大规模网格简化与
自适应等值面生成算法研究

（申请清华大学工程硕士专业学位论文）

培 养 单 位：　软件学院

工 程 领 域：　软件工程

申　请　人：　侯　　涛

指 导 教 师：　陈　　莉　副教授

二〇一三年六月

大规模网格简化与自适应等值面生成算法研究

侯　涛

Research on Massive Mesh Simplification and Adaptive Isosurfaces Generation

Thesis Submitted to

Tsinghua University

in partial fulfillment of the requirement

for the professional degree of

Master of Engineering

by

Hou Tao

(Software Engineering)

Thesis Supervisor: Associate Professor Chen Li

June, 2013

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：

清华大学拥有在著作权法规定范围内学位论文的使用权，其中包括：（1）已获学位的研究生必须按学校规定提交学位论文，学校可以采用影印、缩印或其他复制手段保存研究生上交的学位论文；（2）为教学和科研目的，学校可以将公开的学位论文作为资料在图书馆、资料室等场所供校内师生阅读，或在校园网上供校内师生浏览部分内容。

本人保证遵守上述规定。

（保密的论文在解密后遵守此规定）

作者签名：_____ 导师签名：_____

日　　期：_____ 日　　期：_____

摘 要

大规模网格模型对简化算法提出了挑战，很多不同的技术都试图在大数据的外存访问效率和网格的简化质量中间取得平衡，有的方法只顾及了一方面，有的能做到两者兼顾。本论文的方法能兼顾大数据访问效率和简化质量，相比同类型的方法，它只需要对网格做一次处理，且能处理输出网格无法载入内存的情况。本论文的方法对网格中顶点而非三角形进行分片，同时所使用的边收缩操作对分片中顶点的拓扑结构改变可以保持在分片之内，边界简化因此变得可能。在一次简化处理之内，简化的网格在分片边界处不会出现密度不一致的情况，因此不需再做处理。经测试表明，本论文的算法能够在较短的时间内提供和内存简化算法相比拟的简化质量。

有些体数据生成的等值面过大，超过了内存的装载能力。如果将这些网格完全生成出来再使用 out-of-core 简化，会使得整个处理过程耗时过长。本论文提出了在内存中生成一部分等值面就进行简化的方法。以往同类型的方法因为需要不断输出简化的网格使简化质量打了折扣，且它们对顶点何时退出生成边界的判断没有充分考虑体数据本身的特性。本论文的方法在内存中保持了一个全局的简化操作最优队列，尽量保证他们能够得到更多比较；并结合体数据生成的特点，来指导算法获知顶点何时离开生成边界，使得简化过程内存占用较小且具有可预测性。本论文还在生成过程中对等值面的拓扑关系进行了重建。经测试，本论文的算法能够在较短时间内产生类似于高质量内存简化的结果。

直接对体数据进行自适应的等值面生成能够对等值面进行自动简化，但大部分自适应等值面生成算法无法保证生成的网格流形且无自交。有一种基于对体数据八叉树单形分割的串行算法能够解决这个问题。本论文在原先串行单形分割算法的基础上，对分割方法做了简化以减少体数据函数近似产生的误差，并对它做了并行的改进。本论文算法的主要贡献是提供了一套完整的基于 GPU 并行的八叉树建立与最小边查找方法。它自顶向下分层建立八叉树，并在查找最小边的时候，将当前一层边的分割结果与当前一层八叉树节点所展开的新边、当前一层面分割所引入的新边一起加入到下一层待分割的边中。在实际测试中，本论文的算法能够产生流形且无自交的自适应等值面，并在处理大规模数据时获得了相对于串行算法较高的加速比。

关键词：大规模网格；网格分片；流式处理；单形分割；并行八叉树

Abstract

In this paper, we first propose a simplification algorithm for massive meshes based on cutting the mesh into pieces by partition the vertices instead of the triangles. The trianlge boundaries generated by the vertex-seperating approach need not be preserved during the simplification of each piece. The edge collapse operator we use can keep the topology change of each mesh piece independent from the others automatically, which makes it possible for the bounadaries to be simplified. Such an algorithm needs only one simplification process over the whole mesh without further processing. When stitching the pieces, we only keep the boundary vertices in memory without caching the whole simplified mesh, so our algorithm can generate large simplfied meshes which cannot be loaded into the main memory.

Second, we propose an on-the-fly simplification algorithm for massive isosurfaces which the main memory is not able to hold if fully generated. We use the edge collapse operator and keep the generation boundary unchanged. The boundary marches through the whole mesh as isosurfaces generated. We reconstruct the topology of the original Marching Cubes surfaces for in-core simplification algorithm. To determine when the primitives in the mesh can be modified, we classify the vertices generated by Marching Cubes algorithm and finalize them based on different rules. When a vertex is finalized, it leaves the generation boundary and becomes collapsable. We keep an collapsalbe edge priority queue all along the simplification process and try to make more operators to compare. Thus, simplification quality can be enhanced.

Finally, we propose an adaptive isosurface generation algorithm based on simplicial complex partition of the volume data octree, which is parallel on GPU. This approach can generate manifold and intersection-free isosurfaces without any crack. Our main contribution is that we provide a group of parallel algorithms for building the octrees and traversing the minimal edges on GPU. We build the octree from top to the bottom and calculate the dual vertices, error incurred from expanding the nodes and configuration of child nodes in parallel for each layer. We use a parallel scan to calculate the offset of child nodes and keep a starting offset as well as a count of child nodes for each parent node as pointers to its children. In our octree data structures, all

nodes are stored layer by layer in the GPU memory. In each layer, nodes with identical parent are stored continuously. When traversing the minimal edges, we caculate the splitting number of each edge corresponding to a layer and use a parellel array partition to separate the minimal edges and splittable edges. We also add the new edges incurred from expanding the nodes to the need-to-split edge array for the next layer. We form tetrahedron for each minimal edge and generate isosurfaces from the tertrahedron.

Key Words: Massive Mesh; Mesh Cutting; Streaming Processing; Simplicial Partition; Parallel Octree

目 录

第 1 章 绪论

1.1 研究背景

在图形学应用中，有很多产生网格的途径，比如从点云、CSG 模型、体数据等进行生成。随着计算机技术的不断发展，这些方法生成的网格越来越大，有些甚至超出了普通 PC 机内存的装载能力。它们中间比较著名的有斯坦福[1]和 IBM 实验室[2]对米开朗基罗的雕塑进行扫描与重建所得到的模型。这些模型能够提供非常精细的细节，但普通 PC 机内存无法装载和渲染，因此需要对这些模型进行简化。本文针对以上问题，提出了三种不同应用场景下的简化算法。

大规模网格分片简化是针对通过不同途径生成的超过了内存装载能力的网格，使用基于外存的技术进行简化的方法。大规模网格对简化算法本身提出了很大的挑战，因为常规的内存简化算法无法直接应用于大规模网格。为了解决这个问题，有不同的技术被提出，试图平衡大数据外存文件的访问效率和网格的简化质量。

大规模等值面的即时简化是针对大规模体数据生成的等值面过大、甚至超过了内存的装载能力、难以处理，而提出的生成一部分就进行简化的方法。它不需要生成全部等值面，因此避免了耗时的基于外存的等值面生成和 out-of-core 简化。

基于 GPU 的自适应等值面生成是对 CSG 模型或体数据，在生成等值面的阶段，就考虑模型本身的复杂度，进行具有自动简化功能的自适应生成。

本章接下来将对网格简化与等值面自适应生成做一个简短的综述。由于内存简化算法是大规模简化算法的基础，因此本章将首先对内存网格简化算法做以综述。

1.2 内存网格简化算法

1.2.1 顶点聚类

Jarek R. Rossignac 和 Paul Borrel [3]提出了一种对网格进行均匀重采样的方法。这种方法可以处理任何形式的网格数据，不论网格退化情况有多么严重。它可以处理输入网格是很多个不相邻的离散三角面片这种情况，而其他大部分算法都无法做到这一点。这种方法比一般保持原网格拓扑结构的算法速度都要快，但它的缺点是破坏了网格的拓扑结构，并且不保证输出网格在一定的误差范围之内，所以它的简化效果是比较粗糙的。

这个算法一共包括四个步骤：顶点评定，顶点聚类，顶点合成和面片消除。

顶点评定是对顶点根据一定的评判标准给出一个重要性的权重。权重的度量主要依靠两个标准。其中一个标准是顶点是否有可能在模型的较突出的轮廓上。作者通过计算与某个顶点相邻接的所有边的二面角的最大值的倒数来获得。另一个标准是顶点是否邻接一个面积比较大的面片。这种标准通过计算顶点相邻的所有边的最大长度来计算。作者然后将两个标准进行线性组合得到顶点的权重。

顶点聚类的过程是首先将模型的包围盒均匀地分成几个小立方体。然后处理每一个顶点，将其分配到其所在的立方体之中。

顶点合成的过程是将上一部所得的聚类中的所有顶点合并为某个权重最高的顶点。

面片消除的过程对每个三角形进行处理，以决定其是否在输出网格中保留。判断标准是，如果三角形的三个顶点分属于不同聚类，则留下，否则被剔除。对于被剔除的面，若有两个顶点在同一聚类中，则这个三角形被退化为一条线；若三个顶点均在同一聚类中，则退化为一个顶点。

这个方法中，包围盒的分割尺寸是一个可变参数，算法通过包围盒分割的精度控制输出网格的精细程度。

Kok-Lim Low 和 Tiow-Seng Tan[4]对 Rossignac-Borrel 的聚类算法做了深入分析并对原方法的某些缺陷做了改进。他们将原始的统一空间分割聚类改为一种“浮动体”聚类法。算法首先对顶点进行重要性的排序，然后在最重要的点上放置一个用户指定大小的浮动体。所有在这个体之内的点被收缩为一个代表顶点，对于三角形的处理则与 Rossignac 的方法相同。在这样的一次操作之后，剩下的顶点中最重要的被选择成为下一个体的中心，以此类推。浮动体的聚类消除了原统一聚类中对网格方向的敏感性，而且浮动体的大小对最终简化质量的影响不像原来统一聚类中聚类立方体的大小的选择对最终网格质量的影响那么大。

本文同时对顶点权重的计算方法做了优化。设 θ 为顶点邻接的所有边之间最大的夹角，Rossignac 使用 1/θ 估计顶点在模型特征轮廓上的可能性，而本文使用了 cos(θ/2)来估计，并指出这种方法能给出更好的结果。

Low 和 Tan 的另一创新是在简化网格的同时考虑了聚类对最终渲染结果的贡献问题。他们度量每个聚类在最终渲染出的图像中所占的像素数。这种改进同时也是 Luebke-Erikson 的视角依赖[5]方法的基础。

1.2.2 区域合并

Paul Hinker 和 Charles Hansen [6]提出了一种将某些被过多面片填充

（over-tessellated）的平坦网格区域进行合并与消除进而达到简化目的的方法。将这些平坦的区域以较少数量的网格进行替换并不会显著影响网格的质量。这个方法的缺点是大部分网格模型并不存在大片的平坦区域，对于那些特征分布比较密集的网格，这种方法效果不佳。

这个算法首先建立接近共面的三角形的集合，共面的度量标准为判断法向的相似度。算法首先算出集合面片的平均法向，然后对于每个面片，判断面片的法向与集合的平均法向的相似度，如果两个法向的夹角在某一个阈值之内，则将这个三角形加入到集合中并更新集合的平均法向。在这个面聚合的过程终止时，所有的集合内的面片都是几乎共面的。

接下来，算法创建集合的边界。对于每一个共面集合，算法对所有三角形的边进行排序，并剔除那些重复的边，剩下就是处在边界上的边。

随后算法将所有处在边界上的边通过查找共同端点连接起来。

算法然后将每个集合的边界进行三角化。三角化的过程需要注意共面集合中的孔洞区域。三角化采用贪心策略，从凹进去最多的点开始按照三种方式遍历边界点，只要三角形不违反某种限制就接受这个三角形。虽然贪心算法生成的三角网格的质量不高，但它的效率和速度都很高，而且结果可以接受。

算法最后使用简化后的集合代替原来的集合。

A. Kalvin 和 R. Taylor 提出了另一种区域合并算法[7]，也可以称为区域生长。这个算法包含三个步骤：面集创建，边界拉直，和面集三角化。面集合并的过程选择一个种子面然后不断尝试将其邻接的面加入到面的集合中。如果一个候选面符合拓扑、三角形质量和误差阈值的要求，算法就将其加入到面集中。当一个面集无法继续生长的时候，算法选择另一个种子面创建一个新的面集。面集的创建过程一直持续直到所有面都归属于某个集合。

边界拉直是将所有邻接于两个面集的顶点消除，进而使得面集的边界成为直线。如果拉直之后的边界所产生的误差大于某个误差阈值，则算法会分割这个边界直到满足阈值的限制。

算法随后对每个面集做三角化。三角化首先将多边形分割为多个单调多边形，然后再对这个多边形中间插入一个顶点，组成星形三角环，从而完成三角化。

DeFloriani 等提出了一种将网格中邻接的三角形子集替代为边界相同的简化三角形集合的方法[8, 9]。这种名叫“多样三角化”（multi-triangulation）的局部操作具有最好的通用性。它可以转化为边收缩和顶点移除，同时还能转化为边交换。它同时证明了，边交换和边收缩能够生成任意拓扑结构的网格，因此包含这两种操作的多样三角化操作也能够产生任意网格。

1.2.3 迭代式消除

迭代式面消除

Bernd Hamann[10]提出了一种不断迭代地对面进行消除的方法，它简化的依据是三角形在局部区域的曲率。算法在高曲率区域做较少的简化而在低曲率区域做较多的简化。

这个算法首先通过点内部邻接三角形的角度估算出顶点的曲率，然后对三角形中顶点的曲率做平均得到三角形的权重。算法每次找出具有最小权重的三角形并将其消除。如果消除操作会破坏原先网格的拓扑结构，则不会执行这个操作。算法使用最小平方法对被删除三角形与邻接面片之间的局部曲率求得极值来得出点的位置。然后算法将此三角形的所有邻接三角形的边所组成的环（或者非闭合的线段）与最优顶点组成一个新的星形的拓扑结构。算法使用边交换来优化三角形的质量。每次迭代之后，算法会重新计算被删除三角形周围的三角面片的权重。算法迭代直到达到用户指定的简化个数，或者算法因为拓扑结构的限制无法继续简化。论文[11]中也有类似的实现，只是它使用了二次误差度量（quadrice error metrics）。

顶点移除

William J. Schroeder 等提出了一种顶点移除方法[12]。这种算法对“移动立方体”（Marching Cubes）[13]算法所产生的等值面移除的效果非常好，因此通常作为移动立方体的后处理工具。顶点移除所产生的网格中的顶点是原网格顶点的一个子集。

算法过程是，首先用户指定一个距离误差的阈值，然后算法每次移除一个在这个阈值之内的顶点。在选择顶点的过程中，算法根据顶点的几何信息与拓扑结构将其分类为：简单点，非简单点，边界点，内部点，角落点，并只移除那些不破坏网格局部拓扑结构的顶点（也就是除非简单点以外的点）。

简单点四周的三角形形成一个环，并且每一条邻接边邻接两个面。非简单点或者周围的三角形不形成环状，或者某些邻接边的邻接面片的个数不等于两个。边界点为周围的三角形形成一个半环的简单点。而内部点和角落点是特殊的简单点。算法定义二面角大于某个阈值的边为特征边。内部点为邻接边中有两个边为特征边的点。角落点为临界边中有一个、三个或更多特征边的顶点。

针对简单点，算法计算顶点到所有邻接顶点的平均平面的距离，如果距离小于某个用户指定的阈值，则删除这个顶点。而对于内部点和边界点，算法使用顶点到边界边或者特征边的距离来度量误差。

移除顶点之后，被移除顶点的邻接顶点会组成一个环，算法使用递归的“环分割”方法来对其进行三角化。每次分割找到两个不相邻的顶点作为分割边。算

法首先判断分割是否会破坏网格的局部拓扑。判断方法是定义一个穿过被分割边且垂直于顶点均平面的分割面，如果被分割的两个环上的点处在面的两侧，则可执行分割，否则不执行分割。如果某个环中不存在这样的可分割顶点，则算法将不执行此次点消除操作。随后算法对不同的分割边计算分割质量，试图使获得的三角形具有最好的“方向比率”（aspect ratio）。这个度量标准被定义为环中顶点到分割平面的最小距离除以分割边的长度。具有最好分割质量的分割边会被选择。分割的过程不断递归直到所有环均为三角形。

当没有顶点符合消除标准的时候，算法停止。

迭代式边收缩

Ronfard and Rossignac[14]提出了一种基于贪心的边收缩算法。他们度量误差所使用的方法是顶点到所有邻接平面的最大距离。当边被收缩的时候，两个顶点的邻接平面集合合并为一个集合。因此，当有越来越多的边被收缩至同一点的时候，邻接平面集合会变得越来越大。这个误差度量可以表示为：

$$E_v = \max_{p \in planes(v)} (p \cdot v)^2 \qquad (1\text{-}1)$$

迭代式边收缩是一种非常常用的简化方法，它具有简化质量高，速度快，能简化模型类型较多等特点。其他具有更好效果的迭代式边收缩算法可以参见[15-18]。

1.2.4 随机重采样

网格优化

网格优化[19]由 Hugues Hoppe 等人提出，它能够在提供一个原始网格和一些散点的情况下，重建散点的拓扑结构并使得散点的位置尽量贴合原始网格。网格优化可以同时适用于表面重建和网格简化。在应用于简化时，需要先对网格的表面进行一次随机均匀采样，得到网格优化的原始散点。在随机采样散点的时候，需要对边界重新进行一次采样以达到边界保持的效果。

由于网格优化的目是重建和优化散点的拓扑结构以及优化散点的位置以让输出网格能最优地贴合原始网格，因此，作者定义了一个能量方程，作为整个算法的核心部分。能量方程如下：

$$E(K,V) = E_{dist}(K,V) + E_{rep}(K) + E_{spring}(K,V) \qquad (1\text{-}2)$$

其中 K 为网格的拓扑结构而 V 为网格顶点的位置。能量函数的第一项 E_{dist} 是输出网格中所有点与原始网格的距离平方之和。E_{rep} 是一个系数 c_{rep} 乘以输出网格中的顶点数。E_{dist} 用来保证输出网格与原始网格的近似程度，而 E_{rep} 用来平衡网格的简化程度。因此，优化的过程允许顶点被加入或者被删除。当顶点被加入到网

格的时候，距离项 E_{dist} 有可能会减少，E_{rep} 可能会增加。顶点从网格中删除则反之。系数 c_{rep} 用来平衡两个项之间的权重，通过用户指定的这个系数，算法可以在顶点数目和简化质量之间取得一个平衡。作者同时提出，只使用前两项的能量函数所产生的结果往往并不理想，在某些未被采样的地方会产生很严重的锯齿现象。这说明原始的能量函数 $E_{dist}+E_{rep}$ 可能并不存在一个极小值。因而作者引入了一个弹力项 E_{spring}，来使得网格达到更理想的质量：

$$E_{spring}(K,V)=\sum_{\{j,k\in K\}}\kappa\|v_j-v_k\|^2 \qquad (1\text{-}3)$$

需要说明的是 E_{spring} 项并不是为了消除网格中所有曲率较大的区域(即网格中二面角较大的边)。因为这样的边同时很有可能也是特征边。作者将 E_{spring} 定义为一种规约原能量函数使其能正确找到一个最小极值的影响因子。当优化的过程不断收敛的时候，E_{spring} 的值会不断缩小。

算法的过程就是通过两层嵌套循环来分别优化网格的拓扑结构和顶点位置。外层循环优化网格的拓扑结构，为此，文中定义了几种改变网格拓扑结构的操作：边收缩，边分裂，边交换。其中边收缩减少网格的顶点，边分裂增加网格的顶点，而边交换会优化三角形的“方向比率”（aspect ratio）。作者还指出，这三种操作可以产生所有不同的网格拓扑结构。算法在优化的过程中随机选取优化操作并试探其是否会使能量函数减小。算法的内层嵌套优化顶点的位置，通过这样的两层嵌套，算法就能达到简化并优化网格的目的。

1.3 大规模网格简化算法

1.3.1 分片简化

Hoppe 最早提出了分片简化[20]的思想，他将其基于内存的“渐进网格”（progressive meshes）数据结构扩展到大规模高度场数据的简化中。作者首先将原始的高度场网格数据按照其坐标沿 x、y 轴平均分割成几个矩形的分片，并保证这几个分片能放入内存中。对三角形分片之后，由于某些点和边在边界之上，故不能做简化，只能保持。算法随后对分片内部的点采用边收缩将每个分片简化到一定程度。由于分片边界被保留，使得整体网格密度变得不均匀，因此作者再将分片简化之后的网格（保证这个网格一定能装载进内存）重新进行一次内存简化，进而使最终简化的效果令人满意。

虽然 Hoppe 的算法能够产生很好的效果，但它只能应用于高度场。Prince 将

Hoppe 的算法扩展到了任意网格的情况[21]。Prince 的算法使用一个统一的包围盒分割来对网格中的三角面片进行分割，它计算三角形顶点的平均点并判断它落在哪一个立方体内。而对于将网格分片组装的过程，Hoppe 曾经提出了一种基于 Voronoi 重建的分割与组装方法。这种方法过于复杂且对于原始网格的拓扑结构有限制，因此 Prince 给出了一种更加简单且适用于输入网格是任意拓扑结构的组装算法。组装算法的过程是，每次处理一个分片时，将它所有顶点坐标插入一个哈希表。根据这个哈希表，可以判断点是否重复。重复的顶点可能是分片之内同一个顶点所邻接的不同三角形造成的，也可能是不同分片之中的相同且未简化的边界点。对于新增的顶点，算法给其赋予一个新的索引值。

Bernardini 等人[2]在其重建大型三维网格模型的文章中提到了对网格进行简化的算法。虽然与 Hoppe 同样使用了分片，但其后续对分片边界的处理方式不同。在对整个模型的分片进行一次简化后，作者对模型使用不同的分割边界再做一次简化。这个过程可以不断迭代直到模型的大小能够装进内存。

Brodsky 提出了名为 PR-Simp 的基于任务并行的分片简化算法[22]。PR-Simp 算法使用主/从分布式结构，主机节点负责协调从属节点的行为，而从属节点进行计算。Brodsky 的算法同样基于网格分片，它使用类似于 Prince[21]的空间分割方法，通过在每个从属机器中过滤属于本分片的顶点与面片从而对网格进行分片。同时，PR-Simp 算法创建跨越分片边界（即不是所有顶点都属于这个分片）的面片。这些跨越边界的面片是用来将分片重新组装起来的。算法使用分治策略将分片的部分组合起来。相互邻接节点两个两个结对将边界面片进行合并，组成 n/2 个分片（假设原先有 n 个面片）。然后算法再对这 n/2 个分片进行结对合并组成 n/4 个分片。算法迭代 log(*n*)次直到所有分片都组装在了一起。简化后的网格再由主结点写入磁盘。Brodsky 同时指出，对面片进行分片会影响网格局部操作的全局执行顺序（比如贪心算法中迭代的对当前误差最小的局部操作进行执行的顺序），这是分片算法的一大缺陷。Brodsky 指出如[6, 7]等区域增长算法更适宜于数据分割，因为其数据访问的形式是按照空间顺序的，而某些迭代式消除算法[15, 17-19]则不适宜数据分割。而 PR-Simp 所使用的 R-Simp[23]算法由于基于空间分割，其数据访问形式遵照空间顺序，因此适宜做数据分割。Brodsky 还讨论了分配每个分片中输出顶点数量的不同方法，比如根据分片的原始顶点数目进行分配以及根据每个分片的复杂度进行分配的方法。

1.3.2 使用外部数据结构

Paolo Cignoni 等[24]提出了一种基于八叉树的外存网格数据结构（OEMM）来

对大规模网格进行简化。它提供对复杂大型网格的外存数据管理功能，支持动态地将需要的网格数据从外存装载入内存，并能够维护在对网格局部区域进行修改时的数据一致性。有了这样的数据结构，很多对大型网格的复杂操作（如简化，细节保持，网格编辑，可视化以及网格分析）等就可以在普通 PC 机上实现，而且其时间耗费是可以忍受的。OEMM 数据结构不仅仅是一个基于空间分割和内外存数据交换的方法。它所具有的主要特性包括：(1).对输入的非索引形式的三角形面片（也就是每个三角形用三个不带索引的顶点坐标表示）建立一个全局的网格索引；(2).通过对载入内存的数据进行实时索引更新，支持任何对网格的局部载入、更新以及写回操作。这种方法可以使得任何载入内存的顶点和三角形以索引列表的形式呈现。OEMM 的数据分割使用基于八叉树的空间分割，跨越不同八叉树结点的元素会在数据结构建立的过程中被特殊考虑，邻接不同结点的顶点使用一致的索引值，而边界元素会被指定到一个特殊的结点中。由于载入内存的网格中某些内部顶点可能是边界，而算法可能无法对这些顶点进行特殊考虑，因此 OEMM 使用了对数据加标签的方式来对当前网格中的边界元素进行检测和管理。这同时使很多图形算法的外存版本实现变得简单，因为数据结构底层的空间分割细节被隐藏了。

OEMM 的叶子结点指向一个外存的数据块。数据块中包含所有在叶子结点包围盒之内的顶点。由于有些顶点的邻接顶点在其他结点之内，结点数据块还会存储少量外部顶点的信息。部分顶点在结点包围盒之内的三角形会存储在顶点所在叶子结点集合中的最低层次的叶子结点中。全部在某个结点之内的三角形会存储在相应结点之中。

算法假设输入的网格为大规模的未索引的三角面片集合。如果输入网格已经是索引的格式，则后续的很多处理将可以省略或化简。对于大部分大规模网格来说，其表示形式通常为几个独立索引的网格分片的集合，因此就必须重新对其所有分片的顶点进行全局索引。OEMM 数据结构的建立过程主要分为两步：

(1). 首先对输入网格建立一个原始的基于外存的数据结构。原始的 OEMM 数据结构不带缓存。每个八叉树的根节点保存了所有网格中至少有一个顶点落在这个根节点的包围盒中的三角形。那些落在不同根节点中的三角形在每个结点中都有拷贝。

(2). 遍历原始的 OEMM 数据结构并建立其索引，即对每个顶点指定一个唯一的全局 id。在这个过程的最后阶段，八叉树中的结点会根据之前的定义存储相应的顶点与三角形。

作者随后对建立好的 OEMM 数据结构使用边收缩进行简化。作者在做简化的

时候并没有使用一个全局的堆，因此算法并没有保持内存边收缩算法的全局操作顺序。算法对网格依照其空间顺序进行遍历并只将载入网格的边插入堆中进行收缩。虽然这破坏了全局的简化顺序，但最终算法依然能达到较理想的简化效果。

El-Sana 和 Chiang[25]提出了一种称为 spanned mesh 的数据结构。算法以提供明确拓扑结构的索引网格为输入，将网格中的所有边及其邻接三角形放入一个外部堆中。堆中元素的比较标准为边的长度。之所以使用边的长度来作为标准是因为诸如二次误差度量（Quadric Error Metrics）等方法度量方式更加复杂，初始化堆和收缩顶点之后更新堆中结点的过程中会带来大量的操作，这种频繁的数据交换很难在外部堆中实现。根据可使用内存的大小，算法每次从堆中取出 k 个元素，并重建由载入的三角形所组成的网格。然后，所有那些邻接三角形被载入内存的边会被收缩掉。当可以载入内存的边数量较大的时候，这些边可以组成相对较大的邻接区域，算法可以达到一个较好的结果。在处理大规模网格的时候，由于较短的边可能是均匀的分布在不同区域中，载入内存的网格部分可能包含很多很小的非邻接区域，因此算法可能频繁载入和写出边数据，出现内外存交换的抖动现象。这个算法的一个优势是不会破坏局部简化操作的顺序，即外存简化与基于内存简化算法所产生的局部操作的顺序一致。这个问题在 Brodsky 的论文中也有探讨[22]。

1.3.3 网格批处理

Peter Lindstrom[26]提出了一种对三角面片进行批处理的算法。这种被称为 OOC 顶点聚类的方法与早期 Rossignac[3]提出的网格聚类方法类似。算法在最初获得网格的包围盒。这个包围盒可以由用户指定也可以通过一次扫描算出，而作者则假设包围盒已经给定。在已知算法的包围盒之后，算法根据用户的输入对包围盒的 x、y、z 方向平均分成几个等份，因此组成一个大小相同的立方体集合。然后对于每个三角形的顶点，判断其落在那个立方体之内，并将其加入到立方体之中。作者对原顶点聚类的一个优化是在计算每个聚类中的代表顶点的时候，使用类似论文[18]中的方法，保存落在这个聚类中的所有的面片的二次误差矩阵的和。由于二次误差矩阵乘以某个顶点代表的是这个顶点到集合中所有平面距离的平方和，所以对这个矩阵求最小值既能得到一个最优的代表顶点。在计算代表顶点的过程中，算法对[18]中的方法进行了优化。在误差矩阵无法得出唯一解的时候，算法试图对矩阵做奇异值分解：

$$\mathbf{x} = \hat{\mathbf{x}} + \mathbf{V}\boldsymbol{\Sigma}^{+}\mathbf{U}^{\mathrm{T}}(\mathbf{b} - \mathbf{A}\hat{\mathbf{x}}) \qquad (1\text{-}4)$$

这样做的好处能够求出一个离某个指定点（如聚类立方体的中心点）比较近且能保证矩阵与向量的乘积较小[27]的点。

作者对每个聚类在一个哈希表中保存一个字段，关键字为聚类的坐标，值为顶点个数、二次误差矩阵等。算法对三角形的处理过程与论文[3]相同，把有两个或两个以上顶点在同一个聚类的三角形剔除。

Peter Lindstrom 和 Cláudio Silva[28]针对论文[26]中的方法只能处理输出网格能够载入内存中的问题，提出了基于外部排序和外存临时文件的方法，将网格批处理算法扩展到了能够处理输出网格无法载入内存的情况。

1.3.4 流式简化

流式简化基于对网格移动工作集的处理。移动工作集是指大规模网格载入内存中的一部分数据，这些数据会随着处理的不断进行而向网格的其他部分移动。流式算法跟随移动工作集的移动而一部分一部分地处理网格数据。由于移动工作集是无缝地逐步扫描过整个网格，整个过程就好像“流”过网格一样，因此被称为流式算法。流式算法需要解决的一个问题是何时能够对网格的元素（顶点或者边等）进行消除或者修改。流式算法载入内存的是网格的一部分，因此产生了内存部分与外存部分的边界。就像分片简化中的保持边界不简化一样，流式简化也需要保持这些边界不被简化，以防止出现拓扑结构不一致性的情况。

使用流式算法进行简化最早由论文[29]提出，它假设面片为流形的，并对那些已经读进来全部两个邻接三角面片的边进行收缩。为了加快处理速度，它并没有使用全局的堆结构进行操作的选取，而是随机选取一定数量的候选操作并选择误差最小的。虽然抛弃了堆结构会一定程度上影响简化的质量，但边收缩本身具有高质量的特性，且随机取最优依然在一定程度上做到了特征保持，因此算法的简化质量相对可观。流式简化算法保持一个已读入部分与未读入部分的边界和未输出部分与已输出部分的边界，并保持边界不被简化。

Isenburg 的论文[30]对流式处理做了一个全面的总结并提出了顶点“终结”的概念，完备地解决了何时应该对元素进行修改的问题。它对索引网格格式的输入计算顶点与三角形之间的互相引用关系，并根据这个引用关系判断何时某些元素不会被其他元素所引用。它打破了原索引网格格式“先顶点列表后三角形列表”的形式，提出了“流式网格”的格式，将顶点和三角形的信息在网格数据文件中穿插起来，并显式标定何时顶点的所有邻接三角形都已经被读入。更重要的是，它对流式网格所产生的缓存大小做了深入的讨论并给出了如何优化网格的存储格式（Mesh Layout）以减小网格处理过程中的缓存大小的方案。

1.3.5 小结

本小节综述了四种不同类型的大规模网格简化算法。本文所提出的方法借鉴了其中一些思想，选择了一些效率较高、简化效果较好的方法做出了改进并将他们所使用的技术应用到特定的环境之中。

本文的大规模网格分片简化算法借鉴了原始分片简化算法的一些思想，但在分片方法与网格合并方式上做了改进。它和论文[28]所述的方法以及流式简化算法一样，能够处理输出网格不能够载入内存的情况。

本文所提出的大规模等值面即时简化算法借鉴了流式简化的思想。它简化网格的方式与论文[29]类似，但并不输出简化网格，因此没有输出边界；它对操作的选取没有使用随机选择取最优的方式，而是保存了全局的最优队列。另外，本文方法另一个重要贡献是借鉴了“流式网格”[30]中对索引网格形式顶点的终结信息的判断，在体数据生成过程中根据自身的特性判断终结信息，从而使得简化过程内存占用较小且具有可预测性。

1.4 自适应等值面生成算法

Raj Shekhar 等[31]提出了一种基于八叉树的 Marching Cubes 生成算法，它自底向下地根据某种标准合并子结点并对合并的子结点所生成的面的形状做以严格的限制，以防止拓扑结构的不一致性。它对自适应八叉树中的每一个大小不一的结点使用 Marching Cubes 生成等值面，但这会在不同层次八叉树结点的边界处产生拓扑结构不一致性和碎裂（Crack）。作者解决问题的方法是忽略拓扑结构的不一致，而试图将碎裂的部分粘合到一块，即让边界处由多个较小结点生成的较细粒度的边向较大的结点生成的边移动而使得碎裂消失，这被称为碎裂修补（Crack Patching）。

虽然碎裂修补能够一定程度上解决等值面的瑕疵，但是却无法从根本上解决问题，产生的等值面依然是非流形的。针对类似于 Marching Cubes 的原始曲面（Primal Contouring）生成方法（即在每个体数据网格的边上生成一个等值面中的点，在体数据网格的面上生成边，再在单元体素内根据体数据点的配置情况生成面）所产生的这些问题，Tao Ju 等[32]提出了一种对偶曲面（Dual Contouring）生成方法。它对体数据中每个与等值面的数据值大小关系不同的立方体生成一个对偶点，并将端点与等值面数据值大小关系不同的边所邻接的四个立方体的对偶点连接成一个四边形。对偶曲面方法能够避免碎裂的出现，但它会在某些等值面的局部产生非流形的结构或自交的情况。为此，作者曾经提出了能够避免非流形[33]或

者自交[34]的对偶曲面生成算法，但却无法做到两者兼得。

Scott Schaefer[35]等提出了一种对原始体数据网格生成对偶网格进而生成等值面的方法。它对每个自适应八叉树中的网格结点生成一个对偶点，并将八叉树中每个顶点所邻接的八个或少于八个的网格结点所对应的对偶点连接起来（连接的方式是将每个所在结点相邻接的对偶点相连），生成对偶网格。这些对偶网格中有的是六面体，有的少于六个面。它对这些等于或少于六个面的多面体使用 Marching Cubes 生成等值面。这种方法能够产生流形的等值面，但由于对偶多面体可能退化或者变形，使得产生的等值面有自交的情况。

Josiah Manson 等[36]提出了一种将八叉树做单形分割生成四面体，进而对四面体生成等值面的方法。它对自适应八叉树中所有的边、面和立方体生成对偶点，并将这些对偶点按照他们单形的维度依次连接，进而生成对原始立方体的四面体（三维单形）分割。要注意的是，生成的对偶点需要限制在所在单形之内，以防止四面体的变形或退化产生自交的等值面。

对八叉树做单形分割的方法能够产生流形且无自交的等值面，质量较高，但对偶点的求取会带来误差。另外，对边求对偶点不影响四面体网格的流形性质，却会加大这种误差。本论文所提出的自适应等值面生成算法在原先串行单形分割算法[36]的基础上，对分割方法做了简化以减少体数据函数近似所产生的误差。同时，本论文对它做了并行的改进，充分地利用了 GPU 的并行粒度，使得运行速度取得大幅度增长。

1.5 论文的主要内容及章节安排

本文第一章介绍了本论文的研究背景、所依托的工程项目、内存网格简化算法的研究现状、处理大规模网格的简化算法所使用的技术和自适应的带有自动简化功能的等值面生成算法的现存解决方案。

第二章提出了一种能够在保持较高质量简化结果的情况下需要更少处理次数的大规模网格简化算法。第三章针对大规模体数据生成的等值面过大而无法装入内存的情况，提出了一种生成一部分网格即进行简化的算法。第四章提出了一种完全并行的能够生成流形且无自交等值面的自适应算法。在叙述算法的时候，我们分别阐述了算法的设计思想和一些需要特别说明的实现细节，并对结果进行了分析。

第五章是对本文工作的总结，同时提出了本文成果的不足和对未来改进的建议。

第 2 章　基于点分片的大规模网格简化

使用分片进行大规模网格简化具有简化质量好、操作较为简单等优势。但现存网格分片算法有一些劣势：

(1). 大部分算法由于分片简化的过程中不能对边界进行简化，因此必须对合并后的网格再进行一次处理。

(2). 有的方法在合并网格的过程中需要将所有简化后分片的顶点都载入内存，对简化后网格的大小做出了限制。

相对于原分片简化算法的缺点，本章节算法的主要特点是：

(1). 由于使用了对点而非对三角形进行分割的方法，简化过程不需要对分片的边界进行保持，分片合并后不会产生严重的分片边界与内部顶点的密度不统一，因此不需要在分片合并后再对网格进行处理。

(2). 由于算法在对分片进行合并的时候只需要保持输出网格的一部分信息，因此算法对输出网格的大小没有限制。算法可以输出一个大于内存装载能力的网格。

两种分片方法的对比如图 2.1 所示。其中(a)为原始网格；(b)(c)为对三角形进行分片的简化过程，由于边界不能被简化，导致了密度不统一；(d)(e)为对点进行分片的简化过程，由于所有点均可以被简化，因此没有产生密度不统一的情况。

2.1　边收缩操作对边界拓扑结构的自动保持

在简化分片的时候，我们选择了边收缩作为简化操作。边收缩操作最早在 Hoppe 的论文[19]中被提出。边收缩与边分裂是一对互逆的操作（如图 2.2），它将一条边的两个端点合并为一个新的顶点，并消除邻接这条边的三角形。

如图 2.1-(a)、2.1-(b)所示，在使用对点进行分片的大规模网格简化算法中，使用了边收缩的内存简化不会产生拓扑结构的不一致性。这是因为，边收缩对分片的拓扑结构改变可以保持在分片之内，而不影响其他分片。这保证了分片之间简化的独立性，使得边界被简化变得可能。

如图 2.3 所示，AB 为一条处在边界上的可被收缩的边，CD 为另一边界上的可收缩的边。在收缩操作中，A 与 B 收缩为点 A’，C 与 D 收缩为点 C’。两次不同分片中的收缩操作所影响的不属于本分片的三角形仅仅为处在分片边界上的三角形（图中的亮蓝色三角形区域）。在后续的处理中，算法将移除这些退化的边界三角形。需要注意的是，在边被收缩的时候，需要检查是否有代表顶点跨越了边界

（如图 2.4 所示）。这个问题可以通过判断收缩后的点是否落在了本分片的包围盒之内来解决。如果出现了越界的情况，则可以使用两个端点的中点作为代表顶点。

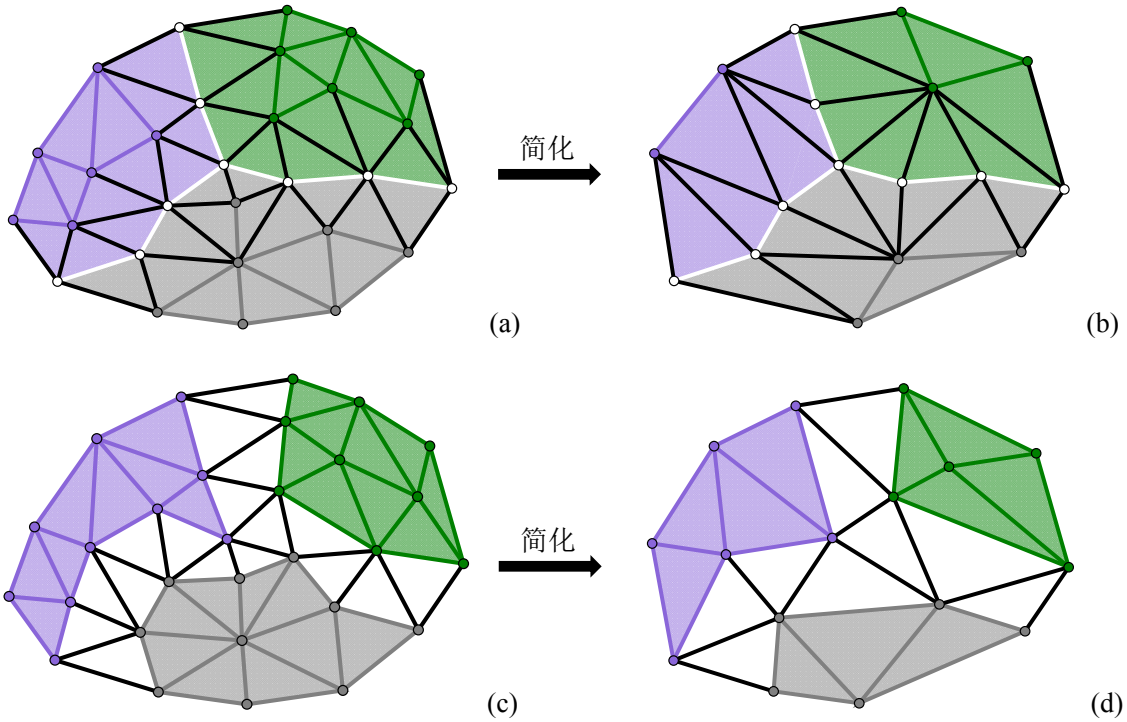


图 2.1 (a)、(b)：对三角形进行分片的简化过程。(c)、(d)：对点进行分片的简化过程。图中，不同颜色（紫，绿，灰）的三角形、边和顶点代表不同分片中的元素。(a)、(b)中白色边和顶点代表对三角形分割的分片边界，边界附近黑色的边代表不能被收缩的边。(c)、(d)中的白色三角形代表对点分割的分片边界，边界上黑色的边代表不能被收缩的边。

并不是所有简化操作都具有这种分片边界简化的独立性。以顶点移除为例，在图 2.5 中，不同分片中的边界顶点 A 与 B 被执行了顶点移除操作，对他们的操作影响到了分片之外的顶点。图 2.5 中对 B 的移除实际上是对 B 和 A 做了一次边收缩，对 A 的移除则是与 B 做了一次边收缩并在由绿色标记的边上做了一次边交换（边收缩和边交换操作可以产生任意拓扑结构的简化网格[37]）。在论文[12]中，作者在三角化的时候试图优化产生的三角形的质量，这使得三角化的方法具有不确定性。

事实上，所有顶点聚类操作都具有对拓扑结构的自动保持特性。边收缩操作可以转化为顶点聚类操作。

2.2 算法概述

本章算法的流程如图 2.6 所示。算法首先对网格中的顶点做一次扫描，获取网

格的包围盒，并将顶点的信息写入二进制顶点文件。二进制顶点文件用来在后续的对三角形进行分片的过程中，根据顶点索引来取得三维坐标。使用二进制文件是因为，这样可以根据顶点的索引来确定数据在文件中的位置。算法随后对网格进行分割。被分割的网格会被写入分片文件中，每个分片将进行一次内存简化。最后，算法合并所有分片并生成最终的网格。

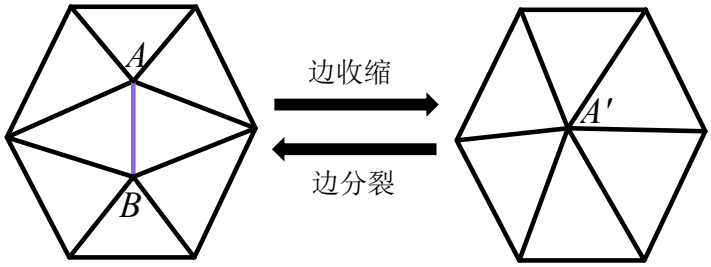


图 2.2 边收缩与边分裂互为逆操作。图中，A 和 B 收缩为 A’，反之 A’分裂为 A 和 B

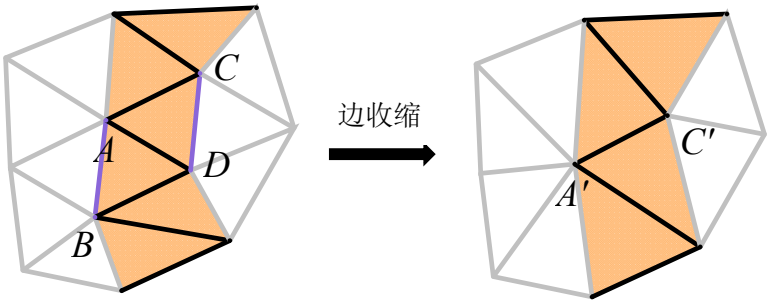


图 2.3 使用点分片的简化中，不同分片的边界顶点可以被独立地收缩。紫色的边为被收缩的边。

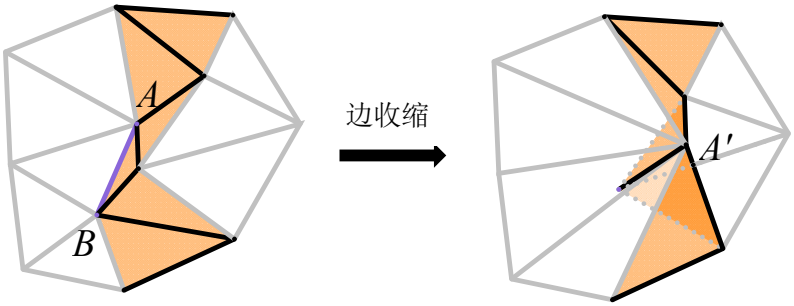


图2.4 边被收缩之后可能产生的跨越边界的问题。图中，紫色的边 AB 收缩为一个顶点 A’，这个点本来属于右边的分片，却跨越到了左边的分片。

为了以后的叙述，我们对网格中不同类型的元素做以定义。

内部点：分片内部的点。所有顶点都是某个分片的内部点。

内部边界点：分片中邻接于其他分片顶点的内部点。这里的边界不是网格的

真实边界，而是分片的边界。内部边界点属于某个分片，并能和其他内部点（包括内部边界点）进行收缩操作。

内部三角形：分片中，所有顶点都属于当前分片的的三角形。

外部点：分片中，内部边界点所邻接的属于其他分片的顶点。

外部三角形：分片中，内部边界点所邻接的分片边界三角形。外部点与外部三角形主要用于边界点的误差度量和代表顶点的计算。在边收缩算法中，度量简化操作的误差和计算代表顶点等过程需要顶点所有邻接元素的信息。除此之外，算法在简化过程中不会对他们进行修改或者收缩。

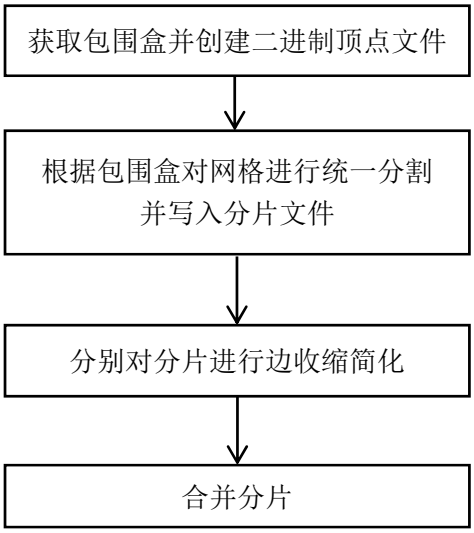


图 2.6 算法的流程

分片边界三角形：顶点属于不同分片的三角形，它不属于任何分片。分片边界上的三角形不参与收缩操作，但可能因为分片内部边界上的顶点的收缩而退化为一条边或者移动位置。对于顶点分属于三个不同分片的三角形来说，它一定不会退化。

分片边界边：类似于分片边界三角形，端点属于不同分片的边被称为分片边界边。它不属于任何分片，不会被收缩，不参与简化操作，但是它可以移动或与其他分片边界边重合。

我们假设算法的输入为索引网格的格式，即文件中先列出网格中所有的顶点，再列出所有的三角形。其中，顶点是以坐标的形式给出，三角形是以三个顶点在顶点列表中出现的位置作为索引给出。对于索引网格格式的输入，在使用某种对顶点进行分割的方法时，三角面片的分割与处理算法为：

(1). 遍历网格中的每一个顶点，对顶点按照某种策略进行分割，并写入分片文

件。

(2). 遍历每一个三角形，根据其顶点所在分片的情况将三角形写入不同分片。如果三个顶点落入同一分片中，则这个三角形属于这个分片。如果三个顶点落入不同的分片，则这个三角形为分片边界三角形，它将被写入一个分片边界的特殊文件。算法同时更新每个相应分片的外部点信息。对于分片边界三角形的某一个顶点来说，不属于它所在分片的其他顶点就是这个分片所邻接的外部点。外部点和外部三角形的信息也会写入各个分片文件中。

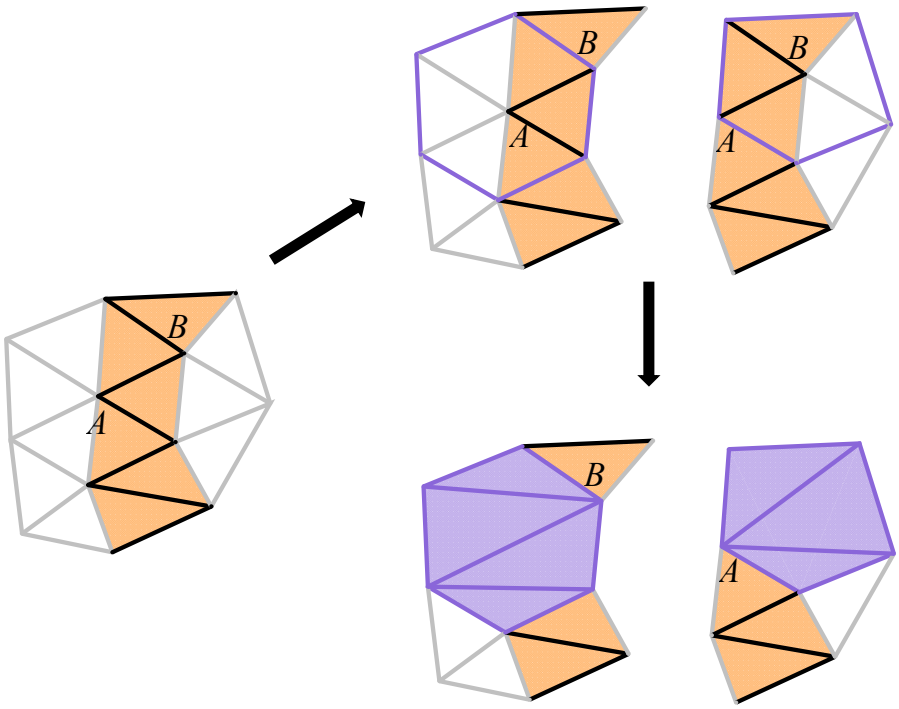


图 2.5 顶点移除不具有分片简化的独立性

我们的算法采用了统一空间分割对大规模网格进行分片。分割三角形或顶点的时候只需要考虑他们的坐标落在了哪个被分割的空间之内，除此之外不需要任何额外的信息。空间分割虽然没有考虑到分片顶点数目以及模型复杂度分配不均匀的问题等，对简化质量有一定影响，但迭代式算法本身保证了算法的质量。我们在对模型进行分割之前，先对顶点做一次扫描计算出模型的包围盒。然后根据包围盒在不同方向上的分割数目，将落入包围盒的每一个分割立方体中的网格部分作为一个分片。一个使用空间分割的模型分片的实例如图 2.7 所示。

分片方法所存在的不考虑分片顶点数目和模型复杂度的问题，可以通过优化每个分片输出顶点数目来解决。事实上，简化后顶点数目的分配要远远难于也重要于原始网格中顶点数目的分配[22]。

本文的算法使用式子（2-1）来分配分片中的顶点个数。

$$Ns_i = \frac{Ns}{Nm} Nm_i \qquad (2\text{-}1)$$

在式子（2-1）中，Ns_i 为分片 i 简化后输出顶点的个数，Nm_i 为分片中输入顶点的个数，Nm 为原网格的顶点个数， Ns 为用户希望达到的简化后网格的顶点个数。

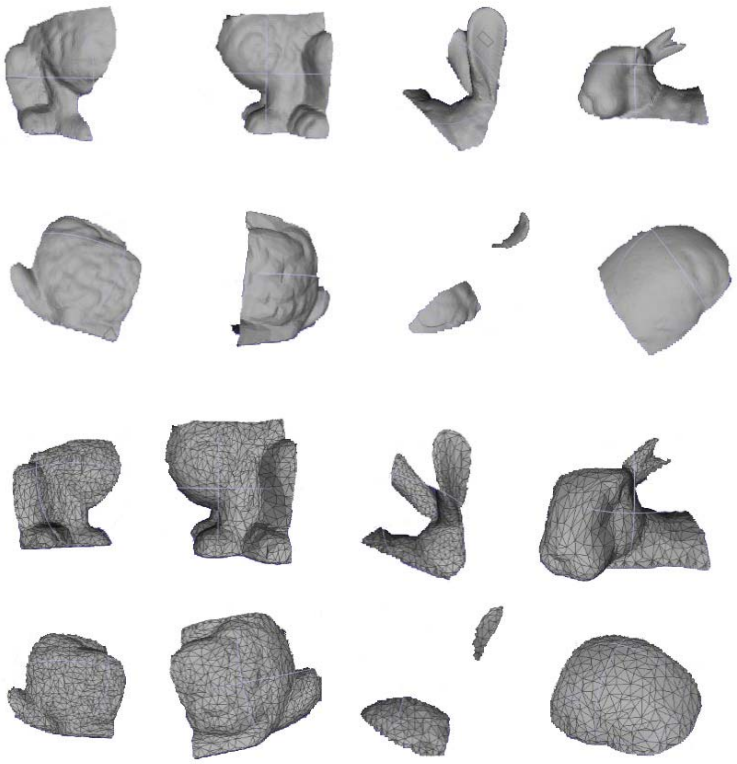


图 2.7 使用空间分割的模型分片实例。图中上半部分为原始模型的分片，下半部分为分片简化后的结果

由于简化算法自身考虑了网格的曲率，具有自适应性，因此直接使用顶点数目分配简化后的网格的方法能够产生较好的效果（可以见 2.7 节的简化结果）。并且，更换分配顶点数目的算法（比如按照面片的曲率进行分配）的代价并不大，本章节的代码实现可以提供很好的扩展性。由于本文研究的侧重点并不在如何最优地分配简化后网格的数量，因此并没有对它做特殊考虑。

需要指出的是，由于边界边不会在任何分片的简化中被收缩（如图 2.1 所示），本算法候选可收缩边的数目比其他分片算法要小。虽然这可能会影响算法的简化质量，但是由于边界元素占整个网格的比例较小，这种影响不大。

边界元素占整个网格的比例根据分片边界的不同而不同，而分片边界取决于

分片所使用的方法与模型自身的几何属性。对于本文所使用的统一分割，影响它的一个最直接的因素是分割的个数。分片个数的选取没有一个固定的数值，但一般需要在保证每个分片都能够装载入内存的情况下尽量减少分片的个数，以减轻分片边界个数过多对全局简化操作顺序的影响。在 2.7 节中，我们的测试一般使用了如 2x2x2 或 3x3x3 这样较小的分割数目，在这种情况下，边界是非常“稀疏”的。对于常规的具有真正视觉意义的模型来说（即最理想状态下是流形的，即使是非流形的模型，大部分区域也是流形的）跨越边界的边只占据整个模型的较小部分（见表 2.2、表 2.5 中“边界面比率”一列）。

2.3 LRU 缓存系统

在对三角面片进行分割的过程中，需要根据三角形的顶点索引取得存储在顶点列表中的坐标值。由于顶点数据是无法完全存储在内存中的，因此如何获取顶点数据是影响算法执行效率的最关键部分。

论文[26, 28]提出了使用外部排序进行顶点数据获取的方法。虽然外部排序完全遵循了对磁盘数据流式读取的原则，但为了获取三角形中的顶点数据，这种方法需要根据不同位置的索引对三角形文件进行三次外部排序以及顶点数据文件的读取。这大幅度增加了整个文件的扫描次数。

本文通过对顶点数据在内存中进行缓存来加快大规模数据的随机访问速度。每当程序发出数据读取请求时，缓存系统会先查询程序需要的数据是否在缓存中。如果缓存中存有需要的数据，系统会直接从缓存中读取并返回，这被称作一次缓存命中。如果缓存不命中，系统会从外存读取一次数据，并将这个数据存储在缓存中。根据程序对数据读取的时间一致性（time coherency），这个刚刚从低速存储器中读取的数据很有可能在不久的将来被程序后续的部分所请求。在将低速存储器中的数据读入到缓存时，由于缓存中数据不断的增加，系统会出现缓存没有空间以继续装载数据的情况。这时候，就需要将一部分数据从缓存中删除，这被称作缓存的交换。由于本文的算法不会对顶点数据进行修改，因此缓存不涉及在交换的时候对外存数据的更新问题。

在交换的过程中，选择哪些数据被删除出缓存是非常关键的策略，因为它会影响缓存的命中率。缓存命中率的提高会显著加快算法的执行时间。本文算法使用 LRU[38]缓存交换策略（least recent used，离当前最远使用）来决定哪些数据被删除出内存。这种策略的思想是删除那些上一次被使用时间离本次读取时间最远的数据。之所以选择这些数据进行删除，是因为基于数据读取的时间一致性，那些在越是在最近被读取的数据越有可能在不远的将来被程序读取。LRU 是一种在

缓存系统中非常常用的缓存策略，而且已被证明是一种非常高效的交换策略。

对于 LRU 缓存来说，数据结构是整个系统的核心。本文的 LRU 缓存系统所采用的数据结构应该支持以下功能：

(1). 根据顶点索引获取顶点的三维坐标数据。这个顶点索引是在一个很大的区间之内随机出现的。

(2). 能够动态地添加和删除顶点数据。在缓存有空闲的时候可以添加；在缓存装满的时候能够删除。

(3). 能够记录并更新顶点数据最近一次被使用的时间，并能够快速获取最近使用时间离当前最远的顶点。

为此，本文的数据结构的设计如下：

首先，为了能够根据一个随机的索引获取顶点数据，并支持动态的添加和删除，本文使用哈希表来存储顶点数据。哈希表中存有一个桶的指针数组，每一个元素指向一个由哈希值与桶的个数的模和当前桶的索引相等的所有节点所组成的链表。每次用户对顶点数据提出请求时，系统首先算出索引的哈希值，然后将这个哈希值与桶的大小取模，得出一个桶的索引。算法根据这个索引得到链表的起始地址，并查找链表中是否存有相应索引所对应的数据。链表结构保证了对数据的动态添加和删除。

```
struct CacheUnit
{
    unsigned int index;       // 顶点的索引
    CacheUnit *bucket_prev;   // 桶中前一个节点的指针
    CacheUnit *bucket_next;   // 桶中后一个节点的指针
    CacheUnit *use_prev;      // 使用时间序列中前一个节点的指针
    CacheUnit *use_next;      // 使用时间序列中后一个节点的指针
    ValType val;              // 顶点数据
}
```

图 2.8 LRU 缓存节点的数据结构

其次，我们并不显式地记录每次数据的使用时间，因为这样需要对数据进行排序。我们将所有的数据节点按照引用时间链接成一个链表。链表头节点存储着最近使用时间离当前最近的数据，链表尾部存储最近使用时间离当前最远的数据。本文缓存系统节点的数据结构需要维护两个链表的邻接关系：桶链表中的邻接关系以及使用时间序列链表中的邻接关系。每次插入数据的时候，除了要将其插入桶的链表，同时还要将其插入使用时间序列链表的头部。对于已经在哈希表中存

储的数据，需要将其从时间序列链表中删除并重新插入到头部，桶中的指针则不需要更改。每次缓存需要删除数据时，算法选择时间序列链表的尾部节点进行删除，并从桶中也删除这个节点。LRU 缓存中，节点的数据结构设计如图 2.8 所示。

2.4 分片文件格式设计

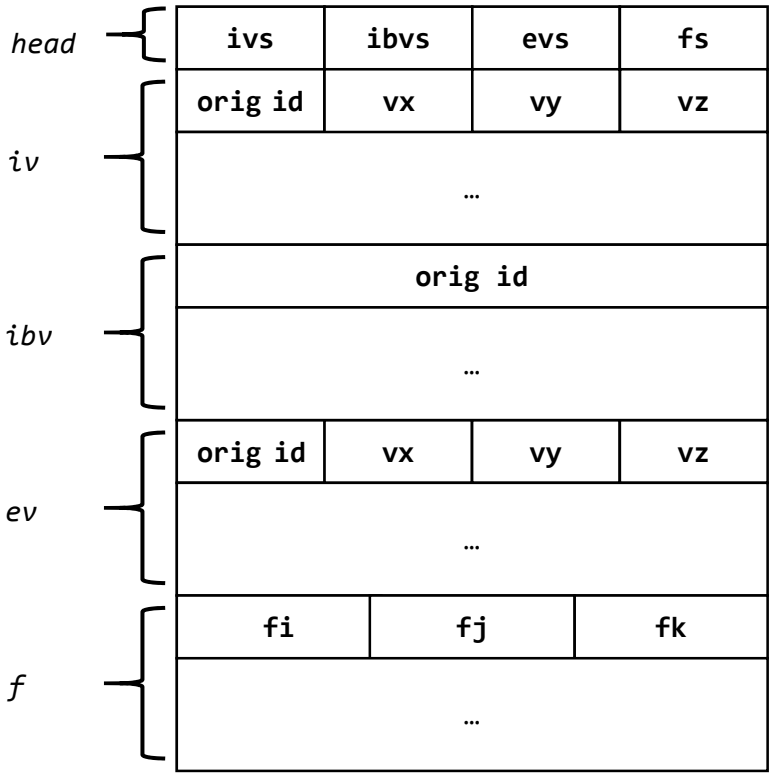


图 2.9 分片文件格式

由于分片的数据较复杂，有不同种类的顶点和三角形的数据。因此，分片文件必须能够区分这些不同的数据。本文的分片文件格式如图 2.9 所示。分片文件以一个首部（head）作为开头。在首部中，ivs 为内部点的个数，ibvs 为内部边界点的个数，evs 为外部点的个数，fs 为三角形的个数。接下来为所有内部点的列表（iv），每一个内部点的记录包含四个字段，分别为顶点在原始网格中的索引（orig id）和顶点的三维坐标（vx，vy，vz）。随后是所有分片边界点在原始网格中索引的列表，这些分片边界点的信息存储在上面的内部点列表中。之后是外部点的列表，外部点的记录与内部点相同。最后是三角形的列表，每个记录中包含了三个顶点的索引（fi，fj，fk）。

除了各个分片对应一个分片文件外，还有一些三角形是不属于任何分片的。

我们将这些三角形在分片的过程中，统一存储到一个分片边界三角形文件中。这个文件的记录包含了三角形的顶点在原始网格文件中的索引。

2.5 单个分片的简化

在对分片进行简化时候，算法将所有可收缩的边放入一个堆中，每次从堆中选取误差最小的边进行收缩并更新邻接边的误差。算法不断迭代直到顶点或三角形的个数达到指定数目。

在误差度量和代表顶点的计算上，我们选用了[18]所提出的二次误差度量（Quadric Error Metrics，QEM）。二次误差度量具有速度快、质量高的特点。

二次误差度量所度量的是某个点与一些三角面片距离的平方和。先考虑一个单独的三角面片，假设三角面片的参数表示形式为：

$$ax + by + cz + d = 0 \qquad (2\text{-}2)$$

对于某个顶点 $V_1 = (x_1, y_1, z_1, 1)$，这个顶点与平面的距离为：

$$D = ax_1 + by_1 + cz_1 + d \qquad (2\text{-}3)$$

顶点与平面距离的平方可以表示为：

$$\begin{aligned} D^2 &= (ax_1 + by_1 + cz_1 + d)(ax_1 + by_1 + cz_1 + d) \\ &= \begin{bmatrix} x_1 & y_1 & z_1 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} \begin{bmatrix} a & b & c & d \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} x_1 & y_1 & z_1 & 1 \end{bmatrix} \begin{bmatrix} a^2 & ab & ac & ad \\ ab & b^2 & bc & bd \\ ac & bc & c^2 & cd \\ ad & bd & cd & d^2 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix} = V_1^T Q V_1 \end{aligned} \qquad (2\text{-}4)$$

其中矩阵 Q 为对称矩阵，只需要知道平面的参数即可算出。顶点与两个三角形的距离之和可以表示为：

$$\begin{aligned} &D_1^2 + D_2^2 = V_1^T Q_1 V_1 + V_1^T Q_2 V_1 \\ &= V_1^T (Q_1 + Q_2) V_1 \end{aligned} \qquad (2\text{-}5)$$

以此类推，将两个平面集合对应的矩阵相加，即可得到顶点与两个平面集合的并集中所有平面的距离平方之和。

由于 Q 为对称矩阵，因此在实际实现中，我们只需要存储矩阵的上三角部分，这减少了程序占用的内存空间。

在求取代表顶点的时候，首先，将距离值

$$V^TQV=\begin{bmatrix}x & y & z & 1\end{bmatrix}\begin{bmatrix}q_{11} & q_{12} & q_{13} & q_{14}\\ q_{21} & q_{22} & q_{23} & q_{24}\\ q_{31} & q_{32} & q_{33} & q_{34}\\ q_{41} & q_{42} & q_{43} & q_{44}\end{bmatrix}\begin{bmatrix}x\\ y\\ z\\ 1\end{bmatrix} \qquad (2\text{-}6)$$

展开并求偏导，令每个偏导等于零，可以得到：

$$\begin{bmatrix}q_{11} & q_{12} & q_{13}\\ q_{21} & q_{22} & q_{23}\\ q_{31} & q_{32} & q_{33}\end{bmatrix}\begin{bmatrix}x\\ y\\ z\end{bmatrix}=\begin{bmatrix}-q_{14}\\ -q_{24}\\ -q_{34}\end{bmatrix} \qquad (2\text{-}7)$$

求取这个线性方程组即可得到代表顶点的位置。在线性方程组无解或解不唯一的情况下，算法将沿着被收缩边的两个端点求取这条线段上使得误差最小的点。若依然无法得到唯一解，则算法将选取两个端点的中点作为代表顶点。代表顶点与矩阵的积即为边收缩的误差。关于二次误差度量的更详细的阐述可以参见 Garland 的论文[11]。

在内存简化算法中，我们使用带有邻接关系的索引网格存储网格数据。这种数据结构中每个顶点保存了邻接三角面片的索引。在每次增加三角面片的时候，算法会向它所包含的三个顶点的邻接面片索引列表加入当前三角面片的索引。有了这样的索引，算法就可以很快计算出顶点所邻接的顶点和边。

在堆数据结构的设计上，由于需要更新被收缩边所邻接的所有边的误差值和代表顶点，因此需要保持边的邻接关系且能对堆动态的更新用于比较的键值。常规的堆数据结构使用数组来存储它的节点，但是如果使用数组来存储待收缩边，由于每条边都需要维护邻接边的索引，堆的不断更新会导致索引的频繁更新。这样会使得邻接边索引的管理变得耗时。为此，我们将边数据存放在进程堆（进程的堆内存空间，而不是堆数据结构）中。在堆数据结构中只存放待收缩边的地址，边数据中的邻接边列表也只存放邻接边的地址。

2.6 分片合并

分片合并过程如图 2.10 所示，在每个分片进行简化的时候，分片中内部点和内部三角形在被简化之后会被直接输出到简化后的模型，边界点简化前后的索引会被写到一个全局的索引映射表中。在分片合并的最后，算法读入所有分片边界三角形，将其中的顶点索引从原始索引映射为简化后索引，并丢弃退化的三角形。

在分片文件中，顶点列表存储着顶点在原网格中的索引。如果在简化过程使用原网格的索引，将会给算法带来很多的不便。因此我们在内存简化的时候对分

片中的顶点采用局部索引。局部索引是某个顶点在分片文件的顶点列表中出现的位置。我们使用了一个哈希表来保存原网格的索引与局部索引的对应关系。在读入分片中的三角形的时候，算法根据哈希表中的索引对应关系，将三角形的全局索引转化为顶点在分片中的局部索引。

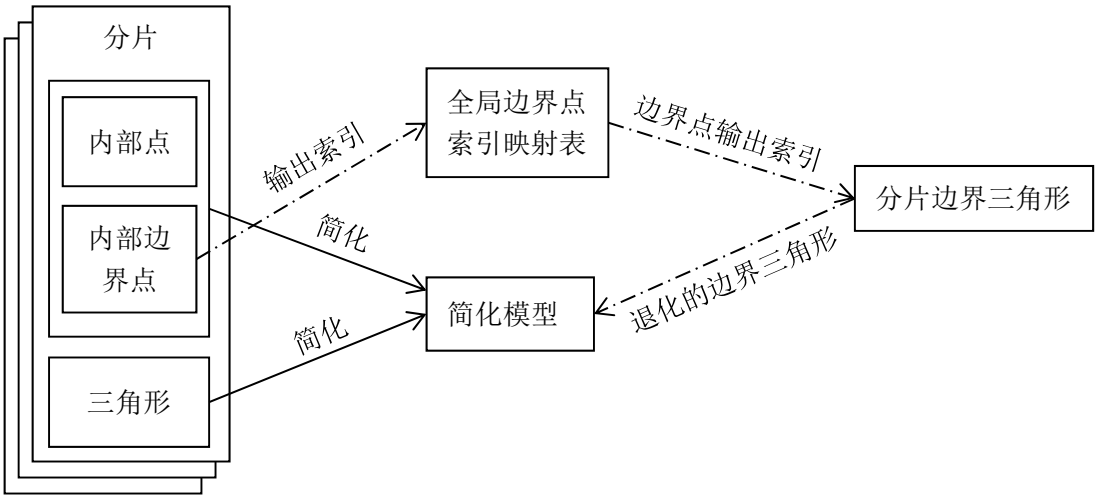


图 2.10 分片合并的过程

由于分片逐个被处理，因此，算法保存一个简化模型中已输出顶点的数目。在指定分片中简化后顶点的索引时，算法将简化后顶点的局部索引加上这个已输出顶点的数目，得到简化后顶点在输出模型中的索引。

2.7 执行结果

我们对不同模型的简化质量和运行时间进行了测试。在实验过程中，编写和测试程序所使用到的设备和平台如表 2-1 所示。

我们采用 Metro[39]工具来衡量简化后网格与原网格的误差。为了测试我们的算法的简化效果，我们将分片算法的简化质量与 QSlim[18]算法的简化质量做以对比。我们对不同模型分别衡量了同等简化率下，两种算法的简化后模型与原模型的误差。由于 QSlim 算法和 Metro 工具只支持能够装载入内存的模型，因此，我们只对那些能够装载入内存的模型进行简化质量的比较。由于这些模型的顶点与面片个数已经足够可观，因此这种比较效果足够有说服力。对不同模型使用两种算法所产生的误差如表 2.2 所示，部分模型的显示结果如图 2.11 所示。

从以上实验结果可以看出，我们的算法对于大部分模型的简化结果与 QSlim 算法没有很大区别，这说明了我们算法的简化质量可以与内存简化算法相比拟。

如表 2.3、图 2.12 所示，我们同时对 Dragon 模型（原模型有 437645 个顶点，

871414 个面）在不同简化率下分片算法与 QSlim 算法的误差做了对比。从中可以看出，在简化率较高的情况下，分片算法的质量与内存简化算法的质量相当；而在简化率较低的情况下，由于不同分片简化后模型的顶点数目分配没有考虑到模型本身的复杂度，因而有些地方被过度简化而有些地方简化不够，影响了分片简化的质量。我们希望能在日后改进简化后模型顶点数目分配的算法，加入对模型复杂度的考虑，缓解在简化率较低的情况下算法出现的分片之间简化程度不均匀的问题。

表 2-1 程序编写和测试所使用的设备和平台

设备或平台	描述
CPU	Intel(R) Core(TM)2 Duo CPU，主频 2.53Hz
GPU	NVIDIA 的 GeForce 9800GT，显存容量为 1G
内存	容量为 2G
操作系统	Windows 7 Ultimate
开发语言	C++，界面部分使用 Qt
开发环境	Visual Studio 2010
其他引用库	OpenGL 3.2，boost 1.49

表 2.2 不同模型使用两种算法进行简化的误差对比

模型	算法	面数	简化后	分割个数	边界面比率	平均误差	最大误差
Dragon	QSlim	871414	5439			0.000077	0.001590
	分片		5441	2x2x2	0.962%	0.000077	0.001590
Bunny	QSlim	69451	1872			0.000099	0.002572
	分片		1872	2x2x2	2.968%	0.000104	0.002572
Buddha	QSlim	1087716	6017			0.000058	0.000374
	分片		6017	2x2x2	0.8801%	0.000059	0.000418
Armadillo	QSlim	345944	5988			0.101051	0.801661
	分片		5988	2x2x2	1.26%	0.110945	0.517656

本文方法对大规模数据的处理效率至关重要。在算法的分片过程中，由于使用了 LRU 缓存，对文件的随机读取次数可以得到极大的削减。缓存系统对文件的读取次数取决于缓存的大小以及网格文件中三角形列表对顶点索引的序列。除了分片过程中的 LRU 缓存系统之外，算法对文件的读写完全遵循了流式读写的原则，并且只对文件读取一次。因此，总体来说，本文算法对大规模文件的读写具有较

高的效率。

表 2.3 Dragon 模型在不同简化率下两种算法的简化误差比较

简化率	算法	误差	误差相差比率①
0.1898%	QSlim	0.000297	19.5%
	分片	0.000355	
4.119%	QSlim	0.000155	-5.16%
	分片	0.000147	
6.244%	QSlim	0.000114	-2.63%
	分片	0.000111	
8.393%	QSlim	0.000087	4.59%
	分片	0.000091	

为了体现我们的 LRU 顶点缓存系统的实际工作效率，我们测试了缓存系统在处理不同模型时缓存的命中率。测试结果如表 2.4 所示。

表 2.4 缓存系统的读取效率比较

模型	顶点数	面数	缓存顶点数	缓存命中率	分片时间（秒）
Bunny	35947	69451	10784	78.0252%	0.4992
Buddha	543652	1087716	131293	83.3396%	6.801
Thai Statue	5000000	10000000	1499998	83.2931%	44.16
Lucy	14027872	28055742	4208361	83.0641%	130.06

从表 2.4 我们可以看到，我们将 30%的顶点进行了缓存，命中率能达到 80%左右。对于后面的两个大规模网格来说（大小分别为 185MB 和 520MB），在分片过程中，由于三角形不需要缓存，而是以批处理形式被访问的，因此余下的内存可以全用来做缓存。从表 2.4 可以看出，即使是最大的模型 Lucy 也能缓存 30%的顶点，使得命中率非常高。

为了测试大规模数据的执行时间与简化结果，我们使用了两个大规模网格进行测试，分别为：Lucy（14027872 个顶点，28055742 个面，共 508 MB），Thai Statue（5000000 个顶点，10000000 个面，共 220 MB）。这两个模型在不同参数下的运行时间统计如表 2.5 所示。由于 QSlim 等内存简化算法无法处理这些大规模网格，因此本文并没有列出这些网格使用内存简化算法和使用本文算法结果的对比。

两个模型的简化后渲染结果如图 2.13、2.14 所示。从上面的结果中可以看出，对于超大规模的数据，本算法具有较高的执行效率和较好的简化结果，能在较短

① 误差相差比率 =(分片算法的误差 - 内存简化算法的误差) / 内存简化算法的误差

的时间内执行完毕，并在较低的简化率下保持模型的细节和特征。

表 2.5 大规模网格的简化时间

模型	面数	简化后	包围盒分割	分片数①	边界面比率	执行时间（秒）
Lucy	28055742	59967	3x3x3	25	0.343%	330.58
Thai Statue	10000000	60000	2x2x2	8	0.288%	115.58

2.8 本章小结

随着三维扫描和模型重建技术的不断发展，GB 级别大小的网格被创造出来。由于常规的内存简化算法无法直接应用于大规模网格，这些模型对简化算法提出了很大的挑战。很多不同的技术都试图在大数据外存访问效率和网格简化质量中间取得平衡，他们中有的方法只顾及了一方面，有的能做到两者兼顾。本章的方法能兼顾大数据访问效率和简化质量，更重要的是，它比之前同类型的方法对整个网格需要更少的处理次数，且能处理输出网格无法载入内存的情况。

本论文的方法基于对网格中顶点而非三角面片进行分片，对顶点进行分割的方法所产生的边界为三角形边界，这种边界在分片各自被简化的过程中不需要被保持。这是因为，本论文所使用的边收缩操作对分片中顶点的拓扑结构改变可以保持在分片之内，而不影响其他分片。这保证了分片之间简化的独立性，使得边界被简化变得可能。在对简化后的分片进行合并的时候，本论文的方法只需要在内存中保存网格分片边界处的顶点信息。经测试表明，本论文的算法能在较短的运行时间内提供和内存简化算法相比拟的简化质量，无论用纯粹的视觉比较或者标准的误差度量工具进行比较。

① 有的包围盒分割不包含模型

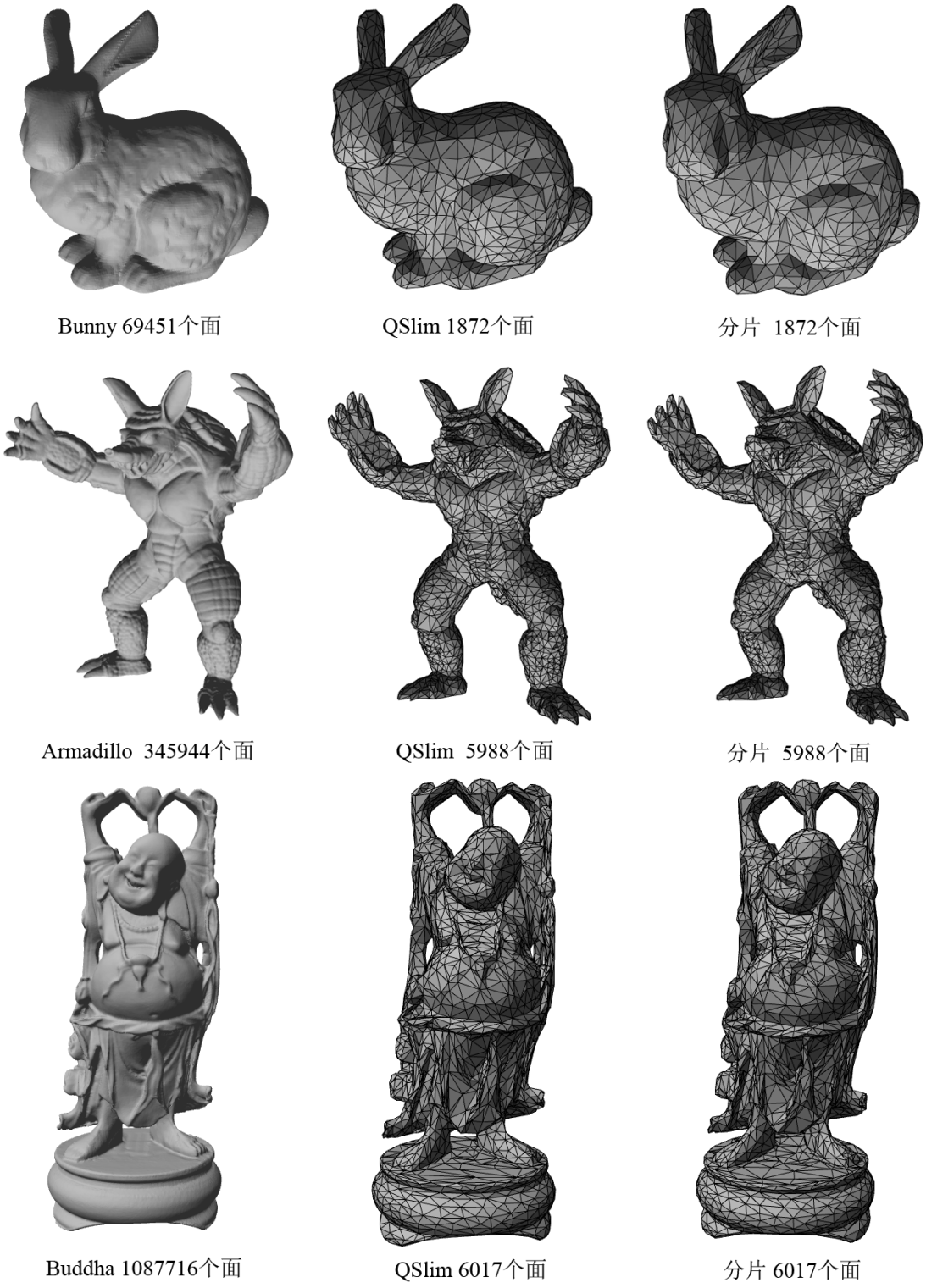
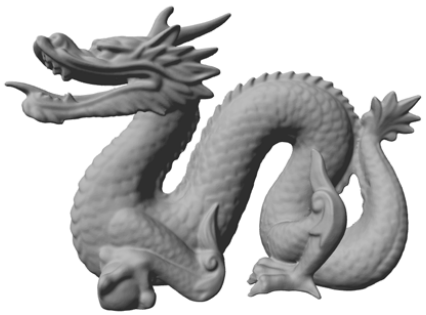
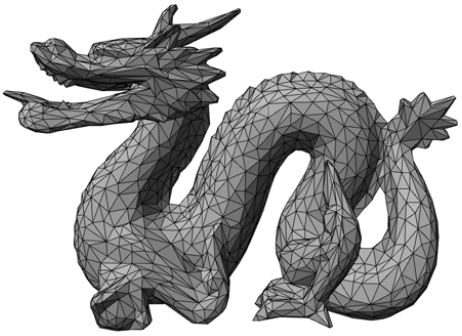


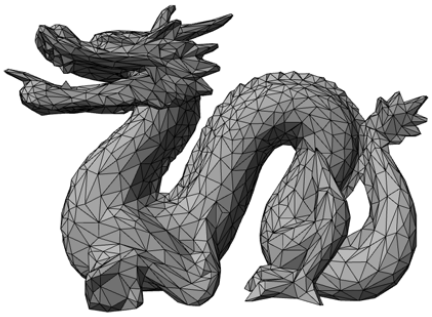
图 2.11 部分模型的简化结果



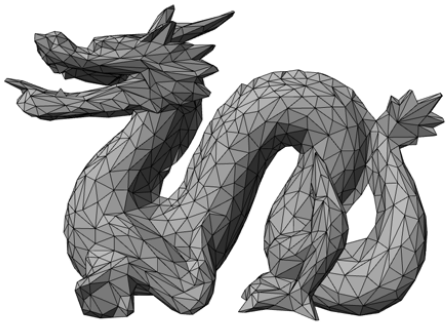
Dragon 871414个面



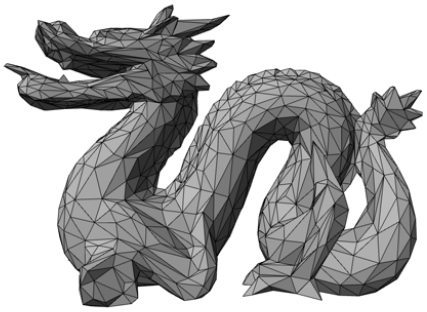
QSlim 5439 个面



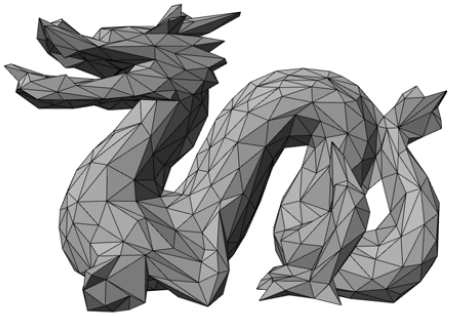
分片 5441个面



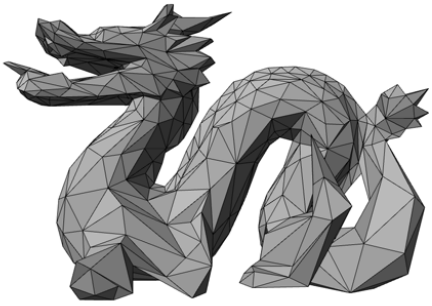
QSlim 3589 个面



分片 3590个面



QSlim 1659 个面



分片 1654个面

图 2.12 Dragon 模型在不同简化率下的显示结果

图 2.13 (a)：Lucy 模型[40] (28055742 个面)　(b)：简化后模型(59967 个面)　(c)(d)(e)：简化网格的部分细节

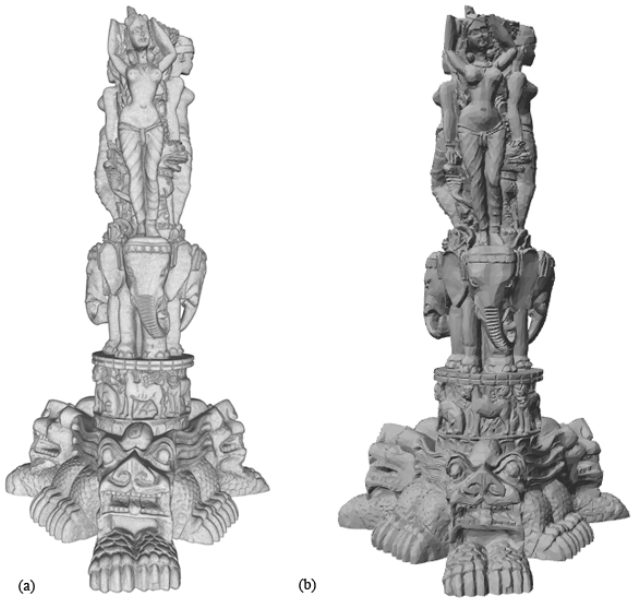


图 2.14　(a)：Thai Statue 模型[40] (10000000 个面)　(b)：简化后模型(60000 个面)

第 3 章　大规模等值面即时简化

在本章中，我们针对某些大规模体数据生成的等值面过大无法装入内存的情况，提出了一种在体数据没有生成完网格时就直接进行简化的算法。这种“生成一些，简化一些”的算法不需要将所有生成的等值面全都装入内存，内存占用较小。我们的算法可以对在内存中无法生成的等值面直接进行简化而不需要在进行一次基于外存的等值面生成之后再进行 out-of-core 简化。

3.1　算法概述

基于流式简化的原理，我们的算法使用边收缩对已生成的网格进行简化，同时保持已生成和未生成的边界不被简化。通过网格的不断生成，生成边界不断移动，网格简化的过程会扩展到整个等值面。为了能够在内存中使用迭代式边收缩简化算法，我们在生成的过程对等值面的拓扑关系进行了重建，以提供边收缩算法需要的顶点和面之间的邻接关系。为了确知顶点何时退出边界并可以被简化，我们在等值面生成过程中加入了对顶点“终结”信息的判断。由于我们算法产生的结果是为了后续的在内存中的渲染，因此，为了优化简化的质量，我们并没有使用流式简化中常用的“随机选取候选操作取最优”的方法，而是在内存中保持了一个全局简化操作的最优队列。在每次生成一部分网格的时候，我们的算法会将可简化的边加入到这个最优队列中，而之前简化过的边同时也在这个队列中。

如图 3.2 所示，在执行程序之前，用户首先指定等值面的简化率和每次简化之前生成的等值面的个数（缓存大小）。其中，缓存大小设置的越大，简化质量越高，但算法占用内存也越大，执行时间越长；反之的则简化质量越低，占用内存越小，执行时间越短。

算法的流程如图 3.1 所示。算法首先生成指定数量的等值面，随后根据用户指定的简化率对内存中已生成的网格（这些网格有些是已经简化的）进行简化。这个生成-简化的过程会不断迭代直到所有等值面已生成。算法会最后做一次简化以保证网格被简化到指定简化率。

3.2　等值面生成过程中终结信息的判断

我们使用了最常用的 Marching Cubes 算法[13]来生成等值面。Marching Cubes

算法判断每个单元立方体体素的数据值与等值面数据值的大小关系，并对立方体中端点与等值面密度值的大小关系不同的边通过线性插值得到一个等值面上的点，然后通过一个查找表生成等值面中点的拓扑关系（即三角形）。

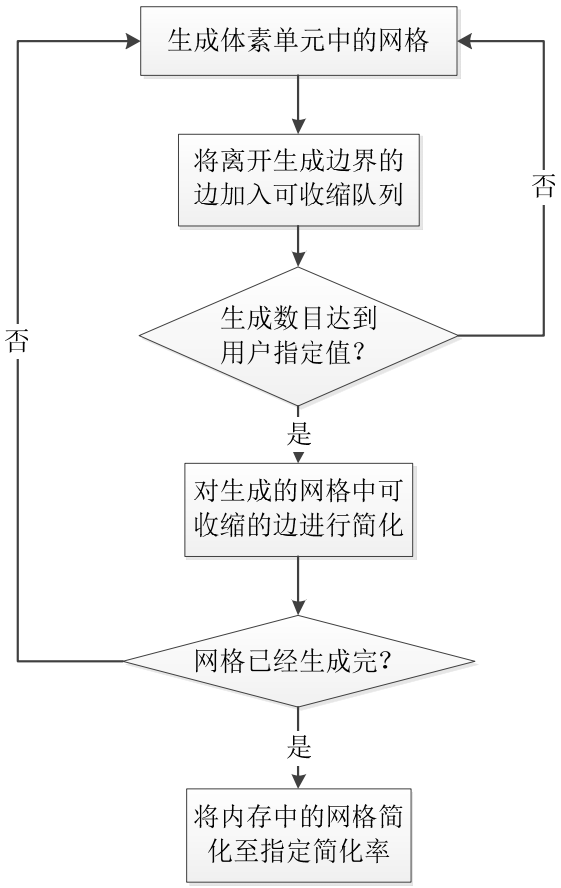


图 3.1 等值面即时简化算法流程

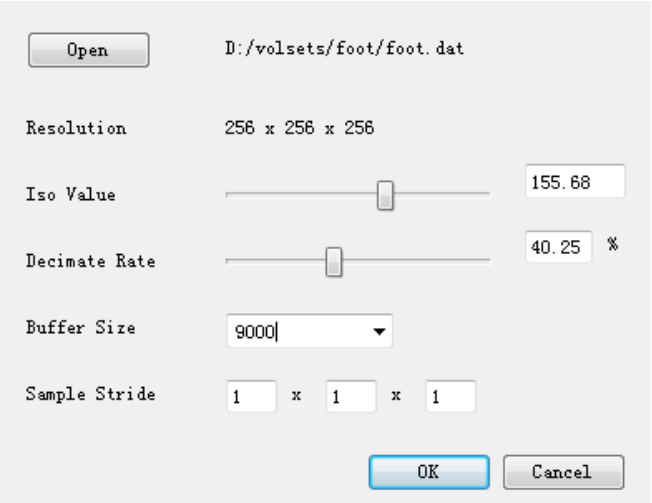


图 3.2 算法执行前的参数设置界面

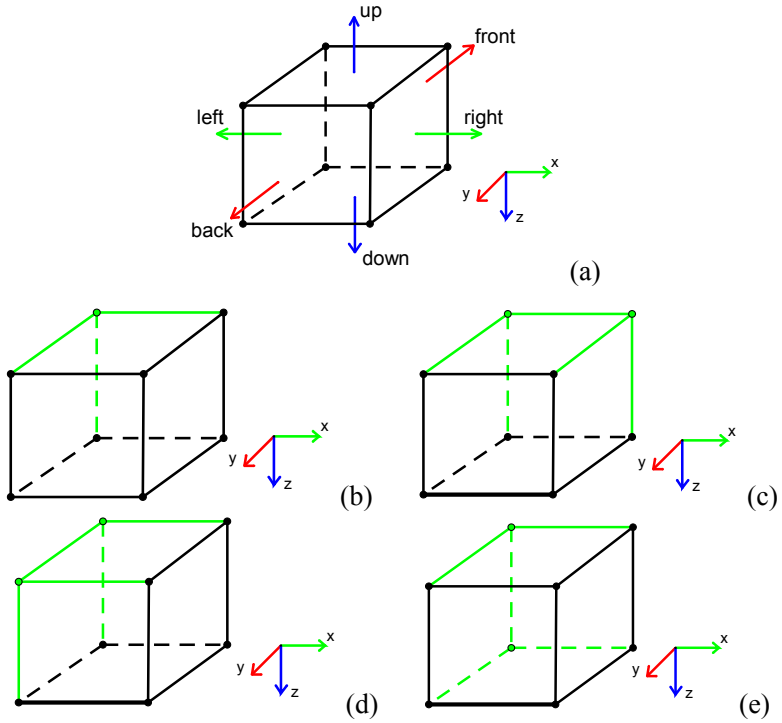


图 3.3 不同边界上立方体的终结信息。绿色的边或点上插值所得顶点由于所邻接的所有立方体已经被遍历过，即邻接的三角形都已被生成，因此被确定可以终结。(a)：遍历的方向 (b)：内部立方体的终结信息 (c)：最右立方体的终结信息 (d)：最后立方体的终结信息 (e)：最下立方体的终结信息

由 Marching Cubes 算法所生成的点在一般情况下会落在体数据所组成的各个小立方体中的边上，但也有可能落在小立方体的顶点上。在体数据所组成的三维空间中，每个小立方体的边邻接其他四个小立方体，每个小立方体的顶点邻接其他八个小立方体。我们的算法规定对立方体的遍历必须按照某种方向和次序进行。比如算法必须按照先 x 方向后 y 方向然后 z 方向，并在每个坐标轴方向都按从小到大进行遍历。依照这种顺序，在遍历到某个立方体的时候，我们能够获知当前立方体的某个边或顶点所邻接的立方体是否已经被遍历完。如果已经遍历完，那么这条边或顶点上所生成的等值面上的顶点就不会再被后续生成的等值面所引用，因此可以对这个顶点做一个“终结”。对顶点进行“终结”意味着，顶点离开了等值面生成的边界且可以被简化[29, 30]。

在遍历立方体的过程中，体数据边界上的立方体和内部的立方体所能确定的顶点终结信息是不同的。为此我们规定，对 x 的遍历方向为自左至右（left to right），对 y 的遍历方向为自前至后（front to back），对 z 的遍历方向为自上至下（up to down）。我们将边界上的立方体分为最左立方体，最右立方体，最前立方体，最后立方体，最上立方体，以及同时满足他们之间的两个或三个条件的立方体。这些

不同边界上的立方体共有 $6+C_3^2 2^2+2^3$ 种不同的情况。他们中的大部分都和内部立方体的顶点终结情况一样，只有那些远离遍历原点的边界立方体（最右，最后，最下及他们的组合）的终结信息才不同。图 3.3 显示了不同边界上立方体顶点的终结情况。图中只列出了最右，最后，最下的情况。其他几种情况可以由它们组合得到。对于最右最后最下立方体（即最后一个遍历的立方体），它的所有顶点和边上生成的顶点均可以被终结。

3.3 生成边界的延伸与模型的简化

在生成等值面的过程中，新生成的面片会不断加入到已经生成的网格中，且不断会有顶点被终结，离开已生成网格的边界，使得网格边界往生成的方向移动（如图 3.4 所示）。图 3.4-(a)为某一时刻，算法保存的被简化过的已生成网格。由于被保持，网格边界的元素密度明显比内部高。图 3.4-(b)为对 1 加入了两个顶点（A、B）和三个三角形之后的已生成网格。其中顶点 C 和 D 被终结，导致其退出生成边界，并产生了新的可收缩边 CD、DE、DF。

随着立方体被一个一个遍历，不断会有顶点被终结，并有新的可收缩边产生。为了保证我们的算法正确的检测到新产生的可收缩边并将每条可收缩边唯一地加入到队列中，我们在终结每个顶点的时候，将这个顶点与它所邻接的所有已被终结的顶点组成的边加入到可收缩队列中。在终结顶点的时候将可收缩边加入到队列中的伪代码如算法 3.1 所示。

算法 3.1 终结顶点时加入可收缩边

```
FINAL-VERT-ADD-COLLAPSE-EDGE(v)
1  s ← STAR(v)
2  for each v_i in s
3     if v_i is finalized
4        add <v, v_i> to heap
```

由论文[29]所启发，我们对模型的简化策略是，每次生成一部分之后，就对当前内存中的网格面片简化到 N_s 个。如式子 3-1：

$$N_s = N_{gen} \cdot R_s \qquad (3\text{-}1)$$

其中，N_{gen} 为已生成等值面的个数，R_s 为简化率。在这个过程中，如果用户设置的缓存太小或者简化率太高，会使得初始模型被过度简化。虽然我们的算法能缓存的模型是有限的，但是我们希望尽量让更多的简化操作互相之间得到比较，所以我们在简化的时候让队列中已经存在很多之前生成的简化操作，而不是只执

行刚刚生成的简化操作。我们设置一个初始简化的阈值 R_i。如果简化率小于 R_i，在初始的时候就只按照 R_i 来进行简化。在下次生成的时候，只保证生成的面片能让缓存中面片个数达到用户指定的缓存大小。我们在每次初始的简化之前预估能够达到的简化率。初始简化过程不断迭代直到预估简化率小于 R_s。整个过程如图 3.5 所示。

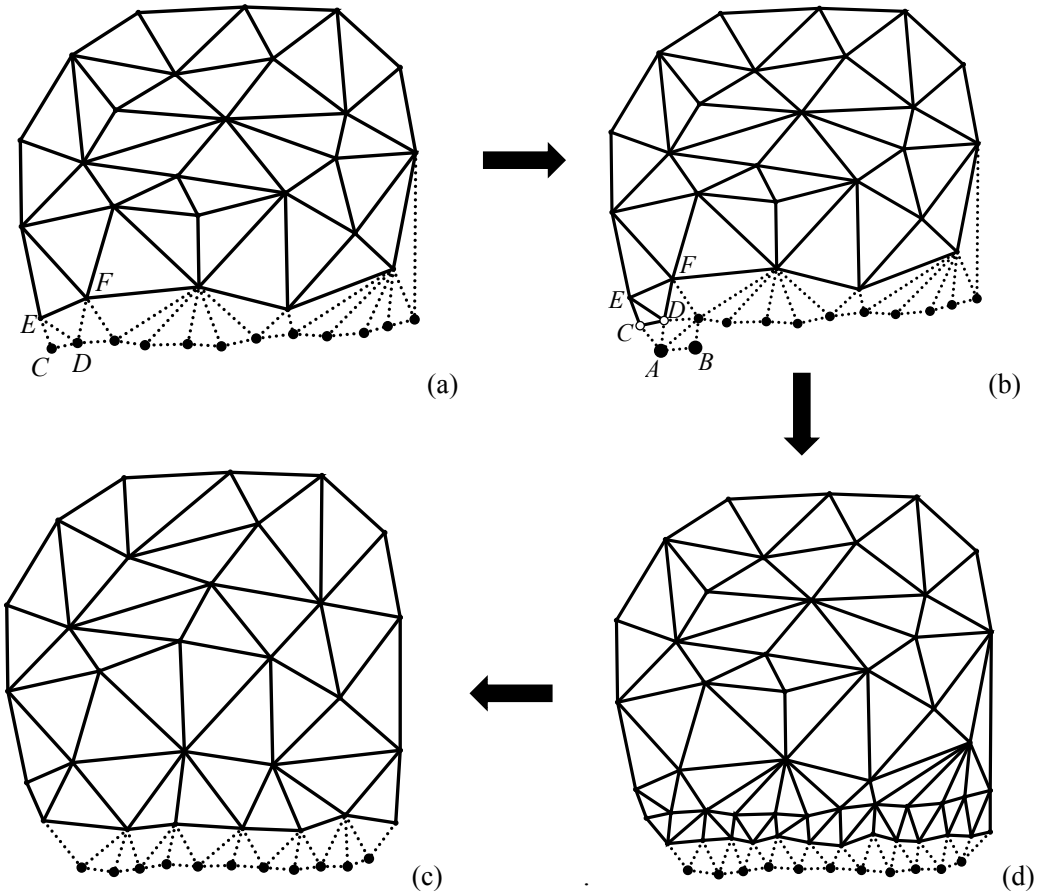


图 3.4 随着等值面的不断生成，边界向生成的方向移动。在简化的过程中，边界上的点会被保持不简化，使得边界上顶点的密度明显高于内部的顶点。(a)：算法的某步中，已经被简化过的模型以及被保持的边界。其中较突出的顶点为边界点，虚线显示的边为不可收缩的边。(b)：三个面片被加入到当前的网格中，使得边界被移动。(c)：在原网格基础上新生成了指定数目的网格。新生成的网格密度明显大于之前就存在的网格，且边界已经与(b)中完全不同。(d)：对本次新生成的网格进行简化的结果。新生成网格被大量简化因此密度已经与其他部分相差无几，原先的内部顶点也有少量被简化。边界在简化过程中被保持。

3.4 顶点拓扑关系的重建与网格数据结构的设计

生成过程中，由于边收缩算法需要显式的网格拓扑关系，因此需要对拓扑关系进行重建。在原始的 Marching Cubes 算法中，得到的网格为每个三角面片直接包含三个顶点的三维坐标的形式。我们以顶点的三维坐标为关键字、索引为值建

立一个哈希表，对每个立方体生成的三角形的每一个顶点，根据顶点的坐标向哈希表里查找这个顶点是否存在，若存在则获取顶点的索引；若不存在则给顶点指定一个索引并将这个一对关键字与值插入到哈希表中。

算法 3.2 对立方体生成的等值面进行拓扑重建

```
RECON-TOP-MC-SURF(farr)
1  for each f in farr
2     if f.vc1 in vhash
3        idxf.vi1 ← vhash[f.vc1]
4     else
5        vhash[f.vc1] ← new index
6        idxf.vi1 ← vhash[f.vc1]
7     retrieve index for v2, v3 as v1
8     add idxf to ifarr
```

在这里，为了减少对内存的占用（如果不对哈希表中的点进行删除，它最后将装入等值面生成过的所有顶点。这也无法实现我们的算法不将所有生成的网格装入内存的目标），我们在哈希表中只保存边界上的顶点。对那些被“终结”而离开边界的顶点，我们将它从哈希表中删除。因为被终结的顶点不会被后面生成的等值面所引用。这样，哈希表中需要保存的顶点信息就只包括边界上的顶点。如图 3.3 所示，边界顶点只是体数据中某一层体素上的顶点，数量较小。对那些被“终结”而离开边界的顶点，我们将它从哈希表中删除。生成一个立方体的等值面之后再重建拓扑关系的伪代码如算法 3.2 所示。

在顶点和面片的容器选择上，我们使用了哈希表。哈希表中以顶点或面片的索引为关键字，以其数据为值。它能支持动态的插入和删除，能保证我们算法不装入生成的全部等值面的目标，且查询时间较短。

3.5 执行结果

在测试中，我们使用了不同大小的模型对我们的即时简化算法做了测试。测试所使用的硬件以及软件平台信息如表 2-1 所示。不同模型的测试数据如表 3-1 所示，简化结果如图 3-6 所示。

从以上的简化结果可以看出，大部分模型都得到了较好的简化结果，即使在很高的简化率下模型依然能较好地保持特征，且大部分模型的运行时间都较快。Head ZMW 模型由于尺寸较大，因此简化过程需要用户等待一定时间。但这个时间也优于生成之后再使用 out-of-core 简化的方法，因为 out-of-core 算法需要写文件和处理大数据等非常耗时的操作（可见本文第 2 章）。

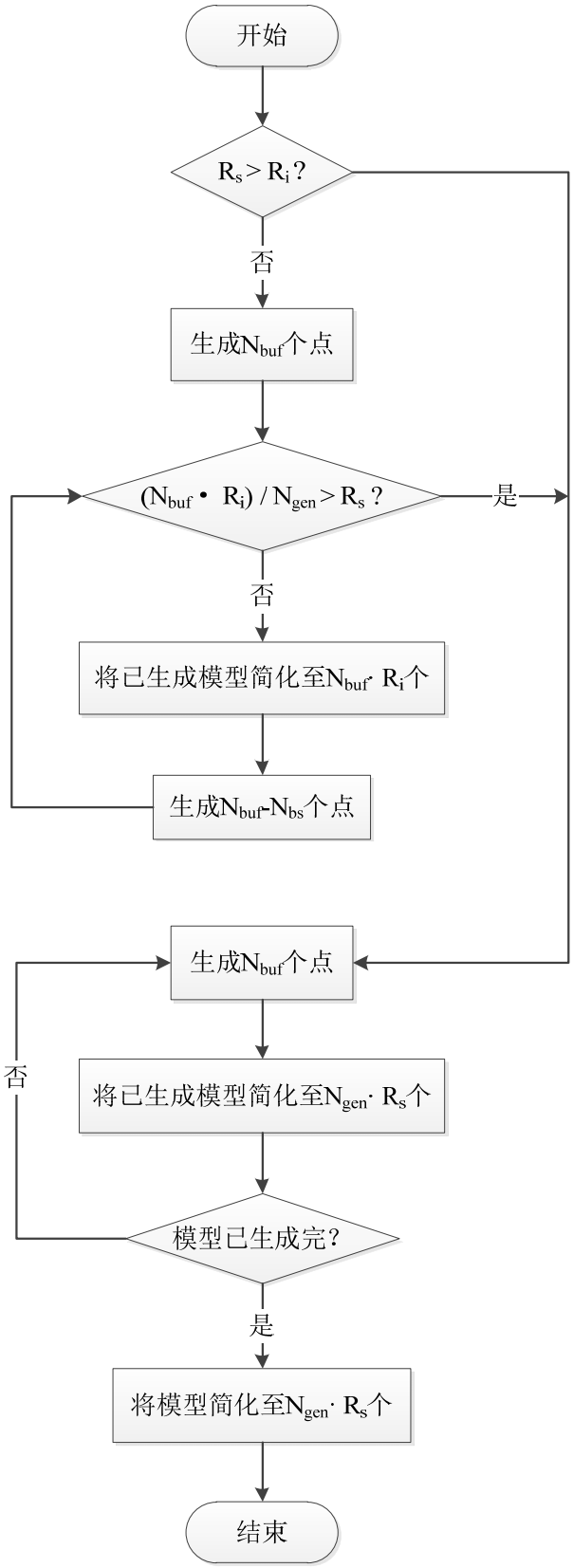

开始
$R_s > R_i$?
否
生成N_{buf}个点
$(N_{buf} \cdot R_i) / N_{gen} > R_s$?
是
否
将已生成模型简化至$N_{buf} \cdot R_i$个
生成N_{buf}-N_{bs}个点
生成N_{buf}个点
否
将已生成模型简化至$N_{gen} \cdot R_s$个
模型已生成完?
是
将模型简化至$N_{gen} \cdot R_s$个
结束

图 3.5 简化模型的流程

表 3-1 不同模型简化的统计数据

模型	模型尺寸	Iso 值	缓存	生成面数	简化率	执行时间(秒)
Ear	128x128x30	127.5	10000	124587	10.75%	1.622
Statue Leg	341x341x93	127.5	10000	242344	20.15%	5.413
Teeddy Bear	301x324x56	127.5	20000	133184	20.15%	3.401
CT_256_256_108	256x256x106	127.5	20000	913934	20.8%	14.089
Foot	256x256x256	127.5	50000	586220	20%	15.132097
Head ZMW	500x500x476	100	50000	4741933	5%	104.318
					10%	101.681

我们对 Boston Teapot 模型（尺寸 256x256x178，简化率 5%，Iso 值 65，全部生成的面数 508142）使用了不同大小的缓存测试了算法执行效率和简化结果，来验证算法在缓存大小不同的情况下简化结果与执行效率的差异。统计数据与简化结果表 3-2、图 3-7 所示。

表 3-2 Boston Teapot 模型使用不同大小缓存简化的统计数据

缓存大小	执行时间
50000	16.427
40000	15.163
30000	16.162
20000	16.287
10000	16.193
5000	16.208
2000	16.333

3.6 本章小结

很多三维图形软件需要生成体数据的等值面，供用户观察扫描物体的轮廓。但有些体数据本身过大，导致生成的等值面很大，甚至超过了内存的装载能力。如果将这些网格完全生成出来再使用 out-of-core 简化，会使得整个处理过程耗时过长。本论文提出了在内存中生成一部分等值面就进行简化的方法。

以往对体数据生成的等值面直接进行简化的方法随着网格的不断简化需要输出简化的网格，减少了简化的比较次数。另外，它们对顶点何时退出生成边界的判断没有充分考虑体数据本身的特性。本论文使用边收缩对已生成的网格进行简化，同时保持生成边界不被简化。为了能够在内存中使用迭代式边收缩简化，本方法在生成过程中对等值面的拓扑关系进行了重建。随着网格的生成，生成不断

移动，简化会扩展到整个等值面。本论文的算法假设简化后的网格能够装载入内存，因此在内存中保持了一个全局的简化操作最优队列，尽量保证更多的边能够互相之间比较收缩的误差，以提高简化质量。本论文方法的另一个重要贡献是借鉴了“流式网格”中对索引网格形式顶点的终结信息的判断，结合体数据生成的特点来指导顶点离开生成边界的判断，从而使得简化过程变得简洁明确、内存占用较小且具有可预测性。本论文使用了不同体数据来测试模型的运行时间和简化效果，并测试了不同大小的缓存对他们的影响。经测试，本论文的算法能够在较短时间内产生类似于高质量内存简化的结果。

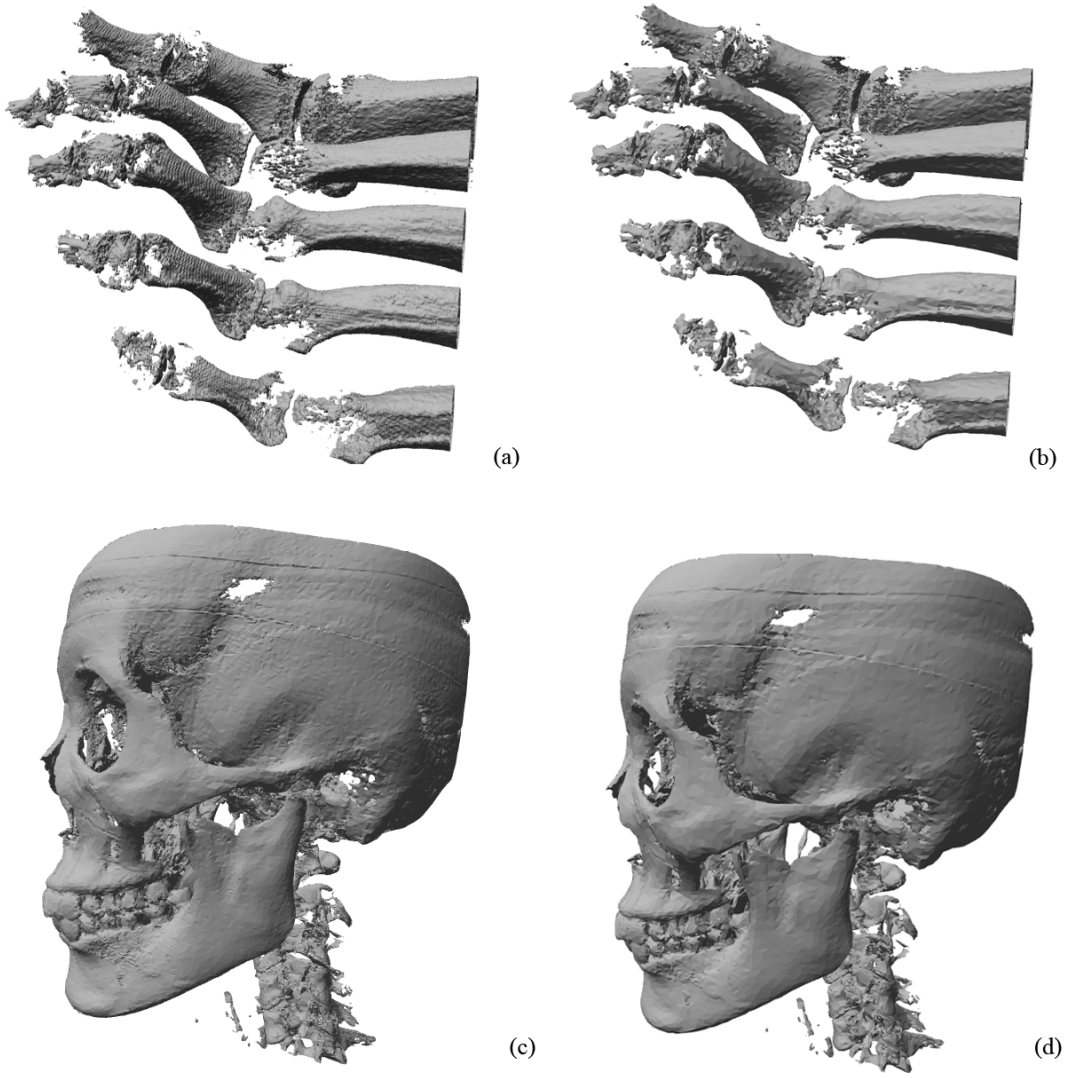


图 3-6 部分模型简化结果。(a)：Foot 模型的原始网格 (b)：Foot 20%简化后的网格 (c)：Head ZMW 10%简化后的网格 (d)：Head ZMW 5%简化后的网格

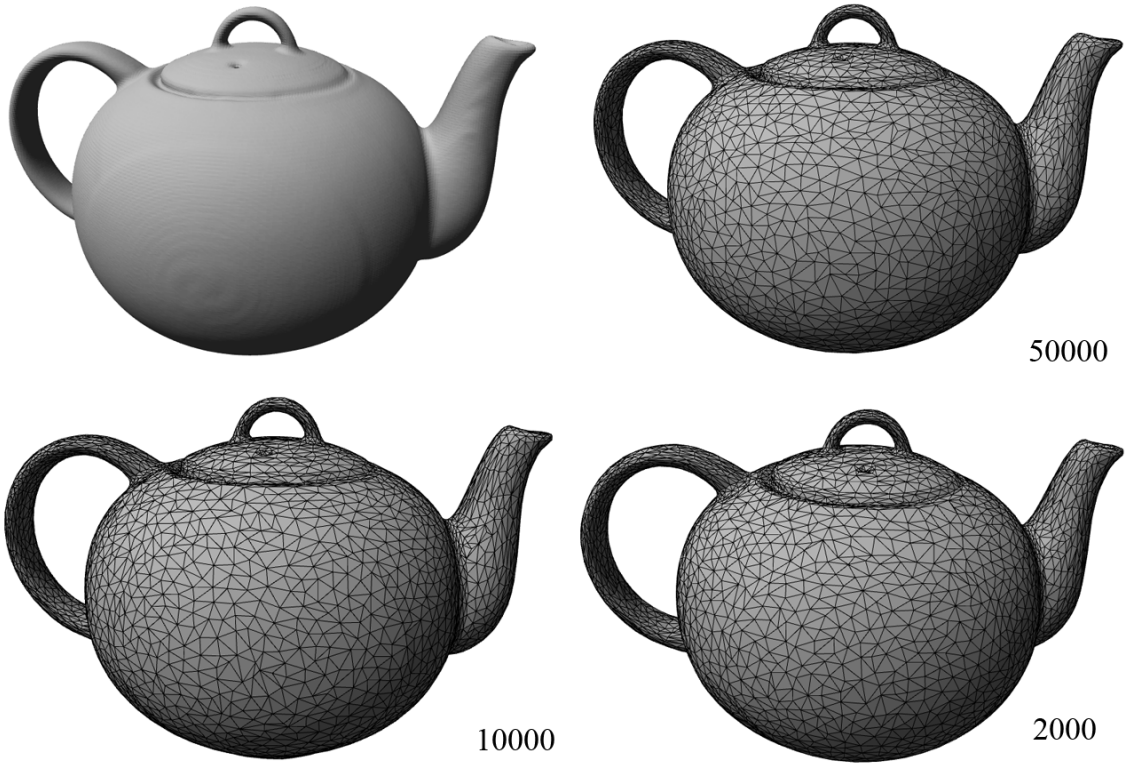


图 3-7 Boston Teopot 模型使用不同大小缓存的简化结果。右上角为原始模型，其他为简化后的模型。每个简化模型下方标注了使用的缓存大小。

第 4 章　基于八叉树单形分割的并行等值面生成

大部分自适应等值面的生成算法无法保证生成的网格流形且无自交。比如，直接对八叉树节点进行原始曲面（primal contouring）生成的算法[31, 41, 42]会在不同大小的八叉树节点交界处产生碎裂（crack）。虽然碎裂修补（crack patching）能够一定程度上解决碎裂产生的视觉瑕疵，但是得到的网格依然存在大量的非流形区域，这严重影响了网格的质量。而使用对偶曲面（dual contouring）[32-34]或对偶体素（dual grid）[35]的算法会产生网格的自交以及局部的非流形。有一种串行的基于体数据单形分割[36]的算法能够解决这个问题。本章的算法在原先串行单形分割算法的基础上，对分割方法做了简化以减少体数据函数近似产生的误差，并对它做了并行的改进，充分的利用了 GPU 的并行粒度，使得运行速度取得大幅度增长。

4.1　算法概述

本章工作的主要贡献是提供了一套完整的基于 GPU 并行的八叉树建立与最小边查找的方法。复杂度为 $O(\lg^2 n)$，n 为体数据中体素的个数。我们使用了自顶向下分层建立八叉树的方法，对每一层的节点并行计算它的对偶点、展开节点的误差以及孩子节点的个数，并使用并行的 scan[43]来计算每个节点对应子节点的偏移量。在查找最小边的时候，我们计算当前一层边的分割数目，并使用并行的数组分割来分割最小边以及可以继续被分割到下一层的边。同时我们将当前一层八叉树节点所展开的新边加入到下一层待分割的边中。使用这样分层建立的方法，我们的算法能够最大限度地利用 GPU 的并行计算资源。

在算法的最后阶段，我们依据每一个最小边来对八叉树节点进行分割。首先对最小边所邻接的最小面进行分割，分割方法是计算当前面的对偶点并将对偶点与最小边连接组成三角形。接下来将分割后的三角形与八叉树节点中的对偶点连接组成四面体。对分割的四面体使用 Marching Tetrahedron 就能得到体数据的等值面。

算法的流程如图 4.1 所示。

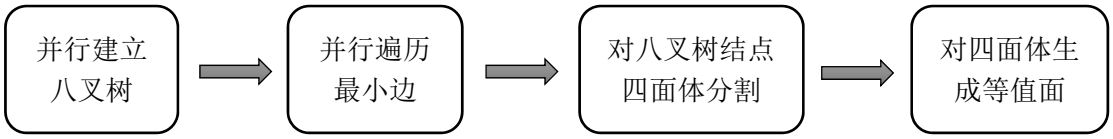


图 4.1 并行自适应等值面生成算法的流程

4.2 自适应八叉树的四面体分割

首先我们考虑在一个自适应的八叉树体数据中如何能够得到流形且无自交的等值面。一个自适应的八叉树体数据是指，在八叉树中：(1).每个结点要么具有八个子节点，要么没有子结点。所有子结点对父结点平均分割；(2).每个结点的顶点都对应原体数据的一个采样点；(3).所有结点要么是原始体数据中的单元立方体体素，要么包含它；(4).所有叶子结点完全充满了整个体数据。

我们使用二维平面中的四叉树来说明这个问题。如图 4.2，在这个四叉树中，共有三层结点。第一层最右下的结点没有孩子，其他均有四个孩子，构成了 12 个第二层的结点。第二层节点中四个节点有孩子，构成了 16 个第三层的结点。第三层由于已经是原始体数据的体素，故不可以再分割。这个四叉树共有 25 个叶子节点，第一层有一个，第二层有 8 个，第三层有 16 个。这些叶子结点共同铺满了整个体数据空间。在不同层次的体数据相交的地方，高层次结点与低层次结点相邻的边会被分割，造成高层次结点不再是一个四边形，而变成多边形。

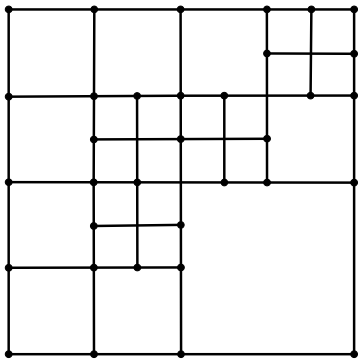


图 4.2 二维空间的自适应四叉树

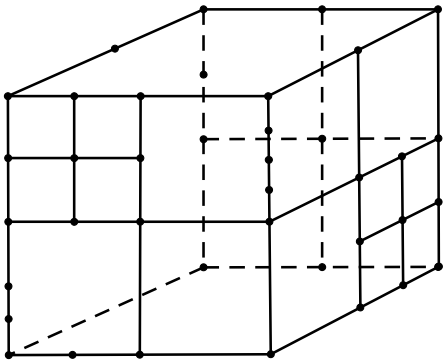


图 4.3 八叉树的高层次结点被分割

三维空间中的八叉树可以此类推。在不同层次的八叉树结点相交的地方，高层次结点的边会被分割，使得面变成多边形。同时面会被分割，使得高层次结点变成多面体。一个八叉树中高层次结点被分割的例子如图 4.3 所示。图中，这个结点变成了多面体，而有些面则变成了多边形。面的分割来源于与这个结点以面相邻的低层次结点，而边的分割来源于与这个结点以边相邻的低层次结点。

如果对类似图 4.3 的结点直接使用 Marching Cubes 或者 Marching Tetrohedron，将会带来生成网格的拓扑结构不一致性。因为它简单地把这个节点当成了一个六面体，而没有考虑它与其他低层次六面体的邻接问题。由于在原始曲面（primal countouring）生成算法中，每一个体素的边会对应生成网格中的点，因此，必须保证对体数据分割所得的非结构化体数据本身的邻接关系是正确的。

我们可以针对不同层次的多面体生成等值面，但是由于多面体的边和面数不一定，拓扑关系的查找表很难建立。因此我们考虑对不同层次结点所对应的多面体进行分割。在三维空间，一个最简单的多面体是四面体，它被称为三维单形。在二维空间中，一个最简单的多边形是三角形，他被称为二维单形。我们使用类似论文[36]中的方法，首先对面进行三角形分割，然后再对体进行四面体分割。与论文[36]中方法不同的是，我们并没有对边进行分割。因为这样并不会影响网格拓扑结构的正确性，而对边进行分割会增加算法的复杂性。另外，由于对元素进行分割需要生成元素的对偶点，而对偶点的计算需要对体数据进行估计，这会增加生成等值面的误差。对八叉树做四面体分割的好处是，使用 Marching Tetrahderon 算法生成等值面的查找表比较小，易于实现。

在对面进行分割的时候，首先对每一个面生成一个对偶点。这个对偶点是一个四维向量(x, y, z, w)，其中 x, y, z 为三维空间的坐标，w 为体数据的密度值。将对偶点与面的每一个边相连，就形成了多个三角形。对图 4.3 中朝向外面的面进行三角形分割的结果如图 4.4 所示。

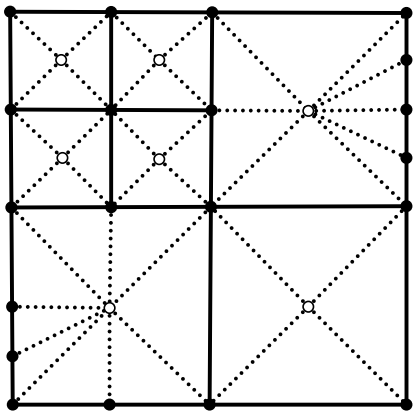


图 4.4 图 4.3 中朝向外的面的三角形分割。白色的点为对偶点，点线为对偶点与八叉树中顶点的连线。

对体进行分割也类似。先对每一个多面体生成一个对偶点，然后将每个面分割所得的三角形与这个对偶点相连，得到四面体。

需要注意的是，为了保证得到的四面体不会退化或者超出所在八叉树结点而导致生成网格的自交，必须将对偶点限制在所对应的元素范围之内。

4.3 对偶点的求取

在求取对偶点的时候，我们使用了论文[35]所提出的四维空间中的二次误差方程（Quadric Error Function）。它类似于 2.5 节所叙述的三维空间中的二次误差度量

（Quadric Error Metric），试图求取某个对偶点使得它与所在四维空间中采样点的切平面距离的平方和最小。

对于体数据，我们可以将它理解为三维空间中的点与一个标量密度值的映射函数：

$$w = f(x, y, z) \qquad (4\text{-}1)$$

这个函数定义了一个四维空间中的曲面

$$f(x, y, z) - w = 0 \qquad (4\text{-}2)$$

体数据本身可以理解为对这个四维空间的曲面，在三维空间的某个立方体内进行均匀采样所得。为了让这个对偶点能够最优地表示它所包含的所有体素，需要让它距离所有采样点的切平面的距离之和最小。对于某个采样点(x_1, y_1, z_1, w_1)的四维切平面，其法向为：

$$(\frac{\delta f}{\delta x}, \frac{\delta f}{\delta y}, \frac{\delta f}{\delta z}, 1) \qquad (4\text{-}3)$$

根据论文[13]，偏导的求取可以近似使用：

$$\begin{aligned} \frac{\delta f}{\delta x} &= \frac{f(x_1+1, y_1, z_1) - f(x_1-1, y_1, z_1)}{2} \\ \frac{\delta f}{\delta y} &= \frac{f(x_1, y_1+1, z_1) - f(x_1, y_1-1, z_1)}{2} \\ \frac{\delta f}{\delta z} &= \frac{f(x_1, y_1, z_1+1) - f(x_1, y_1, z_1-1)}{2} \end{aligned} \qquad (4\text{-}4)$$

因此，采样点的切平面可以表示为：

$$\frac{\delta f}{\delta x}(x - x_1) + \frac{\delta f}{\delta y}(y - y_1) + \frac{\delta f}{\delta z}(z - z_1) + w - w_1 = 0 \qquad (4\text{-}5)$$

我们使用类似于 2.5 节中的 5x5 对称矩阵来表示某个点与采样点距离的平方和，并求解一个 4x4 矩阵 A 和一个 4 维向量 b 组成的线性方程组 $Ax=b$ 来得到对偶点的值。

在实际实现中，我们使用了高斯消元法来解线性方程组，因为它对存储空间的要求较低，能够节省 GPU 运行过程中的寄存器占用。当方程组解不定的时候，我们对所有采样点做平均来获得对偶点。对于欠定方程组有一种更好的方法是使用奇异值分解（singular value decomposition，SVD）[27]，这种方法会在后续对算法的优化中被考虑进去。

在采样点选取的时候，为了避免采样所有点造成的复杂度过高，我们只对八

叉树结点在每个方向均匀做三次采样，因此共有 27 个采样点。我们的算法只对这 27 个采样点进行 QEF 的计算。

4.4 八叉树的建立

在建立八叉树之前，我们需要对数据进行预处理。4.2 节中所述的八叉树需要体数据在每一个方向上的体素个数相等且都为 2 的幂次，而实际的体数据有些无法满足这个条件。

我们将体数据中最大的维度 dim_m 提取出来，

$$dim_m = \max(dim_x, dim_y, dim_z) \qquad (4\text{-}6)$$

并求取出大于这个维度值最小的 2 的幂次 d，

$$d = 2^{\lceil \log dim_m \rceil} \qquad (4\text{-}7)$$

然后以 d 为三个维度上的尺寸组成一个包含了原始体数据的外层体网格。算法将原始的体数据移动到外层体网格的中心，并记录每个八叉树结点在外层体网格中的三维坐标。由于有些外层体网格中的结点是空的，所以我们在实现中对每一层八叉树结点在每个维度上保存了在外层体网格的开始和结束的坐标。

对于体数据边界上的八叉树结点，它可能只占据所在外层体网格结点的一部分。因此它展开后的子结点个数可能不为 8，有可能是 1、2 或 4。在编码过程中需要对它加以特殊考虑。

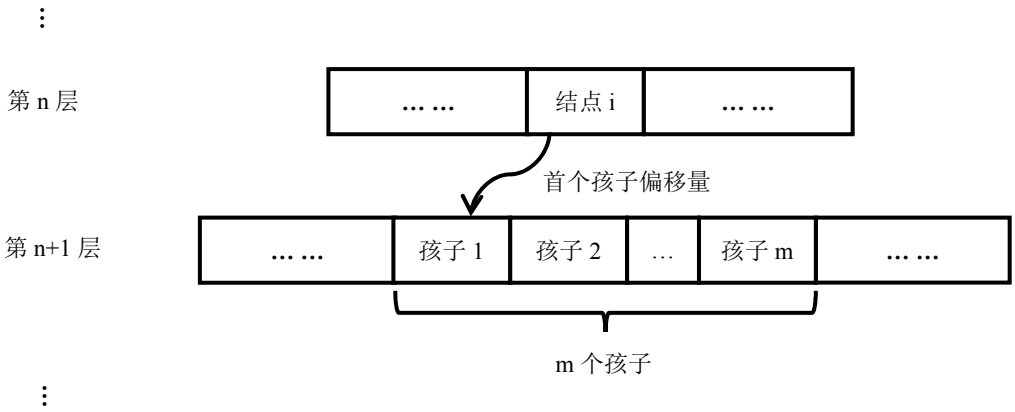


图 4.5 八叉树结点的存储结构

由于八叉树的建立过程是完全基于 GPU 并行的，因此八叉树结点的存储方式不同于普通内存中的八叉树[32]，也不同于在内存中建立好八叉树再传到显存中的方法[44]。我们的八叉树结点的存储方式类似于论文[45]，按分层存储的方式，每一

层的结点连续的存储在一个显存数组之中。每一层中拥有共同父亲的结点在数组中是连续的，因此父亲结点只需要存储孩子结点在下一层数组中的起始偏移量以及孩子结点的个数（如图 4.5 所示）。叶子结点的孩子结点个数字段为 0，首个孩子结点的偏移量字段无效。八叉树结点的数据结构定义如图 4.6 所示。

```
struct OctNode {
    unsigned short cube_index[3];    // 当前层次外层体网格中的坐标
    unsigned char child_config;      // 子结点配置信息
    struct {
        float x, y, z, w;
    } dual_vert;                     // 对偶顶点
}
```

图 4.6 八叉树结点的数据结构

结点的记录中保存了一个 `child_config` 子结点配置信息，它能帮助后续的处理快速的得到子结点在不同维度上的分割个数以及局部坐标。我们并没有显式的存储某个结点所拥有的孩子个数，因为它可以通过 `child_config` 快速计算出来。`child_config` 是一个无符号的 `char` 型变量，它的 0-1、2-3、5-6 位分别存储了 x、y、z 维度上结点分割的信息。我们将某个维度上的子结点分割情况分为占据左边、占据右边和全部占据三种情况，用两个 bit 就能表示。对于没有孩子的叶子结点，这个 `child_config` 字段被置为 0。对于首个孩子结点的偏移量，我们使用一个单独的数组 `child_addr` 来存储，这样能够方便对它的计算。

算法 4.1 建立八叉树下一层结点

```
BUILD-NEXT-LEVEL(level)
1   for each node in nodeptr[level] in parallel
2     node.child_config ← GET-CONFIG(node)
3     node.dual_vert ← GET-DUAL(node)
4     error ←node.dual_vert * QEM
5     if error greater than threshold
6       child_addr_ptr[level][node.idx] ← GET-COUNT(node.child_config)
7     else
8       child_addr_ptr[level][node.idx] ← 0
9   perfrom SCAN on child_addr_ptr[level] in parallel
10  for each node in nodeptr[level] in parallel
11    child_count ← GET-COUNT(node.child_config)
12    for each i from 0 to child_count-1
13      nodeptr[level+1][child_addr_ptr[level][node.idx]+i] ← GET-CHILD(node, i)
```

在已得到某一层八叉树结点的情况下，我们的算法需要三步并行 kernel 调用来得到下一层结点：(1).对当前层的每一个结点并行地计算 child_config 字段、对偶顶点和展开的误差，并将子结点个数存储在 child_addr 数组中；(2).对 child_addr 数组执行并行的 scan 求得子结点的偏移量；(3).根据孩子的偏移量，并行地对每一个当前层的结点生成下一层的孩子结点。其中，scan 又称 all-prefix-sum，是计算数组中所有元素的之前元素的某种运算结果的算法。并行 scan[43]可以在 O(lg *n*)复杂度之内计算出整个数组的结果。整个过程的伪代码如算法 4.1 所示。

为了能够让用户控制生成的粒度同时防止由于计算结点展开误差造成的体数据细节的丢失，我们让用户指定一个开始遍历的层次，而不是每次都从第 0 层（体数据的包围盒）开始。我们在建树的第一步先生成所有开始层次的结点，然后迭代地调用算法 4.1 生成所有其他的层次。

4.5 最小边的查找

在建立八叉树之后，我们的算法遍历八叉树中所有的最小边。最小边是八叉树中不能被分割的边，它对四面体的生成至关重要。最小边的存储结构和八叉树一样，也是按照分层存储，每一层最小边连续地存储在一个显存数组中。

边的数据结构中存储着边的方向、边所邻接的四个八叉树结点的层次和索引。我们将边按照方向分为平行于 x 轴、平行于 y 轴、平行于 z 轴。边的数据结构定义如图 4.7 所示。边所在的层次是这条边的四个结点中最低层的那个结点所在的层次。

```
struct OctEdge {
    EdgeDir dir;                              // 边方向
    int level1, level2, level3, level4;      // 结点所在层次
    int node1, node2, node3, node4;           // 结点在层次数组内索引
}
```

图 4.7 边的数据结构

边的结点位置如图 4.8 所示，它所邻接的四个节点两两之间共组合成四个面，分别为(node1, node2)，(node2, node3)，(node3, node4)，(node4, node1)。如果有两个相邻的结点实际上为同一结点，那么这两个结点的字段会被置为同一个结点的索引，而相应的面也会被排除。

假设已知某一层次结点相邻的所有边，我们称它为待分割边。从当前一层待分割边生成下一层待分割边的方法是：对当前层次的待分割边执行一次分割个数的检查，再将数组中分割个数为零的边（当前层次的最小边）与分割个数不为零

的边执行一次并行的数组分割。然后将连续地存储在数组开头的当前层次的最小边复制到最小边数组中。同时我们对分割个数不为零的边执行分割操作，存储到下一层的待分割边的数组中。下一层待分割边还包含那些当前层次的八叉树结点在展开的时候所引入的新边。这些展开八叉树节点所引入的新的边有的被包含在八叉树结点之内，有的被包含在面上（如图 4.9）。

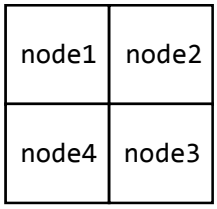


图 4.8 边相邻结点的位置

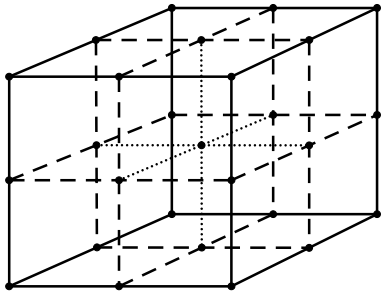


图 4.9 八叉树结点展开所引入的新边。虚线为在面上的新边，点线为结点内的新边。

对于八叉树结点之内的新边，由于已存储了每一层的八叉树结点，算法先并行地求得每一个八叉树结点所展开的新边个数，并对这个数组做一次并行 scan 求出展开结点新边的个数以及偏移量。然后算法求出八叉树结点的新边并根据偏移量将新边写入待分割边的数组。

```
struct OctFace {
    FaceDir dir;            // 面方向
    int level1, level2;     // 结点所在层次
    int node1, node2;       // 结点在层次数组内索引
}
```

图 4.10 面的数据结构

为了能够求出面所包含的边，必须得到每一层的面。我们将这些面定义为待分割面。面的数据结构存储了面的方向和面所邻接的两个结点所在的层次和索引。面的方向分为平行于 xy 平面，平行于 yz 平面，平行于 xz 平面。面的数据结构定

义如图 4.10 所示。

和待分割边一样，为了产生下一层的待分割面，我们的方法需要对当前层待分割面计算每个面在下一层的分割个数，并将边分割的执行结果填充到下一层面的数组中。由于我们的算法不需要记录最小面，因此不需要对它使用数组分割。下一层的待分割面还包含当前层的结点展开所引入的新面。我们将这两个部分组和得到下一层的待分割面。

加入面中新引入边的方法是，先算出每个面由于分割所引入新边的个数并使用并行 scan 算出偏移量，然后将新边填入下一层的待分割边数组。因此，下一层待分割边的数组有三部分组成，分别是上一层分割的边，上一层结点展开引入的边，上一层面分割引入的边。分割某一层边的伪代码如算法 4.2 所示。

算法 4.2 分割某一层边

```
1   并行地获取当前层次边的分割个数
2   并行地根据分割个数分割边数组，分割数为 0 的最小边在前面
3   将最小边存储到当前层次最小边的数组中
4   并行地对分割数不为 0 的边的分割数做 SCAN，得到偏移量
5   并行地获取每个当前层结点展开所引入新边、新面的个数
6   并行地对结点展开引入新边个数做 SCAN，得到偏移量
7   并行地对结点展开引入新面个数做 SCAN，得到偏移量
8   并行地获取每个面分割所引入的新边数、新面数
9   并行地对面分割引入新边个数做 SCAN，得到偏移量
10  并行地对面分割个数做 SCAN，得到偏移量
11  在显存中开辟下一层待分割边数组
12  把三部分的下一层待分割边填入到数组中
13  在显存中开辟下一层待分割面数组
14  把两部分的下一层待分割面填入到数组中
```

在算法开始的时候，我们生成开始层次上的所有边和面，然后不断迭代地执行算法 4.2 直到找到所有层次的最小边。

4.6 四面体与等值面生成

在找到所有最小边后，我们的算法并行的对每个最小边生成四面体。生成的方法是将最小边与所邻接面的对偶点相连生成三角形，再将三角形与面所邻接结点的对偶点相连生成四面体。

我们首先计算最小边生成的所有四面体所生成的三角形的个数，并对它执行

并行 sacn 以得到偏移量和体数据生成面的个数。随后我们在显存中开辟数组，并并行地将每一条边生成的三角形加入到三角形数组中。

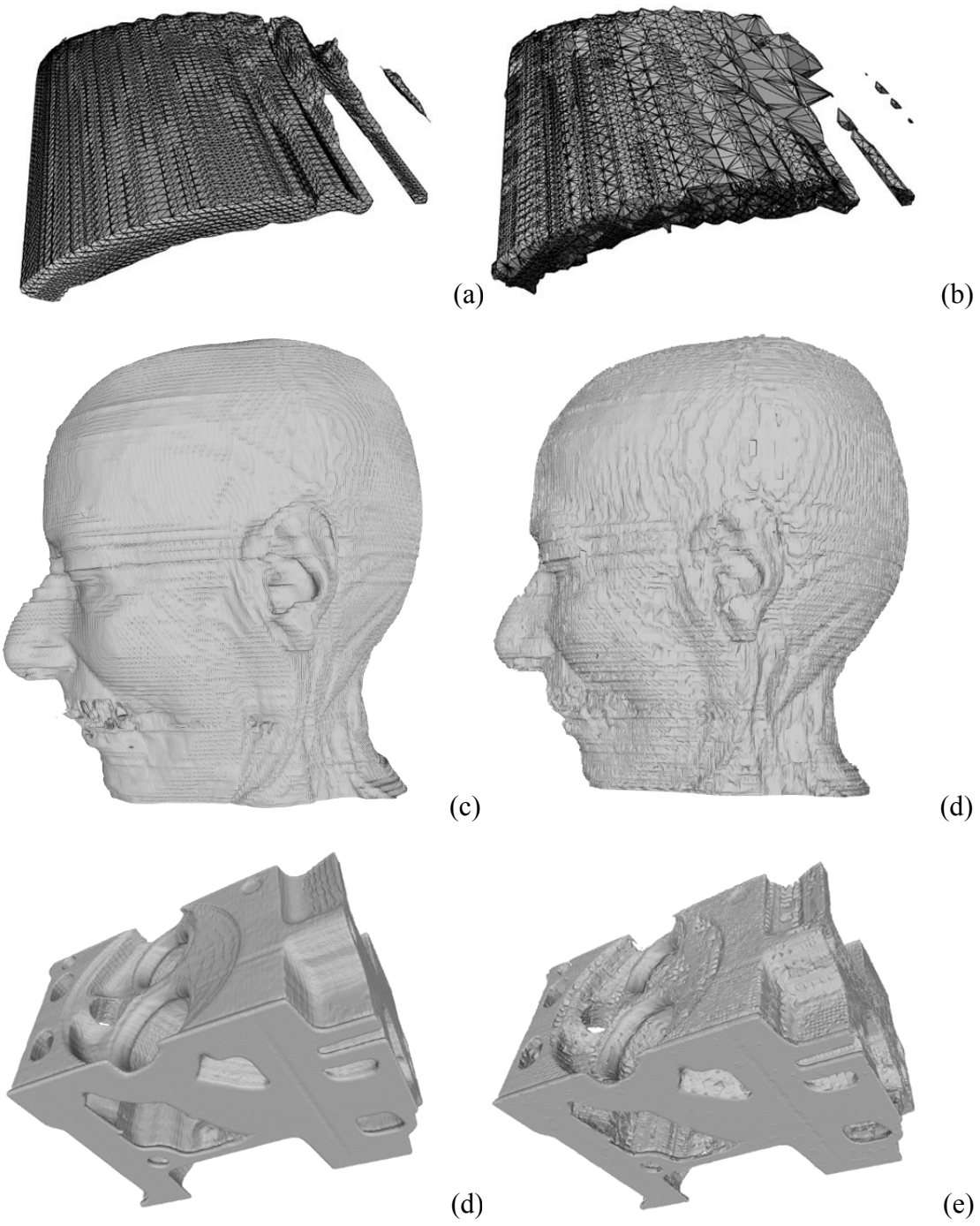


图4.11 部分模型生成的等值面 (a)：Blunt 模型使用 Marching Tetrahedron 生成的等值面 (b)：Blunt 模型使用本文算法生成的等值面 (c)：CT_256x256x106 模型使用 Marching Tetrahedron 生成的等值面 (d)：CT_256x256x106 模型使用本文算法生成的等值面 (e)：Engine 模型使用 Marching Tetrahedron 生成的等值面 (f)：Engine 模型使用本文算法生成的等值面

4.7 执行结果

表 4-1 程序编写和测试所使用的设备和平台

设备或平台	描述
CPU	Intel(R) Core(TM)2 Duo CPU，主频 2.53Hz
GPU	NVIDIA 的 GeForce 9800GT，显存容量为 1G
内存	容量为 2G
操作系统	Windows 7 Ultimate
开发语言	CUDA，C++，界面部分使用 Qt
开发环境	Visual Studio 2010
其他引用库	OpenGL 3.2，Thrust 1.6.0

表 4-2 不同模型的执行时间与对串行简化算法的加速比

模型	尺寸	Iso	Sl①	Et②	面数
Teapot	256x256x178	60	5	4000	293042
Engine	256x256x110	127.5	5	1000	1044735
CT_	256x256x106	100	6	103500	183630
HKugel	64x64x64	100	3	990	32592
blunt	256x128x64	90	3	5000	9206

表 4-2（续） 不同模型的执行时间与对串行简化算法的加速比

模型	执行时间（秒）					加速比
	建树	最小边	生成面	总计	Gs③	
Teapot	0.0803	0.0547	0.166	0.426	13.198	30.98
Engine	0.118	0.125	0.523	0.866	16.614	19.18
CT_	0.102	0.0574	0.191	0.443	4.352	9.82
HKugel	0.0259	0.0106	0.0242	0.141	0.406	2.88
blunt	0.0345	0.0376	0.0673	0.219	0.343	1.57

本研究及实验的软硬件环境如表 4.1 所示。我们对不同模型分别使用了并行的自适应等值面生成算法和第三章所介绍的大规模等值面即时简化算法做简化，并测试了并行算法对串行算法的加速比，其执行时间和执行结果如表 4.2、图 4.11 所

① Sl：Starting Level，建树的开始层次
② Et：Error Threshold，误差阈值
③ Gs：Generation Simplification，指使用第三章的即时简化的运行时间

示。在测试中，即时简化算法所使用的缓存大小统一设置为 20000。

从执行结果可以看到，对于较大规模的数据，并行算法的加速比较大，效果较为明显。另外从模型的简化效果图中可以看出，本算法对较平滑的曲面具有较好的生成效果，而对变化比较剧烈、特征点较多的曲面拟合效果不够理想。这是对偶点对体数据函数的近似误差所导致的。

4.8 本章小结

大部分自适应等值面生成算法无法保证生成的网格流形且无自交，但有一种串行的基于对体数据单形分割的算法能够解决这个问题。本论文在原先串行的单形分割算法的基础上，对分割方法做了简化以减少体数据函数近似产生的误差，更重要的是，本论文对它做了并行的改进，充分地利用了 GPU 的并行粒度，使得运行速度取得大幅度增长。我们提供了一套完整的基于 GPU 并行的八叉树建立与最小边查找的方法。本论文依据每一个最小边来对八叉树节点进行四面体分割，然后对分割出的四面体使用 Marching Tetrahedron 得到体数据的等值面。在实际测试中，本论文的算法能够产生质量较高的自适应等值面，并对大规模数据获得了对串行算法而言较高的加速比。

第 5 章 总结与展望

5.1 总结

本文的主要工作如下：

(1) 基于对网格中顶点而非三角形进行分片来简化大规模网格。这个方法能兼顾大数据访问效率和简化质量，比之前同类型的方法需要更少的处理次数，且能处理输出网格无法载入内存的情况。经测试表明，本论文的算法能够在较短的运行时间之内提供和内存简化算法相比拟的简化质量。

(2) 针对大规模体数据生成的等值面过大而无法装入内存的情况，提出了一种生成一部分网格即进行简化的算法。它在内存中保持了一个全局的简化操作最优队列，尽量保证更多的边能够得到比较，并结合体数据生成的特点来指导算法获知顶点何时离开生成边界，从而使得简化过程内存占用较小且具有可预测性。

(3) 提出了一种基于八叉树节点单形分割的完全并行的等值面生成算法。这种方法的好处是不会在不同大小的八叉树节点的交界处产生碎裂，也不会存在网格的自交以及局部的非流形。本论文算法的主要贡献是提供了一套完整的基于 GPU 并行的八叉树建立与最小边查找的方法。由于本文的算法充分地利用了 GPU 的并行粒度，使得运行速度取得大幅度增长。

5.2 展望

虽然本文对大规模网格简化、等值面即时简化以及自适应等值面生成等题目给出了一些可行的方案，并取得了较为满意的结果。但本文的方法依然存在一些问题。

我们希望大规模网格分片算法能够更好地保持全局简化操作的执行顺序，而我们的算法将边界边的收缩排除了在外。同时，“分片-简化-合并”的过程具有一定的不确定性，还是稍稍逊于纯粹的内存简化。

另外，使用“建立八叉树-最小边查找”的 GPU 单形分割算法需要保存太多的中间数据结构，对内存的占用较大，这给算法的通用性提出了挑战。同时，对体数据函数的预估增加了生成等值面的误差，我们希望在日后的工作中能找到避免或减少这种误差、优化生成网格质量的方法。

所有这些问题，我们期待通过进一步的研究工作，来获得解答或者突破。

参考文献

[1] Levoy M, Pulli K, Curless B. The digital Michelangelo project: 3D scanning of large statues. Proceedings of the 27th annual conference on Computer graphics and interactive techniques, 2000: 131-144.

[2] Bernardini F, Rushmeier H, Martin I M. Building a digital model of Michelangelo's Florentine Pieta. Computer Graphics and Applications, IEEE, 2002, 22(1):59-67.

[3] Rossignac J R, Borrell P. Multi-resolution 3D approximations for rendering complex scenes. Geometric Modeling in Computer Graphics, 455–465. Springer, Berlin (1993).

[4] Low K-L, Tan T-S. Model simplification using vertex-clustering. Proceedings of the 1997 symposium on Interactive 3D graphics, 1997: 75-81.

[5] Luebke D, Erikson C. View-dependent simplification of arbitrary polygonal environments. Proceedings of the 24th annual conference on Computer graphics and interactive techniques, 1997: 199-208.

[6] Hinker P, Hansen C. Geometric optimization. Proceedings of the 4th conference on Visualization'93, 1993: 189-195.

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

[8] De Fioriani L, Magillo P, Puppo E. Building and traversing a surface at variable resolution. Visualization'97. Proceedings, 1997: 103-110.

[9] De Floriani L, Magillo P, Puppo E. Efficient implementation of multi-triangulations. Visualization'98. Proceedings, 1998: 43-50.

[10] Hamann B. A data reduction scheme for triangulated surfaces. Computer Aided Geometric Design, 1994, 11(2):197-214.

[11] Garland M. Quadric-based polygonal surface simplification. Phd thesis, Carnegie Mellon University, 1999.

[12] Schroeder W J, Zarge J A, Lorensen W E. Decimation of triangle meshes. ACM SIGGRAPH, 1992: 65-70.

[13] Lorensen W E, Cline H E. Marching cubes: A high resolution 3D surface construction algorithm. ACM SIGGRAPH, 1987: 163-169.

[14] Ronfard R, Rossignac J. Full-range approximation of triangulated polyhedra. Computer Graphics Forum, 1996: 67-76.

[15] Lindstrom P, Turk G. Fast and memory efficient polygonal simplification. Visualization'98. Proceedings, 1998: 279-286.

[16] Lindstrom P, Turk G. Image-driven simplification. ACM Transactions on Graphics, 2000, 19(3):204-241.

[17] Hoppe H. Progressive meshes. Proceedings of the 23rd annual conference on Computer graphics and interactive techniques, 1996: 99-108.

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

[19] Hoppe H, Derose T, Duchamp T. Mesh optimization. Proceedings of the 20th annual conference on Computer graphics and interactive techniques, 1993: 19-26.

[20]Hoppe H. Smooth view-dependent level-of-detail control and its application to terrain rendering. Visualization'98. Proceedings, 1998: 35-42.

[21] Prince C. Progressive meshes for large models of arbitrary topology. Master's thesis, University of Washington, 2000.

[22] Brodsky D, Pedersen J B. Parallel model simplification of very large polygonal meshes. Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, 2002: 1207-1215.

[23] Brodsky D, Watson B. Model simplification through refinement. Graphics Interface, 2000: 221-228.

[24] Cignoni P, Montani C, Rocchini C. External memory management and simplification of huge meshes. Visualization and Computer Graphics, IEEE Transactions on, 2003, 9(4):525-537.

[25] El-Sana J, Chiang Y J. External Memory View-Dependent Simplification. Computer Graphics Forum, 2000: 139-150.

[26] Lindstrom P. Out-of-core simplification of large polygonal models. Proceedings of the 27th annual conference on Computer graphics and interactive techniques, 2000: 259-262.

[27] Press W H, Flannery B P, Teukolsky S A. Numerical recipes in C: the art of scientific programming. Section, 1992, 10:408-412.

[28] Lindstrom P, Silva C T. A memory insensitive technique for large model simplification. Proceedings of the conference on Visualization'01, 2001: 121-126.

[29] Wu J, Kobbelt L. A stream algorithm for the decimation of massive meshes. Graphics interface, 2003: 185-192.

[30] Isenburg M, Lindstrom P. Streaming meshes. Visualization, 2005. IEEE, 2005: 231-238.

[31] Shekhar R, Fayyad E, Yagel R. Octree-based decimation of marching cubes surfaces. Visualization'96. Proceedings, 1996: 335-342.

[32] Ju T, Losasso F, Schaefer S. Dual contouring of hermite data. ACM Transactions on Graphics, 2002, 21(3):339-346.

[33] Schaefer S, Ju T, Warren J. Manifold dual contouring. Visualization and Computer Graphics, IEEE Transactions on, 2007, 13(3):610-619.

[34] Ju T, Udeshi T. Intersection-free contouring on an octree grid. Pacific graphics, 2006.

[35] Schaefer S, Warren J. Dual marching cubes: Primal contouring of dual grids. Computer Graphics and Applications, 2004: 70-76.

[36] Manson J, Schaefer S. Isosurfaces over simplicial partitions of multiresolution grids. Computer Graphics Forum, 2010: 377-385.

[37] Luebke D, Reddy M, Cohen J D. Level of detail for 3D graphics. Morgan Kaufmann, 2003.

[38] Mattson R L, Gecsei J, Slutz D R. Evaluation techniques for storage hierarchies. IBM Systems journal, 1970, 9(2): 78-117.

[39] Cignoni P, Rocchini C, Scopigno R. Metro: measuring error on simplified surfaces. Computer Graphics Forum, 1998: 167-174.

[40] The Stanford 3D Scanning Repository. http://www-graphics.stanford.edu/data/3Dscanrep/

[41] Müller H, Stark M. Adaptive generation of surfaces in volume data. The Visual Computer, 1993, 9(4):182-199.

[42] Shu R, Zhou C, Kankanhalli M S. Adaptive marching cubes. The Visual Computer, 1995, 11(4):202-217.

[43] Sengupta S, Harris M, Zhang Y. Scan primitives for GPU computing. Proceedings of the 22nd symposium on Graphics Hardware, 2007: 97-106.

[44] Lefebvre S, Hoppe H. Compressed random-access trees for spatially coherent data. Proceedings of the 18th Eurographics conference on Rendering Techniques, 2007: 339-349.

[45] Zhou K, Gong M, Huang X. Data-parallel octrees for surface reconstruction. Visualization and Computer Graphics, IEEE Transactions on, 2011, 17(5):669-681.

致　谢

衷心感谢导师陈莉副教授在读研三年期间给我的各种学习和生活上的指导和帮助。她平易近人的个性、认真工作的态度和严谨求实的科研精神值得我终身学习。同时还感谢计算机图形与辅助设计研究所的雍俊海教授、张慧副教授、王斌副教授三位老师在实验室科研学习过程中给予的指导和督促。

还要感谢可视化小组的邱庆默、彭艺、刘晓强和吴侃等同学，这篇论文从程序部分的编写到论文部分的完成，都离不开你们的细心帮助和鼓励。 在三年的研究生学习生活中，我不断得到同学们的关心与帮助，这里也要感谢所有陪我一起度过人生中最重要的这三年的清华的同学们！另外，还要特别感谢我的家人，是你们一直给予我各方面的关怀和支持，让我不断的成熟和进步。

本课题承蒙国家自然科学基金重大研究计划项目资助，也特此致谢。

声　明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签　名：_____日　期：_____

个人简历、在学期间发表的学术论文与研究成果

个人简历

1988 年 9 月 16 日出生于辽宁省大连市金州区。

2006 年 9 月考入北京理工大学软件学院软件工程专业，2010 年 7 月本科毕业并获得工学学士学位。

2010 年 9 月免试进入清华大学软件学院攻读软件工程硕士至今。