角导向。因此在完成模型计算后，本工作的算法时空复杂度是远远优益于 Xia 等人提出的算法的。这一优势可以从后续章节的渲染结果数据中看出。

综上所述，本工作虽然借鉴了较多国内外研究工作，但并不意味着本工作与其中某个工作有着相似的思路。它更可以被描述为是：为完成曲面相关算法的一个共同远大目标，而做出的另辟蹊径的尝试。

1.4 本章小结

本章主要介绍了本工作提出的曲面重构算法的研究场景，包括对它的研究目的和意义的分析，以及国内外研究现状的思考。从而最终得出确切的研究内容。

第 2 章 相关工作

这一章主要描述与本工作相关的研究，阐明研究的问题，研究的对象以及研究的技术，从而构建工作之间的内在逻辑联系并突出本工作的不同之处。

2.1 工作分类

常见的曲面操作可以分为三种，曲面细分、曲面简化和曲面正则化。其中让“每个三角形更像正三角形”的曲面正则化与本工作相关度很小，可忽略不计。因此这里对相关工作的分类主要涉及曲面细分和曲面简化。除此以外，部分工作中对曲面的操作不止曲面细分或者曲面简化中的一项，它们还会对曲面进行其他操作，例如在细分前寻找满足细分连通性的最大独立顶点集[5]、曲面视角模型构建[12]、根据高度图进行曲面重建[15]等。如表 2-1 所示，按照这个标准分类的相关工作会有所重叠。

表 2-1 参考文献分类

	曲面细分	曲面简化	曲面重构
Sun et al.[1]	√		
Zhou et al.[2]	√		
Arden et al.[3]	√		
Prautzsch et al.[4]	√		
Chen et al.[5]	√		√
Hu et al.[6]	√		
Loop et al.[7]	√		
Yao et al.[8]		√	
Garland et al.[9]		√	
Garland et al.[10]		√	
Cohen et al.[11]		√	
Xia et al.[12]		√	√
Kalvin et al.[13]		√	

Klein et al.[14]		√	
Ban et al.[15]		√	√

2.1.1 曲面细分

有关曲面细分操作的相关工作，可以通过三个角度进行梳理。第一个角度在于对细分基础理论的设计研究，第二个角度在于细分算法本身的优化，第三个角度在于对细分算法的场景应用。

基础理论设计研究的工作，例如经典的 Loop 细分[7]的原理设计和对它在理论基础和实现细节（边界、约束等条件）的展开[3]。细分算法本身优化的工作，例如在细分过程中引入渐进插值方法[1]、细分计算时区域划分与精准边界的重新确定[2]、细分算法与蝴蝶算法的结合[4]。细分算法的场景应用工作，例如曲面重新采样[5]后或者人体数据获取[6]后利用细分平滑曲面。

2.1.2 曲面简化

有关曲面简化操作的相关工作，同样可以通过三个角度进行梳理。第一个角度在于对简化基础理论的设计研究，第二个角度在于简化算法本身的优化，第三个角度在于对简化算法的场景应用。

基础理论设计研究工作，例如经典的二次度量误差算法[10]、映射函数生成算法[11]、基于原始图节点真子集的边界函数算法[13]、基于豪斯道夫距离的三角面片简化算法[14]。简化算法本身优化的工作，例如使用离散曲率保留更多特征的方式优化二次度量误差算法[8]、将二次度量误差算法拓展到能够计算曲面的其他属性[9]（纹理坐标、法向量）。简化算法的场景应用工作，例如基于动态视角变换的在线曲面简化算法[12]、场景重建后的曲面简化[15]。

2.1.3 曲面重构

有关曲面重构操作的相关工作，已经涵盖于前两小节中，故这里不再赘述。

2.2 本工作与曲面相关研究的关系

本工作同时使用了曲面细分与曲面简化研究的经典模型。除此以外，还对模型做了更多细节设计，即曲面重构。本工作与曲面研究的关系可以归纳为以下几点。

（一）本工作在曲面细分模块采用了经典的基于三角面片的曲面细分算法 Loop 细分[7]，在曲面简化模块采用了经典的基于二次度量误差的曲面简化算法[10]，并将二次度量误差引入到细分过程，作为核心的细分依据。

（二）本工作对原始曲面细分和曲面简化的实现进行了改进，以更好的适应本工作对于重构曲面的需求。

（三）本工作不限于朴素的曲面细分与曲面简化对顶点的改变，同样考虑了纹理坐标、法向量的曲面属性在重构过程中的计算。并将本工作模块嵌入到一个完整的渲染管线中。

2.3 本章小结

本章主要介绍了本工作提出的曲面重构算法的相关工作，首先阐述了曲面相关研究的分类，其次基于该分类，详细的阐明了其中三个研究部分：曲面细分、曲面简化和曲面重构的研究内容与内在联系。从而在此基础上，对比分析本工作与曲面细分、曲面简化和曲面重构研究之间的关系。

第 3 章 曲面重构算法的设计

这一章主要描述了本工作提出的曲面重构算法的设计目标、设计考虑和设计细节。设计细节包括对曲面简化和曲面细分特性的归纳总结和完整曲面重构流程设计。最后，为了更好的嵌入渲染流程，在最后一节同样加入了渲染管线的叙述。

3.1 设计目标

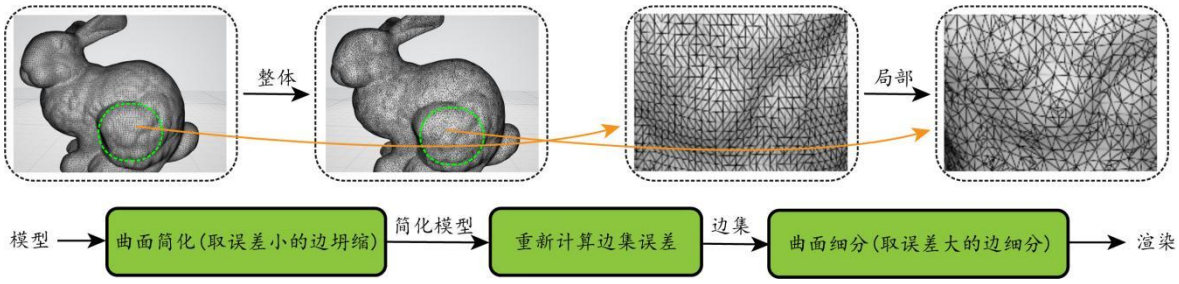

图 3-1 曲面重构过程的核心思路。通过迭代的曲面简化和曲面细分，使得整个曲面三角面片布局趋于更加合理的状态。

本工作提出的曲面重构算法，旨在通过结合曲面简化和曲面细分的特性，来完成三角面片的布局重新分布，而这样针对三角面片的重新分布，即为曲面重构。

如图 3-1 所示，经典的斯坦福兔子模型在这样的过程中，原本均匀的三角面片布局会在平坦处变得稀疏，而在崎岖处变得稠密。即在不改变曲面整体三角面片数量级，不增大曲面渲染负荷的情况下，使得稀疏处仍能够得到合理描述，而稠密处的细节会更加明显。

3.2 简化与细分关系

在研究背景中，一种经典的算法[9]被提出用于进行曲面简化。而这种算法有着一个重要的衡量指标，用于评价一条边坍缩后产生的代价，即二次度量误差。

在具体实现部分，会对二次度量误差的计算做详细阐述。这里可仅简单的将二次度量误差理解为对一条边坍缩后代价的评价。如果将曲面对模型的刻画细节程度作为评价指标。那么曲面简化会对这个过程产生负面影响，而曲面细分则会对这个过程产生正面影响。二次度量误差大，则该边坍缩后对于整体曲面的负面影响更大，而二次

度量误差小，则该边坍缩后对于整体曲面的负面影响更小。因此在经典的基于二次度量误差的曲面简化算法中，会优先选择二次度量误差较小的边进行坍缩。

在分析完经典曲面简化算法的特征后，与之相对应的曲面细分算法同样值得研究。

根据研究背景中的叙述，每一次的曲面细分过程，都会在每条边上新增一个顶点（新顶点不一定会出现在边上，但出现的数量和一定等同原先曲面上三角面片的总边数）。依据该特征，可以获得一个显然的结论，即每一次曲面细分后，整体的三角面片数量会近似以 4 为指数进行递增，且在此过程中，不会出现针对性的细分。

于是，曲面简化和曲面细分的特征可以被概括如下。

（一）曲面简化使用二次度量误差作为边坍缩代价的衡量指标，旨在根据坍缩影响大小依次坍缩对曲面上的边。但随着简化的过程演进，曲面会逐渐失去细节与特征。

（二）曲面细分会无差别的对每条边进行细分，产生指数级的时空消耗，对渲染过程产生压力的同时，产生了大量的冗余细分。

因此，在分析完两个曲面相关的核心问题与经典算法后，本工作有了最初的思考：曲面简化旨在降低渲染消耗，因此不断地坍缩边导致曲面细节丧失。而曲面细分旨在增添曲面细节，因此大量无差别细分边导致曲面膨胀，容易节点冗余。那么，能否结合两者的优势，来做到让曲面的三角面片分布更为合理呢？

在理论分析和实践后，这个过程被证明是可行的。经典曲面简化算法的优势在于通过二次度量误差（也有其他经典曲面简化算法并没有使用二次度量误差，但同样使用了其变体，或者参考了其衡量边坍缩代价的思路）细致的量化了一条边坍缩后的代价。

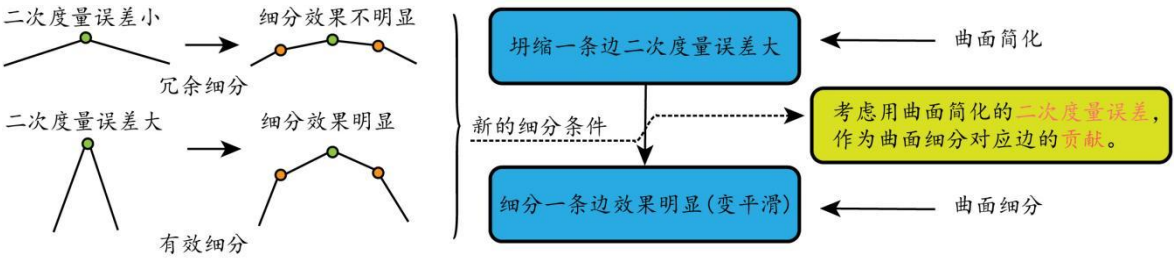

图 3-2 细分和简化的关系。通过细致观察曲面简化的过程，从而归纳出其衡量指标的具体意义。从而试图迁移该特征，以用于曲面细分。

而这个衡量指标本身事实上并不限于曲面简化问题。如图 3-2 所示，二次度量误差可以被用于作为细分的标准。二次度量误差小的边若进行细分，则能带来的细节变化同样较小，原本平坦的表面会变得稍微平坦一点，即细分效果不明显。而二次度量误差大的边若进行细分，则能带来的细节变化同样较大，原本崎岖的表面会变得相对光滑，即细分效果明显。换句话说，可以将经典曲面简化算法中用到的二次度量误差这个衡量指标，作为无差别曲面细分过程中选择细分边的贡献。贡献与最终曲面上三角面片的分布合理程度正相关。

3.3 重构流程

在完成了曲面简化与曲面细分算法的讨论，进行讨论得出思路后，本工作提出了一个完整可行的重构流程。该重构流程细致地描述了整个模型的三角面片是如何通过曲面简化和曲面细分变得更合理的。

如图 3-3 所示，一个完整的重构流程主要分为三个部分：曲面简化，重新计算边集误差，曲面细分。

在曲面简化过程中，二次度量误差较小的边会被坍缩。其过程包括如下。

（一）读取模型参数

（二）计算模型点线面数量

（三）申请设计数据结构空间

（四）根据每个三角面片逆时针次序建有向图

（五）区分无向图中边界点和内部点

（六）邻接表转存无向图

（七）计算每个顶点相邻三角片面所在平面法向量

（八）计算顶点代价矩阵

（九）建立顶点和纹理、法向量映射

（十）计算边集所有边坍缩代价和新增点最优位置

（十一）根据最优位置插值新增点纹理和法向量

（十二）将每条边代价组合后插入自定义小根堆

（十三）初始化并查集

（十四）计算需要坍缩的边数

（十五）循环从堆中取出代价最小的边

（十六）查询并查集，防止边重复坍缩

（十七）检测连通性约束和几何约束，防止出现模型孔洞

（十八）坍缩边并更新无向图结构

（十九）计算新边的坍缩代价和新增点最优位置

（二十）修改并查集

（二十一）离散化点集

（二十二）释放数据结构空间

重新计算边集误差过程包括如下。

（一）重新计算边集坍缩代价

（二）计算需要细分的边数

（三）新建小根堆或许需要细分的边集

在曲面细分过程中，二次度量误差较大的边会被细分。其过程包括如下。

（一）模型前期处理同曲面简化

（二）根据每个三角面片逆时针次序建有向图

（三）计算顶点度数，区分 regular_point

（四）区分有向图中的边界点和内部点

（五）建立顶点和纹理、法向量映射

（六）计算新增顶点位置

（七）更新原始顶点位置

（八）根据细分边集在三角面片中的分布，重构曲面

（九）释放数据结构并写入模型文件

上述过程中虽然已经提出了基本的处理流程，但仍有大量的实现细节未展开，这些内容将会在具体实现章节中详细阐述。

值得一提的是，可以注意到，这样的一个重构流程后，整个模型的三角面片会在一定程度上趋于合理。但这样的重构流程并不只是单次的，该过程可以依据分布的要求进行迭代往复执行。直至达到用户理想的分布状态。

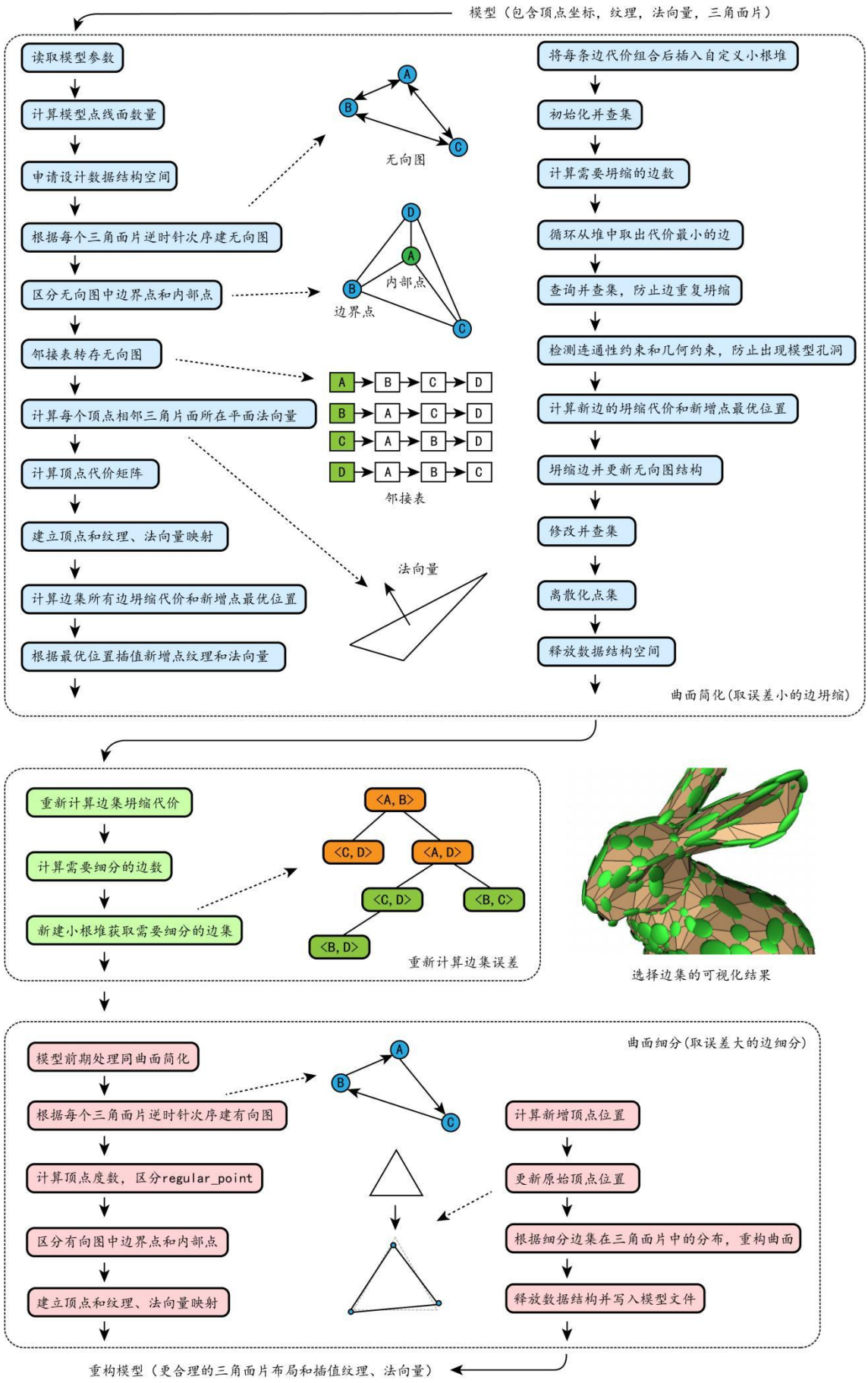
模型（包含顶点坐标，纹理，法向量，三角面片）
读取模型参数
计算模型点线面数量
申请设计数据结构空间
根据每个三角面片逆时针次序建无向图
区分无向图中边界点和内部点
邻接表转存无向图
计算每个顶点相邻三角片面所在平面法向量
计算顶点代价矩阵
建立顶点和纹理、法向量映射
计算边集所有边坍缩代价和新增点最优位置
根据最优位置插值新增点纹理和法向量
无向图
内部点
边界点
邻接表
法向量
将每条边代价组合后插入自定义小根堆
初始化并查集
计算需要坍缩的边数
循环从堆中取出代价最小的边
查询并查集，防止边重复坍缩
检测连通性约束和几何约束，防止出现模型孔洞
计算新边的坍缩代价和新增点最优位置
坍缩边并更新无向图结构
修改并查集
离散化点集
释放数据结构空间
曲面简化(取误差小的边坍缩)
重新计算边集坍缩代价
计算需要细分的边数
新建小根堆获取需要细分的边集
重新计算边集误差
选择边集的可视化结果
曲面细分(取误差大的边细分)
模型前期处理同曲面简化
根据每个三角面片逆时针次序建有向图
计算顶点度数，区分regular_point
区分有向图中边界点和内部点
建立顶点和纹理、法向量映射
计算新增顶点位置
更新原始顶点位置
根据细分边集在三角面片中的分布，重构曲面
释放数据结构并写入模型文件
重构模型（更合理的三角面片布局和插值纹理、法向量）

图 3-3 重构流程。主要分为三个部分：曲面简化，重新计算边集误差，曲面细分。

3.4 完整渲染流程

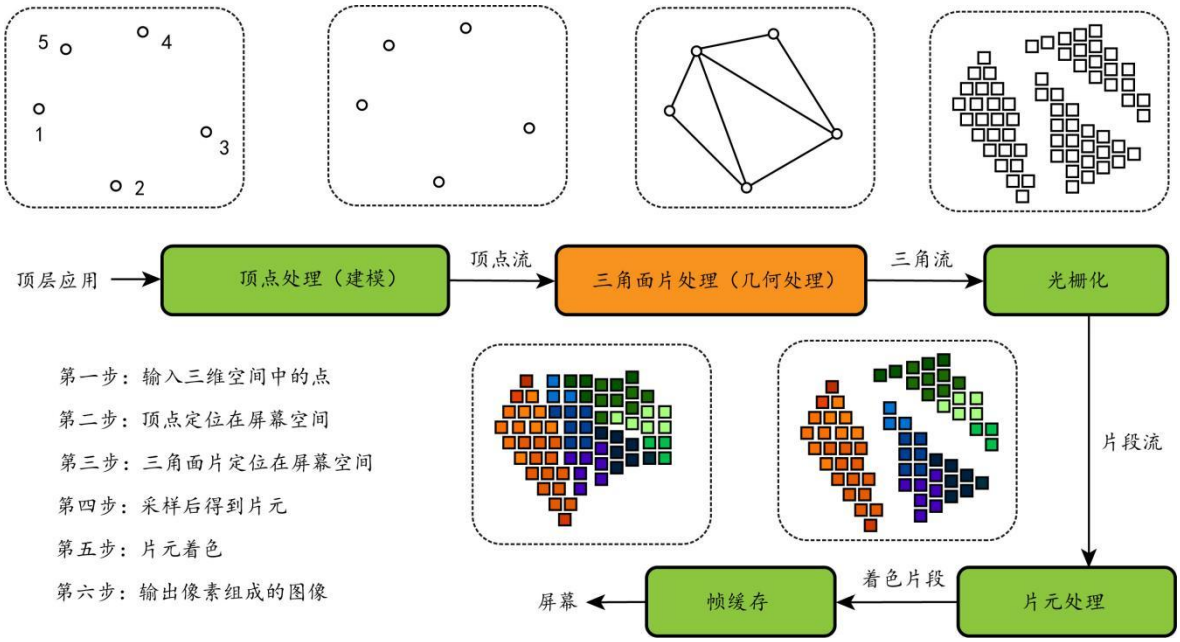

图 3-4 完整渲染流程。曲面重构流程被包含在橙色标识的三角面片处理（几何处理）中。

图形管线（Graphics Pipeline），又名实时渲染管线（Real Time Rendering Pipeline）。它的基本结构如图 3-4 所示，描述了从场景到最后的图像经历的完整过程。

流程的输入是空间中的点集，经过顶点处理（建模）后，点集会在诸多投影变换后投影到（屏幕）平面上。紧接着，这些顶点会形成三角面片。此时三角面片上的所有点属于连续状态，而需要绘制点的屏幕属于离散状态。因此，下一步通过光栅化，三角面片会被离散为像素（像素的原始说法为 OpenGL 中的 Fragment，在不使用抗锯齿（MSAA，Multi-Sampling Anti-Aliasing）等操作时，可认为一个 Fragment 对应一个像素）。当所有三角面片在屏幕上被打散为不同的像素后，即可进行着色，即获得原本三角面片上的点对应到屏幕上应该显示的颜色。最后通过帧缓存，获得最初三维场景对应的完整图像。

在此流程中，渲染器编写中涉及到的其他算法分别出现部分为：模型（Model）、视口（View）、投影（Projection）变换出现在顶点处理部分；像素采样出现在光栅

化部分；深度缓存（Depth Buffer）出现在片元处理部分；着色（Shading）、纹理映射在顶点处理和片元处理上（具体出现在哪个部分，取决于着色方式）。

由于该部分并不是工作的重点，其多数特性也都来自于开源渲染器代码和 OpenGL 官方文档。故这里的叙述略去了其中大量的基础理论和设计细节。包括但不限于着色器（Shaders）编写、齐次裁剪（Homogeneous clipping），透视插值（Perspective correct interpolation）、背面剔除、深度测试、透明混合、天空盒、骨骼动画、切线空间法线贴图（Tangent space normal mapping）、阴影贴图（Shadow mapping）、ACES tone mapping 算法、经典的 Blinn‐Phong 反射模型、时髦的 Physically based rendering（PBR）、用于环境光照的 Image-based lighting（IBL）、类似于 Sketchfab Viewer 的环绕式相机、类似于 Marmoset Viewer 的材质查看器。

3.5 本章小结

本章主要介绍了本工作提出的曲面重构算法的设计，首先阐述了设计目标，其次基于该目标详细分析了曲面简化和曲面细分的特征，获得大致思路。从而在此基础上，完成了整个重构算法的设计。最后一节加入的渲染管线的叙述也更好的让该算法嵌入渲染流程。

第 4 章 曲面重构算法的具体实现

这一章主要描述了本工作提出的曲面重构相关算法的具体实现。具体实现内容包括表面模型的设计与解析，数据结构设计，朴素曲面简化实现，朴素曲面细分实现，重构过程中的曲面细分，纹理与法向量插值与实现细节。

4.1 表面模型

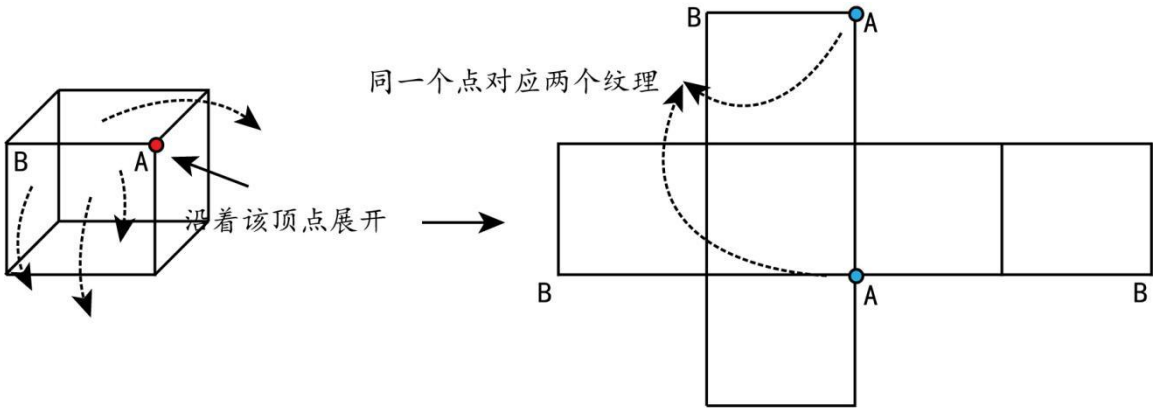

图 4-1 顶点与纹理映射。（a）最基本的三维模型：一个立方体；（b）将该立方体沿着图示箭头进行展开后获得的展开图。可以注意到同一个顶点会在展开图中被映射到多个二位坐标，即会对应多个纹理。

与 2D 模型不同，本工作针对的 3D 模型是具有体积的。然而，建立一个具有内部结构的 3D 模型并不现实。原因可以通过观察人类获取 3D 模型的途径来分析。

常见的创建 3D 模型的途径有两种：扫描仪三维重建模型和模型设计师使用类似 3DMax、Maya、ZBrush 等软件手工创建。对于前者建立的模型，本身不具备任何切割的可能性，因为扫描仪并没法重建物体的内部结构。对于后者建立的模型，虽然它在理论上具有内部结构，但同样不支持任意切割，因为切割后新增的无限可能曲面无法与现有的有限可能贴图纹理或者法向量进行匹配。

因此，考虑到屏幕在同一时刻永远只能展现模型的某个表面。现有的 3D 模型创建与设计大都遵循了只创建表面的逻辑，也即这里提到的表面模型。即使存在某个模

型有被切割的需要，也只会将这样的一个模型分割为若干个模型创建，以支持固定可能性的切割变化。例如如果需要创建一个可以被引爆的炸药桶，就会对创建若干个爆炸碎片模型和爆炸后的纹理、法向量贴图等，最后使用这些模型碎片，无缝拼接成最终展现在屏幕上的模型。

如图 4-1 所示，表面模型对外展现出的纹理由顶点与纹理映射决定。值得一提的是，由于纹理贴图仅仅拥有两个坐标（UV 坐标），因此映射到 3D 模型的展开结果上后，就会存在若干个纹理坐标对应一个顶点的情况。模型中顶点和法向量映射遵循同样的逻辑，这里不再赘述。

需要注意的是，为了使用更通俗的语言描述设计的核心内容。本文不对“模型”、“表面模型”、“3D 模型”、“曲面”这几个概念进行区分。读者可以将它们统一理解为“具有体积的、没有内部结构的、由三角面片构成的模型”。

4.2 数据结构设计

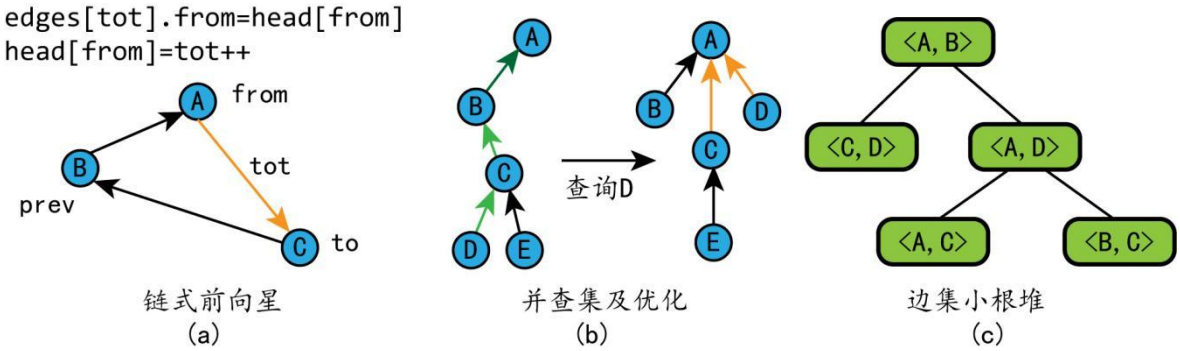

图 4-2 数据结构设计。（a）使用链式前向星存储的图结构；（b）使用并查集维护的顶点更新情况；（c）使用小根堆维护的坍缩边集二次度量误差情况。

如图 4-2 所示，这里列举了实现过程中最主要的三个数据结构：链式前向星，带优化并查集及小根堆。下面详细介绍三个数据结构的设计与实现。

4.2.1 链式前向星

在提及链式前向星之前，需要先引入它的祖先：前向星。前向星是一个特殊的边集数组，通过将边集中的每条边按照起点排序（起点相同时按照终点排序），来维护每个点作为起点的所有边在边集数组中的起始位置和存储长度。而链式前向星的结构

如图 4-2（a）所示，它可以被理解为是一个基于数组实现的头插法的静态邻接链表，是对于前向星的改进，避免了前向星在遍历图结构时，需要以某个点为起点或者终点对所有边排序的操作。

使用链式前向星对边进行操作的示例代码如图 4-3 所示，它支持的操作包括动态增添或删除边，以及遍历某个节点的所有出边（数据结构空间的分配与释放由 malloc 和 free 操作完成）。值得一提的是，原始的链式前向星是不支持动态删边的。本工作在实现过程中，结合三角形逆时针顺序存图和曲面简化在删边过程中的特点，实现了一个简化版的支持动态删边的链式前向星。

```
//链式前向星加边：边起点from，终点to，边集数组编号tot
void add(int from,int to,int prev){
    edges[++tot].from=head[from],head[from]=tot,edges[tot].to=to;
    //链式前向星加边考虑三角形逆时针次序，三角形另一个顶点prev
    edges[tot].prev=prev;
}
//链式前向星删边后加边：删除边起点from，终点to，边集数组编号tot，新增点new_point
void del(int from,int to,int new_point){
    head[tot&1]=-1;//删边
    //三角形另一个顶点prev不用修改

  edges[tot].from=head[new_point],head[new_point]=tot,edges[tot].to=edges[tot&1].to;
}
//链式前向星遍历：处理x的所有相邻节点
for(int i=head[x];~i;i=edges[i].from){
    int y=edges[i].to; //存在一条边起点为x，终点为y
    //对y节点进行处理
}
```

图 4-3 链式前向星支持的操作。包括动态增添或删除边，以及遍历某个节点的所有出边。

4.2.2 并查集

不同于链式前向星解决的是必要的存储图结构的问题，从实现功能角度来看，在实现过程中引入并查集并不是一个必须的行为。但从实现效率角度来看，引入并查集降低了整个重构过程的算法时间复杂度瓶颈。

之所以会产生这样的效果，原因在于曲面简化过程中，会存在频繁的图结构变动。换句话说，查询“一个顶点是否被删除，在删除它之后新增的顶点编号”这个操作会出现多次，尤其是当简化过程集中于若干个模型平坦区域时。这时，采用朴素的链式维护算法是一种可行的选择。一条单链上除了头节点以外的所有节点都表示已经删除的顶点，头节点表示当前模型中最新产生的定点编号，而单链尾部到单链头部节点的

排列顺序表示整个细分过程中，节点删除又新增的历史记录。这样的做法维护了完整的曲面简化信息，也会带来大量指数级的冗余存储空间（每次边坍缩，都会在原本的单链上分裂出来一条单链）和全链遍历。但这些冗余空间和频繁遍历唯一的目的，仅仅在于找到当前链表的头节点。这使得这种做法需要的时空复杂度有了优化的可能。

如图 4-2（b）所示，本工作采用了路径压缩优化的并查集来解决这个问题。并查集的存在，使得节点替换关系被组织为树形结构（森林），这解决了原先链表分裂的冗余空间问题。而路径压缩的优化方式对应了仅关注树形结构根节点（对应原先链式结构头节点）的问题特征，使得树形结构中不至于出现过长的单链。在后续的章节中，有关于该部分在使用路径压缩优化的并查集情况下的时空复杂度分析。

使用路径压缩优化的并查集维护节点替换情况的示例代码如图 4-4 所示，它支持的操作包括并查集初始化、查询与修改某个节点所在树形结构根节点（数据结构空间的分配与释放由 malloc 和 free 操作完成）。

```
//并查集初始化，tag[i]表示删除i点后新产生顶点的id
for(int i=1;i<=num_points;i++) tag[i]=i;
//并查集查询
int find(int x){
    return tag[x]==x?x:tag[x]=find(tag[x]);
}
//通过该语句判断x点是否被删除，如果被删除，则获取删除它后新产生顶点的id
assert(x==find(x));
```

图 4-4 并查集的示例代码。使用路径压缩优化。

4.2.3 小根堆

在后续的小节中，会有关于曲面简化过程的详细阐述。而引入小根堆这一数据结构的作用，如图 4-2（c）所示，则是简化该算法在动态维护坍缩候选边集时的时间复杂度。C++的标准模板库中提供了一个现成的模板堆数据结构，即优先队列（priority queue）。但为了保持现有 C89 代码的跨平台特性和本身的高效运行效率，本工作没有使用优先队列，而是参照优先队列，实现了一个基于 C89 的堆数据结构。除此以外，本工作同样参照标准模板库中的向量（vector）实现了一个动态分配回收空间的数组，因为实现及原理较为简单的缘故，这里不再赘述。

本工作实现的小根堆的示例代码如图 4-5 所示，它支持的操作包括堆初始化、自定义比较器、插入元素、删除堆顶元素、取出堆顶元素、清空堆元素、查询堆元素数量、查找指定元素、查询堆中是否含有指定元素等。

```c
//堆支持操作
struct heap_s{ //堆定义
    unsigned int size; unsigned int count; const void *udata;
    int (*cmp) (const void *, const void *, const void *); //自定义比较器
    void * array[];
};
heap_t *heap_new(int (*cmp) (const void *,const void *,const void *udata),const
void *udata); //初始化新堆
void heap_free(heap_t * hp); //释放堆空间
int heap_offer(heap_t **hp_ptr, void *item); //插入元素
void *heap_poll(heap_t * hp); //去除堆顶元素
void *heap_peek(const heap_t * hp); //取出堆顶元素
void heap_clear(heap_t * hp); //清空堆元素
int heap_count(const heap_t * hp); //查询堆元素数量
void *heap_remove_item(heap_t * hp, const void *item); //去除堆中指定元素
int heap_contains_item(const heap_t * hp, const void *item); //查询堆中是否含有指定元素
```

图 4-5 小根堆的示例代码。实现操作参考 C++ STL 的优先队列。

4.3 朴素曲面简化实现

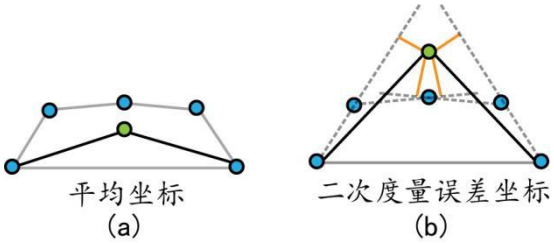

图 4-6 朴素曲面简化过程中坍缩边后新顶点的位置。（a）通过计算平均坐标获得的新顶点坐标；（b）通过最小化二次度量误差获得的新顶点坐标。

朴素曲面简化算法的实现主要涉及几个部分：代价函数设计、探索代价计算和最佳位置。下面详细介绍这几个部分的设计与实现。

4.3.1 代价函数设计

该算法的核心，是要每次找到一个新增顶点，使得曲面简化后，该顶点相邻的平面“变化”最小。

为了方便考虑，将常规曲面的三维结构简化为二维平面示例图。如图 4-6（a）所示如果采用平均坐标的方式来计算新增顶点的位置，那么简化后的平面形状会产生较大改变。因此，朴素曲面简化算法引入了一个名为“二次度量误差”的代价函数用于衡量坍缩边后新顶点到相邻直线的总距离。这个过程可以简单的推广到三维场景中，即代价函数用于衡量坍缩边后新顶点到原始曲面中相邻面的总距离。

设曲面中的顶点 V 存在 n 个相邻的三角面片，第 i 个面片所在平面的平面方程为 $p_i = [a_i\ b_i\ c_i\ d_i]$，且平面法向量是单位向量。则可以将顶点 V 相对于曲面的代价定义为该顶点到它的原始顶点相邻的所有平面的距离平方和。

对于未变化的顶点，因为它本身就在它的所有相邻平面上，所以代价为零。而距离平方和的变化值恒大于等于零，因此该算法的代价和总是在不断变大的。

为了更好的描述上述过程，将该定义转化为符号形式。

取$\Delta(V)$表示顶点 V 的代价，$\mathrm{planes}(V)$表示顶点 V 的相邻平面集合。则$\Delta(V)$可表示为如下形式。

$$\Delta(\mathrm{V}) = \Delta([\mathrm{V_x}\ \mathrm{V_y}\ \mathrm{V_z}\ 1]^{\mathrm{T}}) = \sum_{\mathrm{P}\in\mathrm{planes(V)}} (\mathrm{P}^{\mathrm{T}}\mathrm{V})^2 \tag{4-1}$$

进一步化简(4-1)，可以得到下式。

$$\Delta(\mathrm{V}) = \sum_{\mathrm{P}\in\mathrm{planes(V)}} (\mathrm{V}^{\mathrm{T}}\mathrm{P})(\mathrm{P}^{\mathrm{T}}\mathrm{V}) = \sum_{\mathrm{P}\in\mathrm{planes(V)}} \mathrm{V}^{\mathrm{T}}(\mathrm{PP})^{\mathrm{T}}\mathrm{V} \tag{4-2}$$

取K_P用于描述空间中任意一点到平面 P 的距离平方和。则K_P可表示为如下形式。

$$K_P = PP^T = \begin{bmatrix} a^2 & ab & ac & ad \\ ab & b^2 & bc & bd \\ ac & bc & c^2 & cd \\ ad & bd & cd & d^2 \end{bmatrix} \tag{4-3}$$

将(4-3)代入(4-2)可得下式。

$$\Delta(\mathrm{V}) = \mathrm{V}^{\mathrm{T}}(\sum_{\mathrm{P}\in\mathrm{planes(V)}} \mathrm{K_P})\mathrm{V} \tag{4-4}$$

矩阵 Q 表示与点 V 相邻的平面集合的K_P之和。

$$\mathrm{Q_V} = \sum_{\mathrm{P}\in\mathrm{planes(V)}} \mathrm{K_P} \tag{4-5}$$

将(4-5)代入(4-3)可得下式。

$$\Delta(\mathrm{V}) = \mathrm{V}^{\mathrm{T}}\mathrm{Q_V}\mathrm{V} \tag{4-6}$$

即每一个点只需要维护一个关于它的代价矩阵 Q，就能够在常数时间内计算坍缩一条边的代价。

4.3.2 坍缩代价算计和最佳位置

设当前需要坍缩边的两个顶点分别为V_1和V_2，新增顶点为V_{new}。则V_{new}的代价矩阵可以根据V_1和V_2的代价矩阵进行计算。

$$\mathrm{Q}_{\mathrm{V}_{\mathrm{new}}} = \mathrm{Q}_{\mathrm{V}_1} + \mathrm{Q}_{\mathrm{V}_2} = \begin{bmatrix} q_{11} & q_{12} & q_{13} & q_{14} \\ q_{21} & q_{22} & q_{23} & q_{24} \\ q_{31} & q_{32} & q_{33} & q_{34} \\ q_{41} & q_{42} & q_{43} & q_{44} \end{bmatrix} \tag{4-7}$$

需要注意的是，在计算新增顶点的代价矩阵的时候，并没有根据相邻平面的K_P计算。这会对计算结果造成一定的误差，因为严格来说，(4-6)式仅针对V_1和V_2相邻平面没有交集时成立。但这样的好处在于跟踪平面集合的时空复杂度是常数级别的，不会为整个算法流程引入额外的时间复杂度。

根据(4-6)可得到V_{new}的坍缩代价。

$$\Delta(\mathrm{V}_{\mathrm{new}}) = \mathrm{V}_{\mathrm{new}}{}^{\mathrm{T}}\mathrm{Q}_{\mathrm{V}_{\mathrm{new}}}\mathrm{V}_{\mathrm{new}} = q_{11}{V_x}^2 + q_{22}{V_y}^2 + q_{33}{V_z}^2 + q_{12}V_xV_y + q_{21}V_xV_y +$$

$$q_{13}V_xV_z + q_{31}V_xV_z + q_{23}V_yV_z + q_{32}V_yV_z + q_{41} + q_{42} + q_{43} + q_{44} \tag{4-8}$$

则问题转化为找到一个$\mathrm{V}_{\mathrm{new}}$使得$\Delta(\mathrm{V}_{\mathrm{new}})$最小，分别对 xyz 求偏导，通过$\frac{\partial\Delta(\mathrm{V}_{\mathrm{new}})}{\partial \mathrm{x}} = \frac{\partial\Delta(\mathrm{V}_{\mathrm{new}})}{\partial \mathrm{y}} = \frac{\partial\Delta(\mathrm{V}_{\mathrm{new}})}{\partial \mathrm{z}} = 0$ 确定V_{new}的最佳位置。这等价于求解下式。

$$\begin{bmatrix} q_{11} & q_{12} & q_{13} & q_{14} \\ q_{21} & q_{22} & q_{23} & q_{24} \\ q_{31} & q_{32} & q_{33} & q_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix} V_{new} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{4-9}$$

如果左侧矩阵是可逆的，则$\mathrm{V}_{\mathrm{new}}$的坐标可以直接通过下式求出。

$$V_{new} = \begin{bmatrix} q_{11} & q_{12} & q_{13} & q_{14} \\ q_{21} & q_{22} & q_{23} & q_{24} \\ q_{31} & q_{32} & q_{33} & q_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{4-10}$$

如果矩阵不可逆，则沿着V_1到V_2分段寻找最优的位置。本工作中步进系数为 0.1。

4.3.3 算法流程

朴素曲面简化算法的整体流程包括五个部分。

（一）计算曲面上每个顶点的初始代价矩阵

（二）找到所有点对，即可以坍缩的边集

（三）计算每条边坍缩的代价

（四）将边集推入一个以代价为关键字的小根堆

（五）迭代从堆顶取出代价最小的边删除，调整图结构，更新新顶点涉及的所有点对的代价

其中核心流程的示例代码如图 4-7 所示。

```
//曲面简化时边坍缩代价和新增顶点坐标计算
pair_t cur = pair_set[i];
int a = cur.a, b = cur.b;
mat4_t add_Q = mat4_add_mat4(mat_Q[a], mat_Q[b]);
mat4_t add_Q_1 = add_Q;
add_Q_1.m[3][0] = 0; add_Q_1.m[3][1] = 0; add_Q_1.m[3][2] = 0; add_Q_1.m[3][3] = 1;
vec4_t result = vec4_new(0, 0, 0, 0);
vec4_t p_1 = vec4_new((*positions)[a].x, (*positions)[a].y, (*positions)[a].z, 1);
vec4_t p_2 = vec4_new((*positions)[b].x, (*positions)[b].y, (*positions)[b].z, 1);
float determinant = mat4_determinant(add_Q_1);
if (fabsf(determinant) < EPSILON) {
    float k, min_k = 0.5f, min_cost = INF, cur_cost;
    for (k = 0.0f; k <= 1.0f; k += 0.1f) {
        result = vec4_add(vec4_mul(p_1, (1.0f - k)), vec4_mul(p_2, k));
        cur_cost = vec4_transpose_mul_vec4(vec4_mul_mat4(result, add_Q), result);
        if (cur_cost < min_cost) {
            min_cost = cur_cost;
            min_k = k;
        }
    }
    result = vec4_add(vec4_mul(p_1, (1.0f - min_k)), vec4_mul(p_2, min_k));
}
else {
    result = mat4_mul_vec4(mat4_inverse(add_Q_1), vec4_new(0, 0, 0, 1));
}
pair_set[i].cost = vec4_transpose_mul_vec4(vec4_mul_mat4(result, add_Q), result);
pair_set[i].new_position = vec3_new(result.x, result.y, result.z);
```

图 4-7 核心流程的示例代码。包括五个部分。

4.4 朴素曲面细分实现

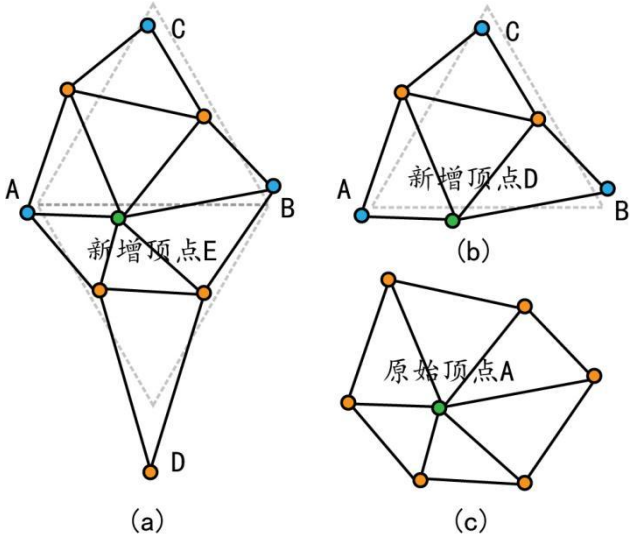

图 4-8 朴素曲面细分过程中顶点位置变化。（a）一次曲面细分过程中新产生的顶点和原始顶点的位置变化情况总览；（b）一次曲面细分过程中新产生的顶点；（c）一次曲面细分过程中原始顶点的位置变化。

相对朴素曲面简化算法，朴素曲面细分算法较为简单。朴素曲面细分算法的实现主要涉及新增顶点坐标计算和原始顶点坐标更新。下面详细介绍这两个部分的设计与实现。

4.4.1 新增顶点坐标计算

细分过程中，每一条边都会新增一个顶点，该顶点由它相邻两个三角面片中的所有顶点决定。对于曲面内部的新增顶点 E，如图 4-8（a）所示，E 的坐标可通过 A、B、C、D 四个顶点的坐标计算得到。

$$V_{E.x,y,z} = \frac{3}{8} * (V_{A.x,y,z} + V_{B.x,y,z}) + \frac{1}{8} * (V_{C.x,y,z} + V_{D.x,y,z}) \qquad (4\text{-}11)$$

(4-11)中的$\frac{3}{8}$和$\frac{1}{8}$是经过前人反复验证后得到的效果良好的权重系数，同时也反映了新增顶点坐标受到周围顶点坐标的影响程度：受到相邻顶点的影响较大，受到非相邻顶点的影响较小。

对于曲面边界的新增顶点 D，如图 4-8（b）所示，D 的坐标可通过它的相邻顶点 A、B 的坐标计算得到，而不受到 C 的坐标影响。

$$V_{D.x,y,z} = \frac{1}{2} * V_{A.x,y,z} + \frac{1}{2} * V_{B.x,y,z} \quad (4\text{-}12)$$

本工作实现的新增顶点坐标计算的示例代码如图 4-9 所示，代码中额外使用了渐进插值的方式[5]，进一步决定细分策略，即将顶点按照度数是否为 4，分为规则顶点和奇异顶点。对于规则顶点和奇异顶点的不同组合，采用不同的细分策略和权重分配。这里仅作为曲面细分环节中的简单优化，不会对算法总体流程和效果产生较大影响，因此不再展开描述。

```
//曲面细分时新顶点位置计算
if (p_opposite == -1) {
    // from -> to
    if ((regular_point[from] == 1 && regular_point[to] == 1) ||
(regular_point[from] == 0 && regular_point[to] == 0)) {
        edge_point = vec3_div(vec3_add((*positions)[from], (*positions)[to]), 2);
    }
    if ((regular_point[from] == 1 && regular_point[to] == 0)) {
        edge_point = vec3_div(vec3_add(vec3_mul((*positions)[from], 5),
vec3_mul((*positions)[to], 3)), 8);
    }
    if ((regular_point[from] == 0 && regular_point[to] == 1)) {
        edge_point = vec3_div(vec3_add(vec3_mul((*positions)[from], 3),
vec3_mul((*positions)[to], 5)), 8);
    }
}
else {
    // prev -> from -> p_opposite -> to
    edge_point = vec3_mul(vec3_div(vec3_add((*positions)[from], (*positions)[to]),
8), 3);
    edge_point = vec3_add(edge_point, vec3_div(vec3_add((*positions)[prev],
(*positions)[p_opposite]), 8));
}
```

图 4-9 新增顶点坐标计算的示例代码。根据顶点是否为奇异顶点区分考虑。

4.4.2 原始顶点坐标更新

细分过程中，每一个原始顶点也都会更新本身的坐标，这个的更新策略由所有与它相邻的原始顶点所决定。对于曲面上的原始顶点 A，如图 4-8（b）所示，A 的坐标可通过它周围的六个顶点的坐标计算得到。

设原始顶点 A 的坐标分别为$V_{A.x,y,z}$，周围共有 n 个相邻顶点，第 i 个顶点坐标为$V_{i.x,y,z}$。则更新后顶点 A 的坐标$V_{\bar{A}.x,y,z}$可由如下公式得到。

$$V_{\bar{A}.x,y,z} = (1 - n\beta) * V_{A.x,y,z} + \beta * \sum_{i=1}^{n} V_{i.x,y,z} \quad (4\text{-}13)$$

其中，β的计算公式如下。

$$\beta = \frac{1}{n}\left[\frac{5}{8} - \left(\frac{3}{8} + \frac{1}{4}\cos\frac{n}{2\pi}\right)^2\right] \tag{4-14}$$

本工作实现的原始顶点坐标更新的示例代码如图 4-10 所示。

```
1 //曲面细分时原始顶点位置更新
2 float n = 1.0 * cnt;
3 float beta = 1.0 / n * (0.625 - (0.375 + 0.25 * cos(2 * PI / n)) * (0.375 + 0.25 *
  cos(2 * PI / n)));
4 float coef = n * beta;
5 vec3_t new_position = vec3_add(vec3_mul((*positions)[from], 1.0 - coef),
  vec3_mul(sum_position, beta));
```

图 4-10 原始顶点坐标更新的示例代码。根据相邻的原始顶点决定新的坐标。

4.5 重构过程中的曲面细分

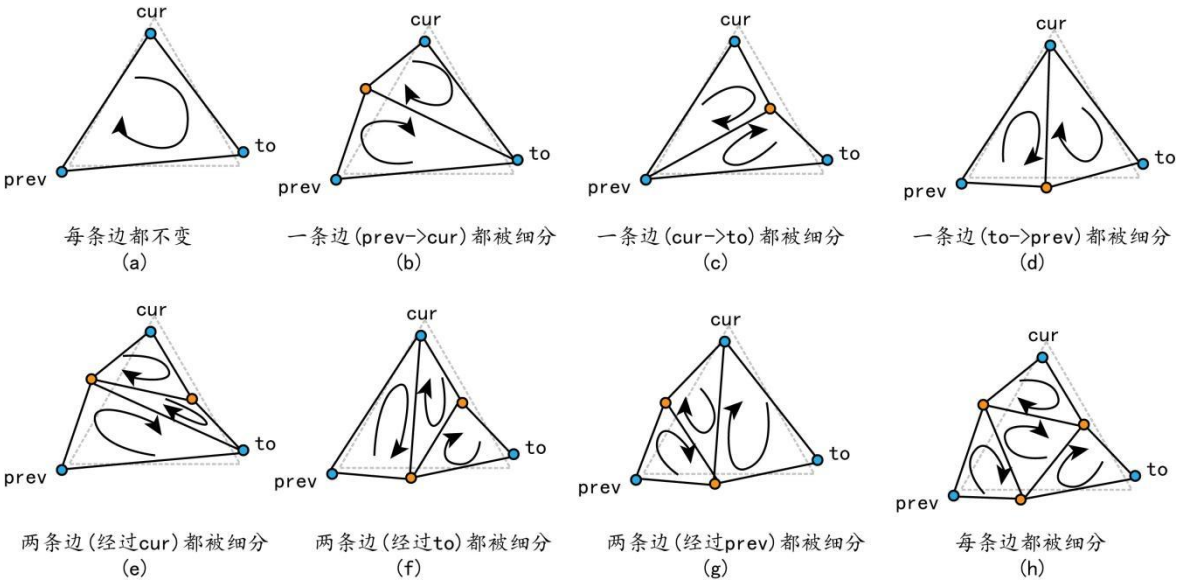

图 4-11 重构过程中的曲面细分。（a）一个三角面片中三条边都不需要细分时的顶点变化；（b）（c）（d）一个三角面片中存在一条边需要细分时的顶点变化；（e）（f）（g）一个三角面片中存在两条边需要细分时的顶点变化；（h）一个三角面片中三条边都需要细分时的顶点变化。

如图 4-11 所示，根据每个三角面片上每条边是否被细分这个条件，可以将重构过程中曲面细分的选择分为如下八种情况。

取当前三角面片的三个顶点按照顺时针顺序分别为 cur，to 和 prev。则每条边都不变的情况如图 4-11（a）所示；存在一条边被细分的情况如图 4-11（b、c、d）所

示；存在两条边被细分的情况如图 4-11（e、f、g）所示；每条边都被细分的情况如图 4-11（h）所示。

4.6 法向量与纹理插值

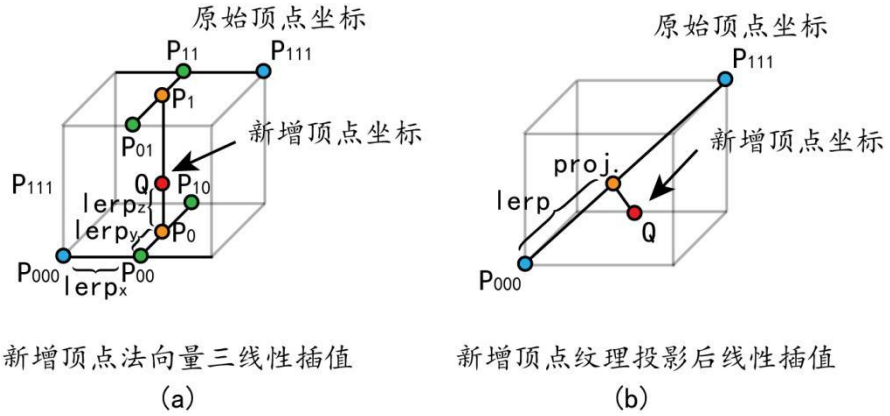

图 4-12 法向量与纹理插值。（a）使用三线性插值计算得到的新增顶点法向量；（b）使用投影后线性插值得到的新增顶点纹理。

这里记录了曲面简化和曲面细分算法在完成顶点的坐标计算后，根据顶点插值其他模型特征值的两个实现细节：法向量插值及纹理插值。在原本的工作里，法向量和纹理计算依赖更大规模的矩阵求解[9]，由于本工作更关注原始模型的顶点分布情况和程序运行效率，故没有再采用更复杂的矩阵求解算法，而是选择了插值算法。下面详细介绍两个插值方法的具体实现。其他特征值，如透明度，反射指数的插值原理类似，不再赘述。

4.6.1 法向量插值

如图 4-12（a）所示，法向量插值可以表述为如下过程。

设当前坐标系为右手坐标系，取一条边上两个原始顶点为P_{000}和P_{111}，新增顶点为 Q。P_{000}，P_{111}和 Q 的法向量分别为$Nor_{P_{000}}$，$Nor_{P_{111}}$和Nor_Q。新增顶点在P_{000}所在 XOY 平面上投影为P_0，在P_{111}所在 XOY 平面上投影为P_1。P_0在P_{000}所在 YOZ 平面投影为P_{00}，在P_{111}所在 XOZ 平面投影为P_{10}。P_1在P_{000}所在 YOZ 平面投影为P_{01}，在P_{111}所在 XOZ 平面投影为P_{11}。

则可据此计算新增顶点 Q 在 XYZ 三个维度上的插值系数$lerp_x$，$lerp_y$和$lerp_z$。

$$lerp_x = \frac{Q.x - P_{000}.x}{P_{111}.x - P_{000}.x} \tag{4-15}$$

$$lerp_y = \frac{Q.y - P_{000}.y}{P_{111}.y - P_{000}.y} \tag{4-16}$$

$$lerp_z = \frac{Q.z - P_{000}.z}{P_{111}.z - P_{000}.z} \tag{4-17}$$

需要注意的是，新增顶点坐标并不保证位于P_{000}和P_{111}所构成的立方体内，但新增顶点的纹理坐标需要保证位于P_{000}和P_{111}所对应的纹理坐标之间。因此$lerp_x$，$lerp_y$和$lerp_z$理应满足如下限制。

$$0 \le lerp_{x,y,z} \le 1 \tag{4-18}$$

对于不满足限制的$lerp_x$，$lerp_y$和$lerp_z$，会被截取到[0,1]之间。

$$lerp_{x,y,z} = \begin{cases} 0, & lerp_{x,y,z} \le 0 \\ 1, & lerp_{x,y,z} \ge 1 \\ lerp_{x,y,z}, & else \end{cases} \tag{4-19}$$

于是可求出新增顶点 Q 的法向量Nor_Q。

$$Nor_{Q.x} = Nor_{P_{000}.x} + (Nor_{P_{111}.x} - Nor_{P_{000}.x}) * lerp_x \tag{4-20}$$

$$Nor_{Q.y} = Nor_{P_{000}.y} + (Nor_{P_{111}.y} - Nor_{P_{000}.y}) * lerp_y \tag{4-21}$$

$$Nor_{Q.z} = Nor_{P_{000}.z} + (Nor_{P_{111}.z} - Nor_{P_{000}.z}) * lerp_z \tag{4-22}$$

考虑到标准模型中所有法向量均需要单位化，方可得到最终结果。

$$Nor_{Q.x,y,z} = \frac{Nor_{Q.x,y,z}}{|Nor_{Q.x,y,z}|} \tag{4-23}$$

本工作实现的法向量插值的示例代码如图 4-13 所示，通过引入矩阵，能够在常数时间内完成插值计算。

```
//法向量插值
float n_x, n_y, n_z;
if (fabsf(p_1.x - p_2.x) < EPSILON) n_x = 0.0f;
else n_x = (result.x - p_1.x) / (p_2.x - p_1.x);
if (fabsf(p_1.y - p_2.y) < EPSILON) n_y = 0.0f;
else n_y = (result.y - p_1.y) / (p_2.y - p_1.y);
if (fabsf(p_1.z - p_2.z) < EPSILON) n_z = 0.0f;
else n_z = (result.z - p_1.z) / (p_2.z - p_1.z);
n_x = float_saturate(n_x), n_y = float_saturate(n_y), n_z = float_saturate(n_z);
pair_set[i].new_normal = vec3_normalize(vec3_lerp_vec3(positions_normals[a],
positions_normals[b], n_x, n_y, n_z));
```

图 4-13 法向量插值的示例代码。在常数时间内完成三个维度的插值计算。

4.6.2 纹理插值

如图 4-12（b）所示，纹理插值可以表述为如下过程。

设当前坐标系为右手坐标系，取一条边上两个原始顶点为P_{000}和P_{111}，新增顶点为 Q。P_{000}，P_{111}和 Q 的纹理坐标分别为$Tex_{P_{000}}$，$Tex_{P_{111}}$和Tex_Q。P_{000}和P_{111}为$length$。

则可据此计算新增顶点 Q 从 XYZ 三个维度投影到 UV 两个维度上的插值系数$lerp_r$。

$$lerp_r = \frac{(Q-P_{000})\cdot\frac{P_{111}-P_{000}}{|P_{111}-P_{000}|}}{length} \tag{4-24}$$

需要注意的是，新增顶点坐标并不保证位于P_{000}和P_{111}所构成的立方体内，但新增顶点的纹理坐标需要保证位于P_{000}和P_{111}所对应的纹理坐标之间。因此$lerp_r$理应满足如下限制。

$$0 \le lerp_r \le 1 \tag{4-25}$$

对于不满足限制的$lerp_r$，会被截取到[0,1]之间。

$$lerp_r = \begin{cases} 0, & lerp_r \le 0 \\ 1, & lerp_r \ge 1 \\ lerp_r, & else \end{cases} \tag{4-26}$$

于是可求出新增顶点 Q 的纹理坐标Tex_Q。

$$Tex_{Q.u,v} = Tex_{P_{000}.u,v} + (Tex_{P_{111}.u,v} - Tex_{P_{000}.u,v}) * lerp_r \tag{4-27}$$

本工作实现的纹理插值的示例代码如图 4-14 所示，通过引入矩阵，能够在常数时间内完成插值计算。

```
//纹理坐标插值
float t_r;
float proj_length = vec3_dot(vec3_sub(pair_set[i].new_position,
vec3_from_vec4(p_1)), vec3_normalize(vec3_sub(vec3_from_vec4(p_2),
vec3_from_vec4(p_1))));
float edge_length = vec3_length(vec3_sub(vec3_from_vec4(p_2),
vec3_from_vec4(p_1)));
if (edge_length < EPSILON) t_r = 0.0f;
else t_r = proj_length / edge_length;
t_r = float_saturate(t_r);
pair_set[i].new_texcoord = vec2_lerp_vec2(positions_texcoords[a],
positions_texcoords[b], t_r, t_r);
```

图 4-14 纹理坐标插值的示例代码。在常数时间内完成两个维度的插值计算。

4.7 实现细节

这里列举了实现过程中最主要的三个实现细节：边界处理、坍缩限制、迭代次数。下面详细介绍三个细节的考虑原因和处理方式。

4.7.1 边界处理

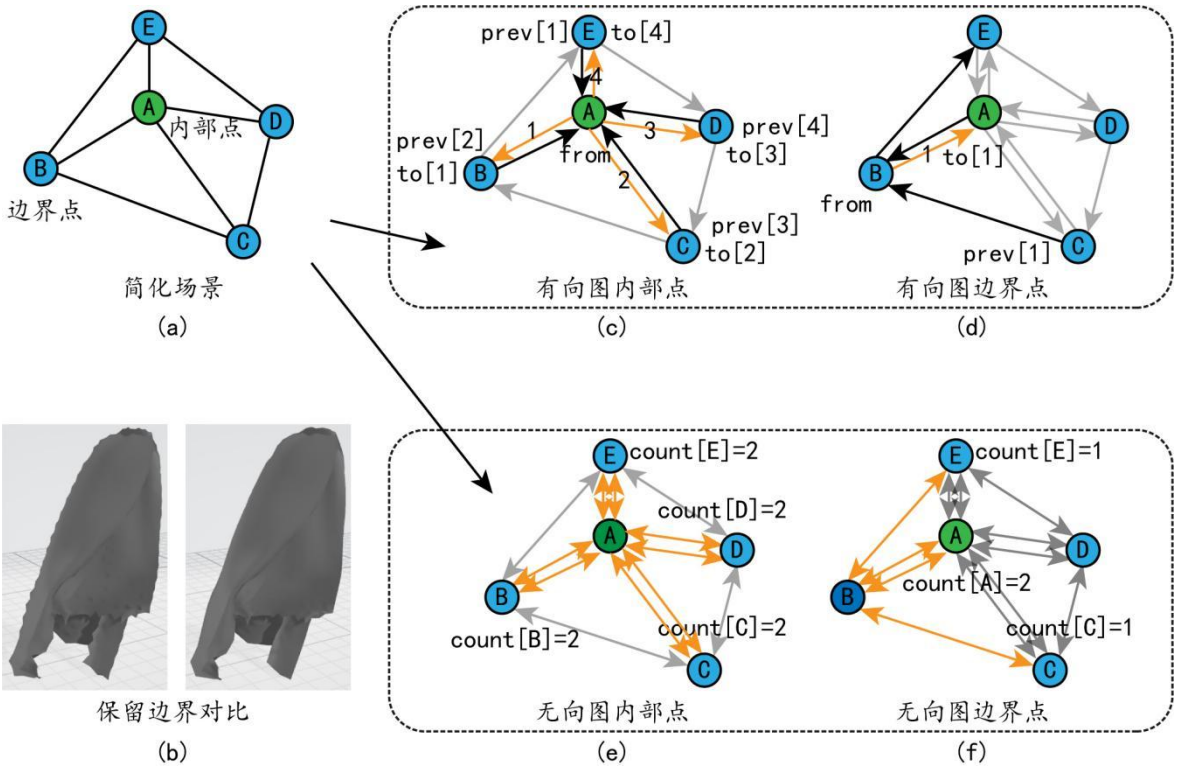

图 4-15 边界处理。（a）简化后的示例模型；（b）是否对边检处理获得的模型对比；（c）有向图中内部点的特征；（d）有向图中边界点的特征；（e）无向图中内部点的特征；（f）无向图中边界点的特征。

按照顶点是否处于曲面边界，可将曲面中的顶点分为内部点和边界点。重构算法中的曲面简化部分和曲面细分部分都需要考虑对于内部点和边界点采取不同的策略。在图 4-15（b）中，可以看到是否区分内部点和边界点对结果的影响。

之所以不区分内部点和边界点，会带来较差的视觉效果，是因为边界点的位置更新，会不可逆的对曲面轮廓产生影响。在使用前文提到的顶点位置变更公式的前提下，封闭曲面不会收到影响（封闭曲面并不存在边界点），但大多数开放曲面流畅的外延都会变得曲折。

对内部点采取的策略，本工作已经在前文描述过。而对于边界点的操作，这里选择了一个简洁的做法：当需要更新顶点位置时，如果该顶点为边界点，则跳过该顶点的位置。

因为重构算法中的曲面简化部分和曲面细分部分对于曲面建模方式不同（无向图和有向图），因此需要分别找到这两种图中，一个顶点是边界点的充要条件。

本工作实现的有向图边界点检测的示例代码如图 4-16 所示，通过“以当前顶点作为起点的任意出边指向顶点，是否都存在反向边”来检查当前顶点是否是有向图边界点。这个规则，可以通过图 4-15（c、d）更好的理解。

```
//有向图边界点判断
for (from = 0; from < num_points; from++) {
    for (j = head[from]; ~j; j = edges[j].from) {
        int to = edges[j].to;
        int flag = 0;
        for (k = head[from]; ~k; k = edges[k].from) {
            int prev = edges[k].prev;
            if (prev == to) {
                flag = 1;
                break;
            }
        }
        if (flag == 0) {
            on_boundary[from] = 1;
            break;
        }
    }
}
```

图 4-16 有向图边界点检测的示例代码。用于检查所有顶点是否为有向图边界点。

本工作实现的无向图边界点检测的示例代码如图 4-17 所示，通过“以当前顶点作为起点的任意出边指向顶点，与当前顶点之间的边都存在重边”来检查当前顶点是否是无向图边界点。这个规则，可以通过图 4-15（e、f）更好的理解。

```
//无向图边界点判断
for (from = 0; from < num_points; from++) {
    int* temp = NULL;
    for (j = head[from]; ~j; j = edges[j].from) {
        int to = edges[j].to;
        if (count_visit[to] == 0) darray_push(temp, to);
        count_visit[to]++;
    }
    int size = darray_size(temp);
    if (temp != NULL) for (i = 0; i < size; i++) {
        if (count_visit[temp[i]] == 1) {
            on_boundary[from] = 1;
            break;
        }
    }
    if (temp != NULL) for (i = 0; i < size; i++) count_visit[temp[i]] = 0;
    darray_free(temp);
}
```

图 4-17 无向图边界点检测的示例代码。用于检查所有顶点是否为无向图边界点。

4.7.2 坍缩限制

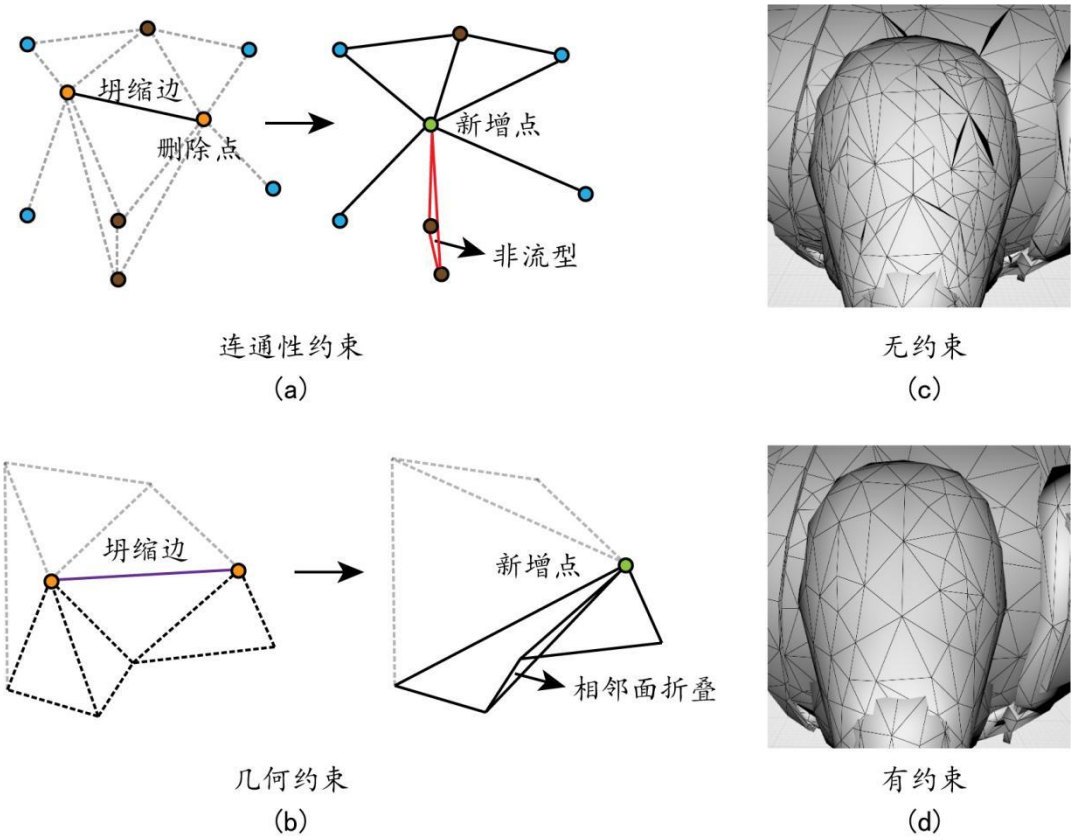

图 4-18 坍缩限制。（a）需要使用连通性约束的抽象场景；（b）需要使用几何约束的抽象场景；（c）无连通性约束和几何约束处理得到的模型；（d）有连通性约束和几何约束处理得到的模型。

边坍缩算法需要使用额外的约束来防止出现模型空洞[3]。通过图 4-18（c）和图 4-18（d）的对比，可以看到是否使用额外约束的情况下，边坍缩产生模型的对比。

（一）在坍缩边的每一侧，只能合并一对边。这个约束被定义为连通性约束。考虑图 4-8（a）中的例子，实线所示的边正在确定是否需要坍缩：如果坍缩后，出现了超过一对边被合并，即红色的三角面片呈现非流形形态。

本工作实现的连通性约束的示例代码如图 4-19 所示，通过计算坍缩边的两个顶点（必须正好有两个）的相邻顶点来检查连通性约束。

```
//连通性约束
int same_point_count = 0;
int* point_set = NULL;
for (cur_id = 0; cur_id <= 1; cur_id++) {
    int x = cur_edge[cur_id];
    for (i = head[x]; ~i; i = edges[i].from) {
        int to = edges[i].to;
        if (cur_id == 0) {
            darray_push(point_set, to);
            continue;
        }
        int point_set_size = darray_size(point_set);
        if(point_set != NULL) for (j = 0; j < point_set_size; j++)
          if (point_set[j] == to) {
            same_point_count++;
            break;
        }
    }
}
```

图 4-19 连通性约束的示例代码。通过坍缩边的两个相邻顶点来检查连通性约束。

（二）边坍缩期间，三角面片不得翻转。这个约束被定义为连通性约束。考虑图4-18（c）中的例子，其中箭头标明的实现三角面片的法线在坍缩过程中翻转。

在坍缩边的每一侧，只能合并一对边。这个约束被定义为连通性约束。考虑图4-18（a）中的例子，实线所示的边正在确定是否需要坍缩：如果坍缩后，出现了超过一对边被合并，即红色的三角面片呈现非流形形态。

本工作实现的几何约束的示例代码如图 4-20 所示，对于保留的三角形，可以通过计算翻转前和翻转后三角形法线之间的角度来检查几何约束。

```
//几何约束
int edge_flip = 0;
for (cur_id = 0; cur_id <= 1; cur_id++) {
    if (edge_flip == 1) break;
    int x = cur_edge[cur_id];
    for (i = head[x]; ~i; i = edges[i].from) {
        int to = edges[i].to;
        int prev = edges[i].prev;
        if (tag[to] != 0) continue;
        if (tag[prev] != 0) continue;
        vec4_t normal = vec3_plane_normal((*positions)[x], (*positions)[to],
(*positions)[prev]);
        vec4_t new_normal = vec3_plane_normal(new_pos, (*positions)[to],
(*positions)[prev]);
        float dot_res = vec3_dot(vec3_from_vec4(normal),
vec3_from_vec4(new_normal));
        if (dot_res < 0) {
            edge_flip = 1;
            break;
        }
    }
}
```

图 4-20 几何约束的示例代码。通过计算翻转前和翻转后三角形法线之间的角度来检查几何约束。

4.7.3 迭代次数

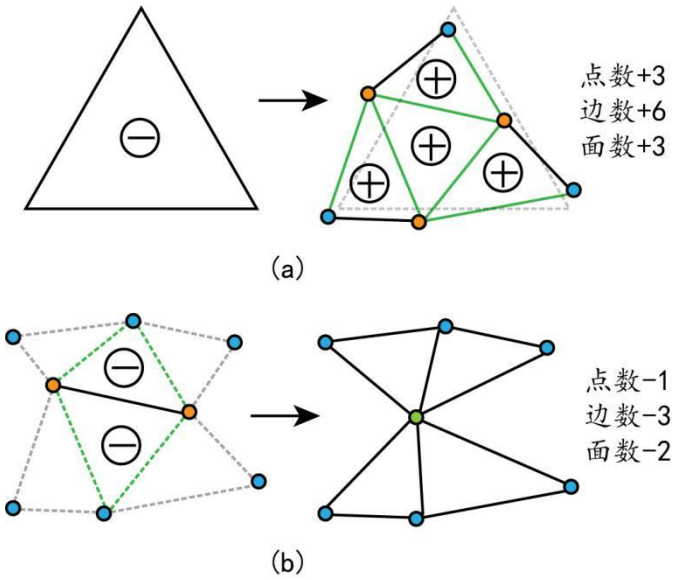

图 4-21 迭代次数计算。（a）一次朴素曲面细分过程中，模型点数、边数、面数的变化；（b）一次朴素曲面简化过程中，模型点数、边数、面数的变化。

为了确保曲面重构过程不会对模型复杂程度产生较大影响，从而在尽可能保证原本渲染效率的情况下，达到更好的渲染效果。本工作进行了迭代次数的计算，为了更好的描述计算结果，在该部分引入如下符号。

曲面简化次数 Sim_Num（Simplification Number），曲面细分次数 Sub_Num（Subdivision Number）。

根据图 4-21（a、b）给出的一次朴素曲面细分和一次朴素曲面简化过程中模型点数、边数、面数的变化。在保证点数不变的情况下，曲面简化次数与曲面细分次数满足关系式 $\mathrm{Sim\_Num} = 3 * \mathrm{Sub\_num}$；在保证点数不变的情况下，曲面简化次数与曲面细分次数满足关系式 $\mathrm{Sim\_Num} = 2 * \mathrm{Sub\_num}$（该式经过化简）；在保证点数不变的情况下，曲面简化次数与曲面细分次数满足关系式 $2 * \mathrm{Sim\_Num} = 3 * \mathrm{Sub\_num}$。因此，可得出最终在确定曲面简化次数下，保证模型复杂程度的合适曲面细分次数范围为 $\frac{1}{3} * \mathrm{Sim\_Num} \leq \mathrm{Sub\_Num} \leq \frac{2}{3} * \mathrm{Sim\_Num}$。

4.8 本章小结

本章主要介绍了本工作提出的曲面重构相关算法的具体实现，首先阐述了通过表面模型的介绍引入分析，其次叙述了朴素曲面简化和曲面细分的实现。从而在此基础上，完成了重构算法中曲面细分的设计。最后加入的纹理法向量插值和实现细节的叙述也更好的让该算法适应各类灵活变化的情景。

第 5 章 结果展示

这一章主要展示了本工作提出的曲面重构相关算法的实现结果。具体实现内容包括重构结果线框模型和对应渲染结果的展现，特殊重构结果介绍，以及性能与复杂度分析。

5.1 重构结果

这里列举了效果部分中最主要的两个展现部分：线框结果及渲染结果。下面通过线框结果和渲染结果对照，更多渲染结果及特殊重构结果介绍本工作提出的曲面重构相关算法的实现结果。

5.1.1 线框结果与渲染结果对照

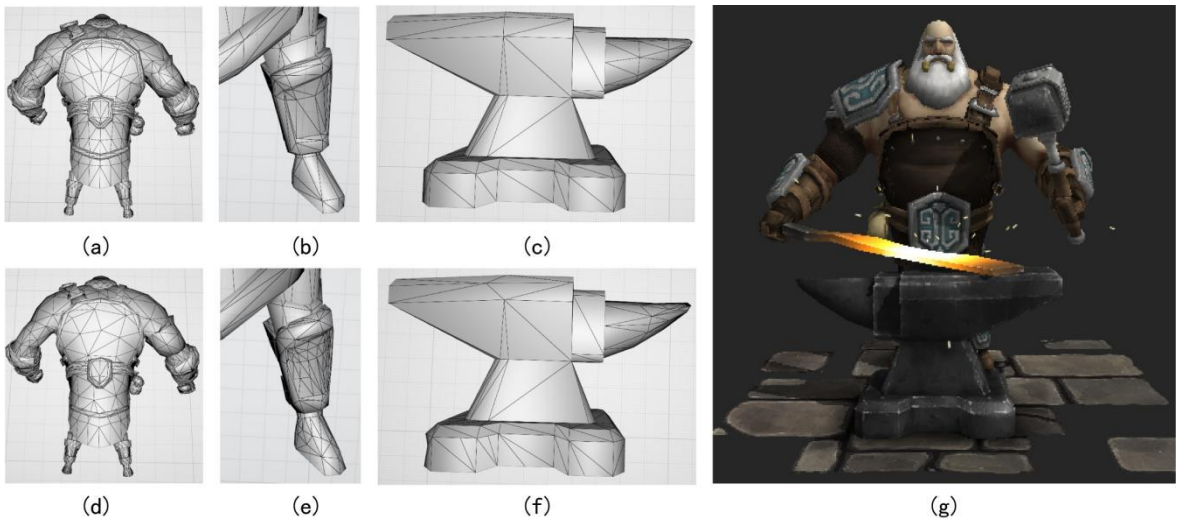

图 5-1 线框结果与渲染结果对照（人物模型：工匠）。（a）原始模型背部；（b）原始模型护膝；（c）原始模型环境静物；（d）重构模型背部；（e）重构模型护膝；（f）重构模型环境静物；（g）最终渲染结果。

如图 5-1 所示，该模型为人物模型：工匠。共计包含 7k 三角面片，4.2k 顶点。在实际渲染过程中，平均帧率为 40 帧/秒，每帧平均渲染时间为 25 毫秒。

通过图 5-1（a、d），图 5-1（b、e），图 5-1（c、f）三组模型不同部位对比，可以注意到曲面重构算法进行的改变。在图 5-1（a）中相对平坦的背部，被简化掉

了较多的三角面片，在基本没有丢失特征的情况下，变成了图 5-1（d）的状态；而在图 5-1（b、c）中相对崎岖的护膝和环境静物，进行了较多的细分操作，在使得曲面更为光滑的情况下，变成了图 5-1（e、f）的状态。

最终曲面重构后的渲染效果如图 5-1（g）所示。

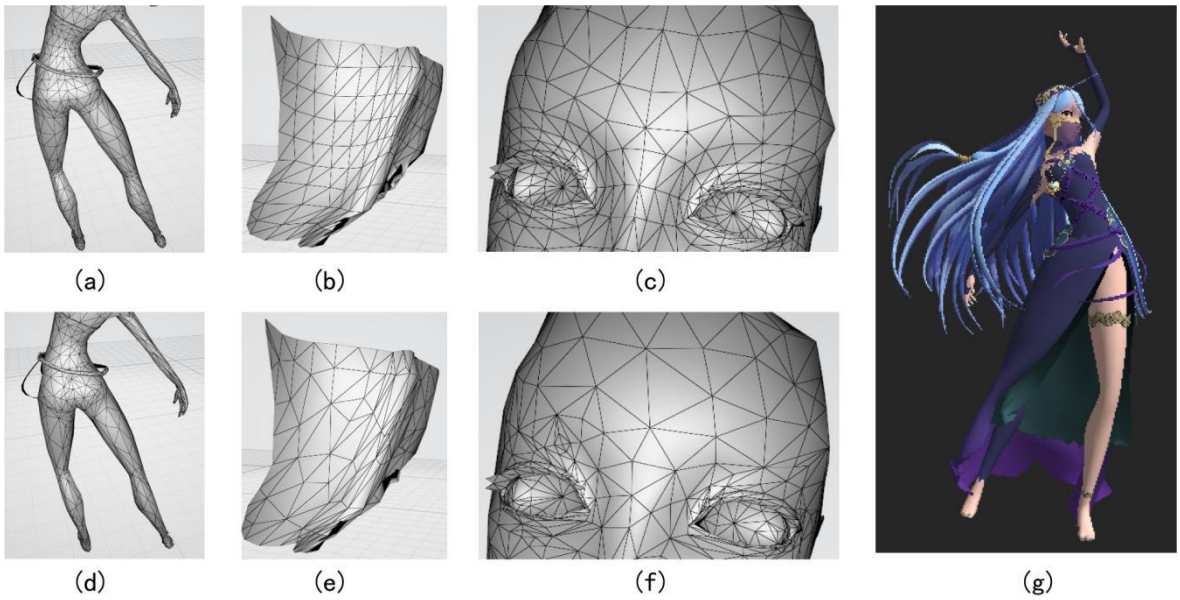

图 5-2 线框结果与渲染结果对照（人物模型：异域舞女，来自知名游戏《火焰纹章：命运》）。（a）原始模型背部；（b）原始模型裙角；（c）原始模型面部；（d）重构模型背部；（e）重构模型裙角；（f）重构模型面部；（g）最终渲染结果。

如图 5-2 所示，该模型为人物模型：异域舞女。共计包含 19.7k 三角面片，19.5k 顶点。在实际渲染过程中，平均帧率为 36 帧/秒，每帧平均渲染时间为 28 毫秒。

通过图 5-2（a、d），图 5-2（b、e），图 5-2（c、f）三组模型不同部位对比，可以注意到曲面重构算法进行的改变。在图 5-2（a）中相对平坦的背部和腿部，图 5-2（b）中相对平坦的裙边下摆，图 5-2（c）中相对平坦的额头，被简化掉了较多的三角面片，在基本没有丢失特征的情况下，变成了图 5-2（a、b、c）的状态；而在图 5-2（a）中相对崎岖的胯部，图 5-2（b）中相对崎岖的裙边褶皱，图 5-2（c）中相对崎岖的眼眶，进行了较多的细分操作，在使得曲面更为光滑的情况下，变成了图 5-2（d、e、f）的状态。

最终曲面重构后的渲染效果如图 5-2（g）所示。

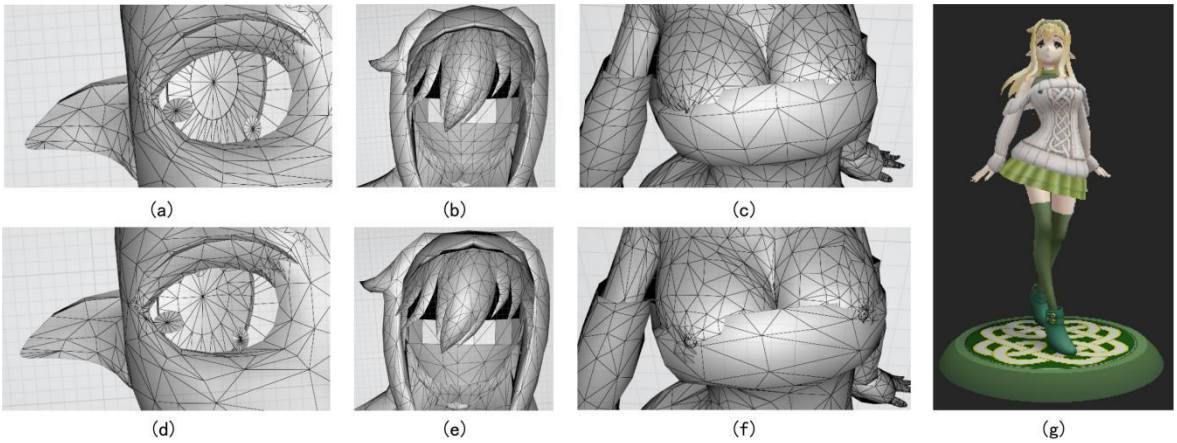

图 5-3 线框结果与渲染结果对照（人物模型：精灵女孩）。（a）原始模型眼睛；（b）原始模型头部；（c）原始模型胸部；（d）重构模型眼睛；（e）重构模型头部；（f）重构模型胸部；（g）最终渲染结果。

如图 5-3 所示，该模型为人物模型：精灵女孩。共计包含 7.5k 三角面片，7.6k 顶点。在实际渲染过程中，平均帧率为 91 帧/秒，每帧平均渲染时间为 10 毫秒。

通过图 5-3（a、d），图 5-3（b、e），图 5-3（c、f）三组模型不同部位对比，可以注意到曲面重构算法进行的改变。在图 5-3（a）中相对平坦的面颊，图 5-3（b）中相对平坦的后脑，图 5-3（c）中相对平坦的乳房表面，被简化掉了较多的三角面片，在基本没有丢失特征的情况下，变成了图 5-3（a、b、c）的状态；而在图 5-3（a）中相对崎岖的眼窝，图 5-3（b）中相对崎岖的发梢，图 5-3（c）中相对崎岖的乳头，进行了较多的细分操作，在使得曲面更为光滑的情况下，变成了图 5-3（d、e、f）的状态。

最终曲面重构后的渲染效果如图 5-3（g）所示。

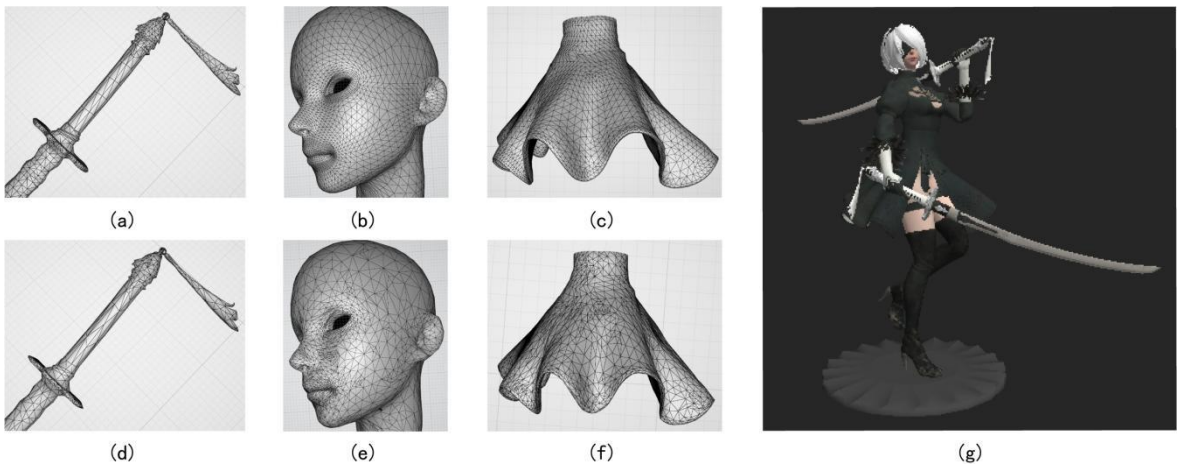

图 5-4 线框结果与渲染结果对照（人物模型：ヨルハ 2 号 B 型，来自知名游戏《尼尔机械纪元》）。（a）原始模型武士刀；（b）原始模型头部；（c）原始模型裙角；（d）重构模型武士刀；（e）重构模型头部；（f）重构模型裙角；（g）最终渲染结果。

如图 5-4 所示，该模型为人物模型：精灵女孩。共计包含 62.3k 三角面片，32.3k 顶点。在实际渲染过程中，平均帧率为 46 帧/秒，每帧平均渲染时间为 21 毫秒。

通过图 5-4（a、d），图 5-4（b、e），图 5-4（c、f）三组模型不同部位对比，可以注意到曲面重构算法进行的改变。在图 5-4（a）中相对平坦的刀柄、刀身，图 5-4（b）中相对平坦的面颊，图 5-4（c）中相对平坦的裙角下摆，被简化掉了较多的三角面片，在基本没有丢失特征的情况下，变成了图 5-4（a、b、c）的状态；而在图 5-4（a）中相对崎岖的刀格，图 5-4（b）中相对崎岖的燕窝，图 5-4（c）中相对崎岖的裙边褶皱，进行了较多的细分操作，在使得曲面更为光滑的情况下，变成了图 5-4（d、e、f）的状态。

最终曲面重构后的渲染效果如图 5-4（g）所示。

5.1.2 更多渲染结果

图 5-5 更多渲染结果。（a）（人物模型：麦克雷，来自知名游戏《守望先锋》）；（b）（幻想生物：半人马）；（c）（幻想生物：万圣节女巫）；（d）（人物模型：黑魔导士，来自知名游戏《最终幻想Ⅸ》）；（e）（场景模型：18 世纪灯塔）。

如图 5-5 所示，本工作在更多模型上进行了测试。除了上一小节里的人物模型外，还包括了不同风格（写实、动画等）的幻想生物模型，场景模型。并且在这些模型上，都获得了预期的渲染效果，证明了算法的普适性。

其中图 5-5（a）模型为人物模型：麦克雷。共计包含 2.6k 三角面片，1.9k 顶点。在实际渲染过程中，平均帧率为 121 帧/秒，每帧平均渲染时间为 8 毫秒。其中图 5-5（b）模型为幻想生物：半人马。共计包含 14.6k 三角面片，13.7k 顶点。在实际渲染过程中，平均帧率为 68 帧/秒，每帧平均渲染时间为 15 毫秒。其中图 5-5（c）模型为幻想生物：万圣节女巫。共计包含 4.4k 三角面片，2.4k 顶点。在实际渲染过程中，平均帧率为 72 帧/秒，每帧平均渲染时间为 13 毫秒。其中图 5-5（d）模型为人物模型：黑魔导士。共计包含 12.1k 三角面片，12.9k 顶点。在实际渲染过程中，平均帧率为 70 帧/秒，每帧平均渲染时间为 14 毫秒。其中图 5-5（e）模型为场景模型：18 世纪灯塔。共计包含 11k 三角面片，4.4k 顶点。在实际渲染过程中，平均帧率为 42 帧/秒，每帧平均渲染时间为 24 毫秒。

5.1.3 特殊重构结果

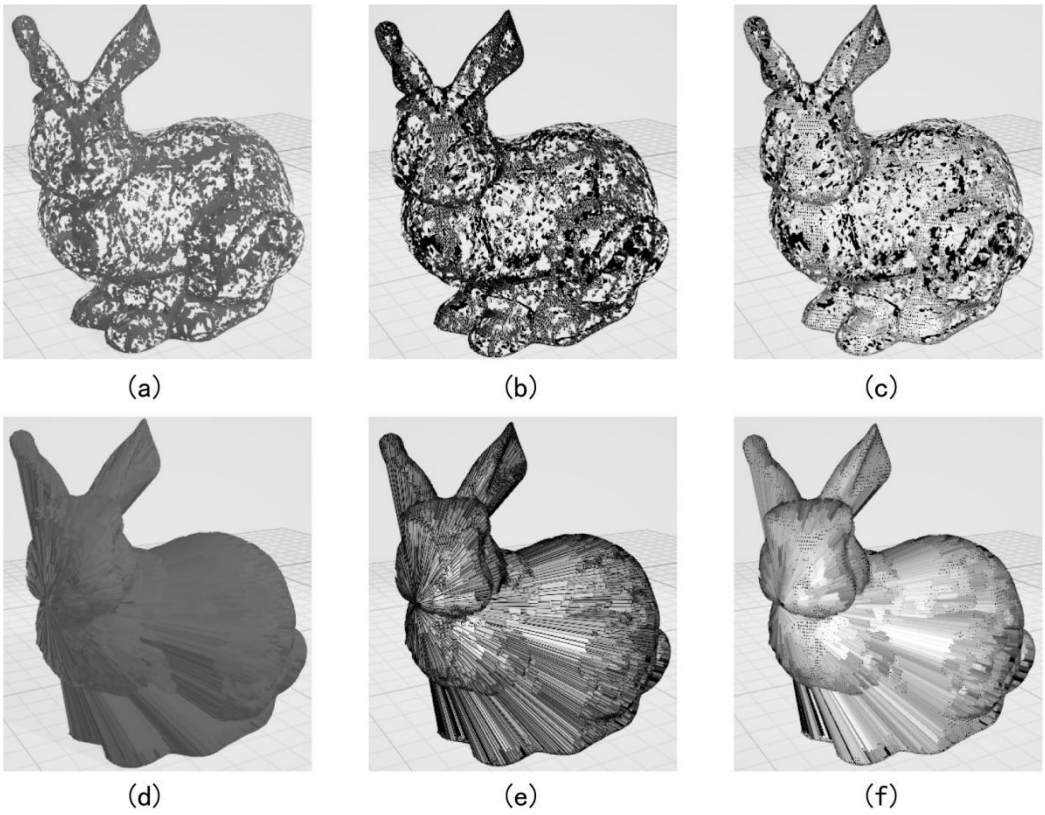

图 5-6 特殊重构结果。（a）镂空斯坦福兔子模型概览；（b）镂空斯坦福兔子模型三角面片分布；（c）镂空斯坦福兔子模型顶点分布；（d）单点射线斯坦福兔子模型概览；（e）单点射线斯坦福兔子模型三角面片分布；（f）单点射线斯坦福兔子模型顶点分布。

本工作在编程实验过程中，还因为某些代码“错误”，产生了一些奇妙的特殊重构结果。如图 5-6 所示，这些特殊的重构结果大致包含两类：镂空效果、单点射线效果。图 5-6（a、b、c）从概览、三角面片分布、顶点分布三个部分展示了带有镂空效果的斯坦福兔子模型；图 5-6（d、e、f）从概览、三角面片分布、顶点分布三个部分展示了带有单点射线效果的斯坦福兔子模型。

5.2 性能和复杂度分析

这里列举了本工作提出的重构算法的的其他值得描述的内容，包括程序信息与运行环境及时空复杂度分析。

5.2.1 程序信息与运行环境

本工作所使用程序完整信息如下。

（一）使用语言：模型文件预处理使用 Python 语言，内核代码使用 C89 语言。

（二）编译方式：CMake 3.10。

（三）代码量：9286 行，55 份源代码文件。

（四）运行平台：跨平台，目前已在 Linux（支持发行版本包括 Ubuntu、Debian、Fedora、RHEL、openSUSE、SUSE）、MacOS、Windows 三个平台上正常测试运行。

（五）支持数据类型：TGA、HDR、OBJ、glTF。

（六）开源代码框架许可证：MIT 许可证。

本工作获取运行结果时使用的硬件配置如下。

（一）CPU：Intel(R) Core(TM) i5-10210U CPU 160GHz

（二）内存：16GB

（三）显卡：NVIDIA GeForce MX350

5.2.2 时空复杂度分析

本工作进行了性能和复杂度分析，并对实际运行结果进行了完整记录。为了更好的描述复杂度，在该部分引入如下符号。

顶点数 V（Vertex），纹理坐标数 T（Texcoord），法向量数 N（Normal），三角面片数 F（Face），简化三角面片比率 R（Ratio）。

以下是使用这些符号罗列出的模型重构过程中主要部分时间复杂度与空间复杂度。

在曲面简化过程中，时空复杂度分析如表 5-1。

表 5-1 曲面简化各过程时间复杂度与空间复杂度

	时间复杂度	空间复杂度
读取模型参数	$O(\max\{V,T,N\})$	$O(\max\{V,T,N\})$
计算模型点线面数量	$O(\max\{V,T,N,F\})$	$O(\max\{V,T,N,F\})$
申请设计数据结构空间	$O(\max\{2V,12F\})$	$O(\max\{2V,12F\})$
根据每个三角面片逆时针次序建有	$O(6F)$	$O(12F)$

向图		
区分无向图中边界点和内部点	$O(V\log_2 V)$	$O(V\log_2 V)$
邻接表转存无向图	$O(\max\{12F\})$	$O(\max\{V,3F\})$
计算每个顶点相邻三角片面所在平面法向量	$O(V\log_2 V)$	$O(2V)$
计算顶点代价矩阵	$O(V\log_2 V)$	$O(2V)$
建立顶点和纹理、法向量映射	$O(6F)$	$O(2V)$
计算边集所有边坍缩代价和新增点最优位置	$O(48F)$	$O(6F)$
根据最优位置插值新增点纹理和法向量	$O(\max\{T,N\})$	$O(\max\{T,N\})$
将每条边代价组合后插入自定义小根堆	$O(\max\{6F\log_2(6F)\})$	$O(\max\{6F\log_2(6F)\})$
初始化并查集	$O(2V)$	$O(2V)$
计算需要坍缩的边数	$O(1)$	$O(1)$
循环从堆中取出代价最小的边	$O(RF\log_2(RF))$	$O(RF\log_2(RF))$
查询并查集，防止边重复坍缩	$O(RF\log_2(RF))$	$O(2V)$
检测连通性约束和几何约束，防止出现模型孔洞	$O(RF\log_2(RF))$	$O(RF\log_2(RF))$
坍缩边并更新无向图结构	$O(RF\log_2(RF))$	$O(RF\log_2(RF))$
计算新边的坍缩代价和新增点最优位置	$O(48RF\log_2(RF))$	$O(RF\log_2(RF))$
修改并查集	$O(RF\log_2(RF))$	$O(2V)$
离散化点集	$O(V\log_2 V)$	$O(V\log_2 V)$
释放数据结构空间	$O(\max\{2V,T,N,12F\})$	$O(\max\{2V,T,N,12F\})$

重新计算边集误差过程中，时空复杂度分析如表 5-2。

表 5-2 重新计算边集误差各过程时间复杂度与空间复杂度

	时间复杂度	空间复杂度

重新计算边集坍缩代价	$O(48F\log_2(F))$	$O(48F\log_2(F))$
计算需要细分的边数	$O(1)$	$O(1)$
新建小根堆或许需要细分的边集	$O(RF\log_2(RF))$	$O(RF\log_2(RF))$

在曲面细分过程中，二次度量误差较大的边会被细分。其过程中时空复杂度分析如表 5-3。

表 5-3 曲面细分各过程时间复杂度与空间复杂度

	时间复杂度	空间复杂度
模型前期处理同曲面简化	$O(\max\{2V, T, N, 12F\})$	$O(\max\{2V, T, N, 12F\})$
根据每个三角面片逆时针次序建有向图	$O(3F)$	$O(6F)$
计算顶点度数，区分 regular_point	$O(V\log_2 V)$	$O(V)$
区分有向图中的边界点和内部点	$O(V\log_2 V)$	$O(V\log_2 V)$
建立顶点和纹理、法向量映射	$O(3F)$	$O(V)$
计算新增顶点位置	$O(V\log_2 V)$	$O(V\log_2 V)$
更新原始顶点位置	$O(V\log_2 V)$	$O(V\log_2 V)$
根据细分边集在三角面片中的分布，重构曲面	$O(3F)$	$O(3F)$
释放数据结构并写入模型文件	$O(\max\{2V, T, N, 12F\})$	$O(\max\{2V, T, N, 12F\})$

在以上所有过程的时空复杂度分析表中，可能成为算法瓶颈的部分已经被标出。可以注意到最终整个过程的时间复杂度为 $O(\max\{V\log_2 V, T, N, \alpha F\log_2 F\})$，空间复杂度为 $O(\max\{V\log_2 V, T, N, \alpha F\log_2 F, \beta F\})$，其中$\alpha, \beta$可视为对结果有影响的常数，通常情况下 $1 \le \alpha \le 100$，$1 \le \beta \le 12$。

以上分析过程可以更清晰地表述为，即当模型对应的有（无）向图稠密时，时空复杂度均为模型的三角面片数量决定，时间复杂度为 $O(\alpha F\log_2 F)$，空间复杂度为 $O(\max\{\alpha F\log_2 F, \beta F\})$；即当模型对应的有（无）向图稀疏时，时空复杂度均为模型的顶点、纹理、法向量数量决定，时间复杂度为 $O(\max\{V\log_2 V, T, N\})$，空间复杂度为 $O(\max\{V\log_2 V, T, N\})$。

因此，当模型所涉及顶点数量、纹理数量、法向量数量、三角面片数量较小时（各指标不超过 10k 数量级），整个曲面重构过程的时空复杂度可以理解为是线性的。

5.3 本章小结

本章主要展示了本工作提出的曲面重构相关算法的实现结果，首先展示了不同类型模型重构前后的线框结果和渲染结果对照，其次展示了更多模型的渲染结果以及特殊重构结果。最后加入的性能与复杂度分析也更有力的证明了该算法在时空效率的优越性。

第 6 章 总结与展望

本工作提出的基于二次度量误差的曲面重构算法，良好地将经典二次度量误差在衡量边坍缩代价上的优势与曲面细分过程结合。从而达到了使得曲面顶点分布更合理的效果。

在未来的研究中，可以在如下方面寻求改进。

（一）离线计算方式考虑转变为离线在线计算结合。使得在不大幅影响渲染效率的情况下，渲染观察的每个时刻都可以完成动态的曲面重构。

（二）适应更多种类曲面。引入更加普适性的曲面操作算法，而不限于三角面片构成的曲面。

（三）指标化模型顶点分布。使用数据描述结果，而不限于视觉效果呈现。

参考文献

[1] 孙立镌, 鞠志涛. 渐进插值的 LOOP 曲面细分[J]. 计算机应用研究, 2011, 28(2): 766-768.

[2] Zhou G, Zhang Y, Shan M, et al. An exact bound on regular Loop subdivision surface[C]//2011 IEEE International Conference on Computer Science and Automation Engineering. IEEE, 2011, 2: 416-419.

[3] Arden G. Approximation properties of subdivision surfaces[M]. University of Washington, 2001.

[4] Prautzsch H, Umlauf G. Improved triangular subdivision schemes[C]//Proceedings. Computer Graphics International (Cat. No. 98EX149). IEEE, 1998: 626-632.

[5] Tiantian C, Gang Z. Loop subdivision surface reconstruction with subdivision connectivity[C]//Proceedings 2011 International Conference on Transportation, Mechanical, and Electrical Engineering (TMEE). IEEE, 2011: 2123-2126.

[6] Hu X, Xiong N, Kim T, et al. Mesh Smoothing for Parameterized Body Model with Loop Subdivision Algorithm[C]//2009 International e-Conference on Advanced Science and Technology. IEEE, 2009: 56-59.

[7] Loop C. Triangle mesh subdivision with bounded curvature and the convex hull property[M]//MSR Tech Report MSR-TR-2001–24. 2001.

[8] Yao L, Huang S, Xu H, et al. Quadratic error metric mesh simplification algorithm based on discrete curvature[J]. Mathematical Problems in Engineering, 2015, 2015.

[9] Garland M, Heckbert P S. Simplifying surfaces with color and texture using quadric error metrics[C]//Proceedings Visualization'98 (Cat. No. 98CB36276). IEEE, 1998: 263-269.

[10] Garland M, Heckbert P S. Surface simplification using quadric error metrics[C]//Proceedings of the 24th annual conference on Computer graphics and interactive techniques. 1997: 209-216.

[11] Cohen J, Manocha D, Olano M. Simplifying polygonal models using successive mappings[C]//Proceedings. Visualization'97 (Cat. No. 97CB36155). IEEE, 1997: 395-402.

[12] Xia J C, Varshney A. Dynamic view-dependent simplification for polygonal models[C]//Proceedings of Seventh Annual IEEE Visualization'96. IEEE, 1996: 327-334.

[13] Kalvin A D, Taylor R H. Superfaces: Polygonal mesh simplification with bounded error[J]. IEEE Computer Graphics and Applications, 1996, 16(3): 64-77.

[14] Klein R, Liebich G, Straßer W. Mesh reduction with error control[C]//Proceedings of Seventh Annual IEEE Visualization'96. IEEE, 1996: 311-318.

[15] ji Ban Y, sun Kim H, joon Park C. 3D Mesh Reconstruction from Height map and Post Processing[C]//2018 International Conference on Information and Communication Technology Convergence (ICTC). IEEE, 2018: 1491-1493.

致　　谢

柴门闻犬吠，风雪夜归人。

留一句没有完全读懂的诗句于此，希望在未来某个提灯而来的夜里，想起不知是否正安眠于枕畔的爱人，想起两鬓斑白的父母，想起从师到友的师长，以及...

那段或许可以成为年少轻狂的岁月吧。

本科期间主要研究成果

发表论文：

[1] Richen Liu, Min Gao, **Shunlong Ye**, and Jiang Zhang. IGScript: An Interaction Grammar for Scientific Data Presentation. ACM CHI Conference on Human Factors in Computing Systems (ACM SIGCHI'21), pages x:1-x:14, Yokohama, Japan, May 8-13, 2021. (除导师外二作，CCF A 类会议，长文)

[2] Yeyang Zhou, Yixin Chen, Yimin Chen, **Shunlong Ye**, Mingxin Guo, Ziqi Sha, Heyu Wei, Yanhui Gu, Junsheng Zhou, and Weiguang Qu. EAGLE: An Enhanced Attention-based Strategy by Generating Answers from Learning Questions to a Remote Sensing Image. CICLing: International Conference on Computational Linguistics and Intelligent Text Processing (CICLing'19), pages x:1-x:12, La Rochelle, France, April 7-13, 2019. (四作，CCF B 类会议，长文)

[3] **Shunlong Ye**, Guang Yang, Ziyu Yao, Xueyi Chen, Ting Jin, Genlin Ji, and Richen Liu*. Robust 3-D Field Line Query Based on Data Fusion of Multiple Leap Motions. *IEEE Visualization 2020* (IEEE VIS Poster Paper), Salt Lake City, USA, October 25-30, 2020. (一作，CCF A 类会议，poster)

[4] Min Gao, Hailong Wang, **Shunlong Ye**, Lijun Wang, and Richen Liu*. Interactive Seismic Data Visualization via Virtual Reality Devices. *IEEE Visualization 2020* (IEEE VIS Poster Paper), Salt Lake City, USA, October 25-30, 2020. (三作，CCF A 类会议，poster)

软件著作权：

[1] **叶顺龙**，田易，刘芃杰，王海龙.《在线评测系统的学习过程管理系统》.

[2] 杨光，**叶顺龙**，姚子裕，高敏等.《基于 Leap Motion 的三位轨迹数据交互式查询系统 V1.0》.

[3] 王海龙，**叶顺龙**，田易，刘芃杰.《基于程序结构的查重系统》.

[4] 田易，刘芃杰，**叶顺龙**，王海龙.《在线评测系统的服务器端平台》.

[5] 田易，刘芃杰，**叶顺龙**，王海龙.《在线评测系统的客户端平台》.

完成项目：

[1] **叶顺龙**，田易，刘芃杰，王海龙，宋濂鸿. 大学生创新训练项目. 南京师范大学，2020~2021，0.35 万. （第一主持人，国家级，已优秀结项）