INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA

UMI[®]
800-521-0600

Mesh Simplification in Parallel

by

Christian Langis, B.Sc.

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of

Master of Computer Science

Ottawa-Carleton Institute for Computer Science

School of Computer Science

Carleton University

Ottawa, Ontario

August 1st 1999

© copyright

1999, Christian Langis

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file *Votre référence*

Our file *Notre référence*

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-48438-6

Canada

The undersigned hereby recommend to the Faculty of Graduate Studies
And Research acceptance of the thesis,

Mesh Simplification in Parallel

Submitted by

Christian Langis

In partial fulfillment of the requirements
for the degree of Master of Computer Science

Thesis Supervisor

Thesis Co-Supervisor

Director, School of Computer Science

Carleton University
October 25, 1999

Abstract

Christian Langis

CARLETON UNIVERSITY - OTTAWA, 1999

Under the supervision of: Gerhard Roth & Frank K.H.A. Dehne

In this thesis, the author presents a method to simplify computer graphics meshes in parallel. Meshes have been processed in parallel before, but rarely in an optimal way. Meshes are today's most popular computer graphics model. Current technologies allow production of meshes whose size exceeds the hardware and software capability to display them conveniently. To answer this issue, a recently new mesh operation emerged. This operation, called mesh simplification, reduces a mesh down to a simpler expression which is much faster to render. The high quality of reduced meshes generated by our mesh simplifier comes to a price: CPU time. Hence, one way to yield faster computations is to parallelize the procedure.

We parallelized the simplification procedure by dividing the meshes between processors. And by tuning our algorithm for maximum speedup, we were able to produce a parallel algorithm (and implementation) optimal in execution time. That is, its execution time is inversely proportional to the number of processors it uses. This optimality has been our main contribution. As far as we know, the specific mesh simplification procedure has never been implemented in parallel

This thesis comprises two separate yet related topics: mesh simplification and graph partitioning. The author deals with each in turn, from theory to practice. The thesis begins with a complete study of different mesh simplification methods focusing on one of particular interest that led to a previous sequential implementation. The author studies this implementation in great detail and devises a way to parallelize it. Then he changes his focus to a broad study of different graph partitioning methods. After selecting one that best suits the application needs, he implements and tests it thoroughly. Finally, he combines both topics into a parallel mesh simplifier, algorithm and implementation, fully tested and analysed.

Table of Contents

Title Page	i
Acceptance Sheet	ii
Abstract	iii
Table of Contents	iv
List of Tables	vii
List of Figures	viii
CHAPTER 1: Introduction	1
1.1 Meshes	1
1.2 Mesh simplification	2
1.3 Mesh partitioning	4
1.4 Parallel mesh simplification	5
CHAPTER 2: Meshes in Computer Graphics	7
2.1 Mesh production	8
2.2 Some definitions	9
2.3 Mesh formalism	11
2.4 Mesh attributes	13
CHAPTER 3: Mesh Simplification	14
3.1 Goals in surface simplification	16
3.2 Simplification methods	17
3.3 General simplification framework	20
3.4 An overview of mesh decimation	22
3.4.1 A generic mesh decimation algorithm	23
3.4.1.1 Characterizing the local vertex geometry/topology	25
3.4.1.2 Evaluating the decimation criteria	23
3.4.1.3 Triangulation	27
3.5 Mesh optimization	28
3.5.1 Definition of the energy function	29
3.5.2 Minimization of the energy function	31
3.5.2.1 Optimization for fixed simplicial complex	32
3.5.2.2 Optimization over simplicial complexes	34
3.5.3 Improvements exploiting locality	35
3.6 Progressive Mesh representation	37
3.6.1 Preserving Attributes	40
3.6.2 Overview of the PM procedure	41
3.6.3 Geomorphs	43
3.6.4 Progressive transmission	45
3.6.5 Mesh compression	45
3.6.6 Selective refinement	48

3.7 Summary & Discussion	49
CHAPTER 4: Mesh Partitioning	51
4.1 Graph partitioning background	53
4.2 Recursive methods	54
4.2.1 Recursive bisection	54
4.2.2 Spectral methods	56
4.2.3 Geometric methods	59
4.2.3.1 In practice	61
4.2.3.2 Discussion	62
4.3 Other partition-related algorithms	63
4.3.1 Multilevel method	63
4.3.1.1 Coarsening step	64
4.3.1.2 Uncoarsening step	65
4.3.1.3 Discussion	66
4.3.2 Optimization methods	68
4.4 Greedy methods	70
4.4.1 An intuitive start	71
4.4.2 Ciarlet's algorithm	75
4.4.2.1 Discussion on connectivity	79
4.4.3 Analysis	81
4.4.3.1 First (intuitive) algorithm tests	85
4.4.3.2 Second algorithm (greedy) tests	89
4.4.3.3 Third algorithm (revised greedy) tests	92
4.4.3.4 Comparison of algorithms	96
4.5 Conclusion	98
CHAPTER 5: Parallel Mesh Processing	101
5.1 Parallelism at large	102
5.1.1 Different kinds	102
5.1.1.1 Functional parallelism	102
5.1.1.2 Temporal parallelism	103
5.1.1.3 Data parallelism	103
5.1.2 Parallel algorithm concepts	104
5.1.2.1 Coherence	104
5.1.2.2 Task/data decomposition	105
5.1.2.3 Granularity	105
5.1.2.4 Scalability	106
5.1.3 Design & implementation issues	107
5.2 Different alternatives	108
5.2.1 Fine-grain	108
5.2.2 Coarse-grain	109
5.3 Our version	111
5.3.1 How does it meet the parallel paradigm?	113
5.3.2 Border problem	116

5.4 Conclusion	118
CHAPTER 6: Implementation, Tests & Analysis	120
6.1 Implementation	120
6.1.1 Sequential implementation	121
6.1.2 Parallel extension	125
6.2 Tests & analysis	133
6.2.1 Timing analysis	133
6.2.2 Quality analysis	139
6.3 Improvements	142
6.4 Summary & conclusion	145
CHAPTER 7: Conclusion	147
Bibliography	149

List of Tables

4.1	Partitioning test results on the Bunny models	85
4.2	Partitioning test results on the Duck models	85
4.3	Partitioning test results on the Dragon models	86
4.4	Partitioning test results on the Elephant, Grapple models	86
4.5	Partitioning test results on the Bunny models	89
4.6	Partitioning test results on the Duck models	89
4.7	Partitioning test results on the Dragon models	90
4.8	Partitioning test results on the Elephant, Grapple models	90
4.9	Partitioning test results on the Bunny models	92
4.10	Partitioning test results on the Duck models	92
4.11	Partitioning test results on the Dragon models	93
4.12	Partitioning test results on the Elephant, Grapple, Nefertiti models	93
6.1	Parallel Duck simplification statistics	135
6.2	Parallel Dragon simplification statistics	135

List of Figures

2.1	Vertex and edge stars	10
3.1	Topological operations	20
3.2	Local mesh geometry	24
3.3	Plane evaluation for mesh decimation	25
3.4	Triangulation	27
3.5	Mesh accuracy/size chart	36
3.6	Simplification/refinement operation	38
3.7	Vertex split operation	44
4.1	An exploded view of a 8-way partition of the NRC Duck	73
4.2	Bunny (Surfaces)	82
4.3	Bunny (Full Wireframe)	82
4.4	Duck (Surfaces)	82
4.5	Duck (Full Wireframe)	82
4.6	Dragon (Surfaces)	82
4.7	Dragon (Full Wireframe)	82
4.8	Elephant (Surfaces)	83
4.9	Elephant (Full Wireframe)	83
4.10	Grapple (Surfaces)	83
4.11	Grapple (Full Wireframe)	83
4.12	Nefertiti (Surfaces)	83
4.13	Nefertiti (Full Wireframe)	83
5.1	Edge-cut face deletion	117
6.1	Merging of collapsed faces in PM	132
6.2	Duck in Progressive Mesh version at different resolutions	141

Chapter 1

Introduction

This thesis is the symbiosis of two different computer science problems. One, mesh simplification, aims at optimizing mesh shape, storage size and rendering time. The other, graph partitioning, which at first appears unrelated, explores ways to divide graphs into sub-graphs. Both will be necessary to derive a parallel mesh simplifier.

1.1 Meshes

Nowadays, meshes are the most popular computer graphics model in use. They are widespread throughout society, whether in business, science or entertainment. The mesh model itself is very simple. A mesh consists of a set of triangles, adjacent to each other by their edges. This group of adjacent triangles forms a surface. The surface represents any 3D object. The number of triangles in the mesh determines its resolution. The more triangles there are in the mesh, the smoother the surface appears. The demand for

high-quality, high-resolution meshes yields bulky mesh over 10 million faces (gigabyte mesh files). Chapter 2 discusses meshes in computer science in both practical and theoretical terms.

1.2 Mesh simplification

Surface simplification deals with the approximation of a surface (mesh) with another surface of lower triangle count. Although this field is young, there are already many algorithms to carry out this task (see Section 3.1-3.4). We have implemented a sequential Mesh Optimization method [Hop93] which generates excellent quality mesh approximations. This quality however, comes at the cost of increased execution time.

Mesh Optimization uses the edge collapse operator (see Figure 3.1) as its topological operation applied locally to edges of the mesh to simplify it. The edge collapse eliminates two faces from the mesh, therefore reducing the mesh resolution and visual quality. But different edges, once collapsed, affect the mesh quality differently. That is, they introduce geometric error of different magnitude. For this reason, Mesh Optimization assigns a collapse cost to every edge under the form of an Energy Function. This Energy Function minimizes the error involved with an edge collapse and computes a cost associated with it. Then once a cost is assigned with every edge, all the potential edge collapses are sorted on that cost. Mesh Optimization performs the edge collapses one after another in the order of the lowest to highest cost collapses (see Section 3.5).

Mesh Optimization manages to generate simplified versions of the initial mesh, which are a good fit to this initial mesh. But isn't it a shame to discard all the successive states of the mesh through the execution of the different edge collapse operations, and then save only the coarsest representation of the mesh? The originator of the Mesh Optimization algorithm also made this observation, and proposed a new mesh format, the Progressive Mesh [Hop96]. This is a mesh format which stores the coarsest version of the mesh generated by Mesh Optimization along with all the edge collapse operations, ordered chronologically from the first to the last to be executed (see Section 3.6). Schematically, we represent it as $PM = M^0 \leftrightarrow M^1 \leftrightarrow M^2 \leftrightarrow \dots \leftrightarrow M^n$. This mesh format allows the user to first see on the screen the coarse version of the mesh (M^0) which is faster to render than M^n due to its reduced face count. And then, as the user desires, the mesh can be rendered in any resolution up to M^n .

The edge collapse has an inverse operation called the vertex split (see Figure 3.6). Therefore, if the user wants to view the current mesh at a lower resolution, the edge collapse list will be traversed downwards (towards M^0). If the user wants increased resolution, the edge collapse list will be traversed upwards (towards M^n) by performing vertex splits.

Therefore, with the Progressive Mesh format, it is possible to quickly access all mesh resolutions from the highest to the lowest. The Progressive Mesh solves the problems associated with huge uni-resolution meshes: excessive rendering time,

transmission time, and memory usage. Furthermore, the format lends itself naturally to display features such as geomorphs and faster rendering by selective refinement (see Section 3.6.3 and 3.6.6). Finally, due to the Mesh Optimization algorithm, the Progressive Meshes maintain very good visual quality even when the resolution is reduced to as low as 25% of the original mesh on average.

1.3 Mesh partitioning

Meshes are very common data objects in today's computer industry. Graphs, a similar object in computer science, have a huge body of research as well. Graphs have proven useful in many practical applications, such as networks, for example. In our application, we map the problem of mesh partitioning into the problem of graph partitioning. Abstractly, the graph partitioning problem is the division of the corresponding graph into subgraphs.

Partitioning a graph generally amounts to dividing the vertices of the graph into disjoint subsets of approximately equal size with as few edges as possible joining them. This is an NP-Hard problem. Therefore, the different partitioning methods we surveyed all rely on heuristics to yield approximations to the optimal partition. In Chapter 4, we presented three families of partitioning algorithms: greedy, geometric and spectral.

The greedy method uses a graph traversal technique to gather vertices to form the different vertex subsets of a partition (see Section 4.4). The geometric methods instead, use only the vertices of the mesh as input. They perform different geometric transformation on the mesh vertices before separating them geometrically with planes (see Section 4.2.3). The spectral methods use only the connectivity between vertices to minimize the edge-cut size represented by a function. This function can be minimized by finding the eigenvector of the Laplacian Matrix of the mesh (see section 4.2.2). These methods all have different characteristics. We decided to aim at speed, choosing the fastest method regardless of the quality of the partitions. Therefore, we partitioned the mesh using the greedy method, which is the fastest of the three.

1.4 Parallel mesh simplification

We seem to have already defined the basis of our parallel algorithm. But parallel design should be performed the other way around. First we have to identify what kind of parallelism our application falls into. How to divide the data between the processors, is a particularly sensitive issue. The granularity of the algorithm must be identified. The scalability must be enforced. And the use of coherence is important in facilitating the goals of any parallel algorithm.

In Chapter 5, we discuss these issues in the context of our application, parallel mesh simplification, and we present our parallel algorithm. In Section 5.3.2, we discuss the

partition border problem. The graph partitioning method applies to any problem which maps onto graphs and is solved in parallel. But the partition border problem is solved differently for each application. In our case, the border problem has been solved to maximize program execution by minimizing the inter-process communication. Our solution to the border problem enables our application to have a linear speedup, which is optimal. To our knowledge, no such parallel implementation of continuous mesh creation exists in the literature.

Finally in Chapter 6, we first reveal some technical details about the sequential mesh simplifier. Then from that basis, we discuss our parallel implementation. We then describe a set of experiments which compare both implementations (sequential and parallel) in terms of speed and Progressive Mesh quality.

Chapter 6 closes with a discussion of the possible improvements to the current parallel implementation. In Chapter 7, we conclude and summarize with the contributions we brought to the field of mesh simplification and discuss the future of Progressive Meshes.

Chapter 2

Meshes in Computer Graphics

As a result of growing expectation for greater realism in computer graphics, increasingly detailed geometric models are becoming commonplace. Within traditional modeling systems, highly detailed models are created by applying versatile modeling operations (such as extrusion, constructive solid geometry and freeform deformations) to a vast array of geometric primitives (B-splines, implicit surfaces...). However, for the purpose of display efficiency, these models must normally be converted into polygonal approximations, meshes [Hop96]. In fact, polygons have always been a popular graphics primitive for computer graphics applications. Besides having a simple representation, computer rendering of polygons is widely supported by commercial graphics hardware and software [Sc92]. Contemporary graphics packages directly rely on triangle meshes as a universal surface representation.

In the simplest case, a mesh consists of a set of vertices and a set of faces. Meshes can be embedded into any dimension, two or over (this thesis deals exclusively with 3D

meshes). Each vertex specifies the (x, y, z) coordinates of a point in 3D space, and each face defines a non-intersecting polygon by connecting together an unordered subset of the vertices with edges. Although the polygons may in general have an arbitrary number of vertices, we consider in this work the special case of meshes, triangle meshes, in which all faces have exactly three vertices. This does not constitute a restriction for arbitrary meshes since they all can be converted to triangle meshes through repeated triangulation operations [Hop98]. From now on in this thesis, a mesh will be assumed to be a triangle mesh and a face to be a triangle (unless stated otherwise).

2.1 Mesh production

Meshes were synthesized in the past, using mathematical primitives and operations. Recently however, a new technique greatly enhanced the mesh generation tool set. Just as it has been possible to scan 2D documents, now there exists hardware to scan 3D objects. Indeed, automatic acquisitions of the 3D object's surface are emerging (e.g. 3D range scanners) and their very high precision infers very complex meshes [Ciam]. The Biris family of laser range cameras developed at the NRC is an example of such technology. The main objective of this synchronized laser scanner development is the realization of a versatile high resolution 3D digitizer. Registered color digitizing (X, Y, Z, R, G, B) is also a feature of one of the laboratory prototypes [NRCa]. Typically, those machines are built to crop large amounts of 3D data on the surface of an object with range sensors (such sensors are also called geometric sensors because they can directly capture the

geometry of an object). In general, an optical source such as a laser is used to obtain the distance to the object's surface. Another less popular technique is the X-ray tomography that retrieves the cross sections of the object. These scanners acquire data in different ways (whole images, 3D profiles, object slices or points). A typical situation is when 3D data is produced as an unordered set of 3D points. This raw cloud of points has no connectivity information from the scanned object. An algorithm must be applied on this set of points to create a triangle mesh, i.e. add a set of faces over the set of points [Roth]. These initial triangulations are typically generated by preprocessing algorithms. Those algorithms make a best attempt to link vertices together. To do so, they rely on the set of points and heuristics rather than on additional information about the object's surface.

2.2 Some definitions

A triangle mesh is a 3D triangulated surface $S(\{v_i\}, \{t_j\})$ consisting of a set of vertices V and a set of triangles T , each triangle defined by three vertex indices. Those vertices may be ordered in two ways. Viewing the object from outside, the vertices can be listed (in each face) in a clockwise or counter-clockwise manner. Usually, for the sake of software efficiency, triangle vertices are all stored with the same orientation. Edges are the links between two vertices which both belong to at least one triangle. Intuitively, a triangle mesh may be thought of as a number of triangles pasted together along its edges.

The set of triangles that share a vertex v is called the star of v , $*(v)$. We also define here the edge star $*(v_1, v_2)$ where v_1 and v_2 are the edge endpoints (stars are also called neighborhoods in the literature). The edge star is the set of all triangles that meet v_1 or v_2 ; it is the union of $*(v_1)$ and $*(v_2)$. The cardinal (number of faces) of the star is called the valence of the vertex v , $\nu(v)$. The boundary of the star is called the link, $l(v) = \{\forall e(v_1, v_2) \in *(v) \mid v_1 \neq v \text{ and } v_2 \neq v\}$. The link can be seen as a polygonal curve made by linking up boundary edges around v or v_1 and v_2 (the figure below shows a vertex star and an edge star, with links in bold lines). A *manifold* vertex has a link formed of a simple polygonal curve; otherwise it is *non-manifold*. A vertex with an open link is a boundary vertex (it stands on a mesh boundary).

Figure 2.1: Vertex and edge stars

A necessary and sufficient condition for a mesh to be manifold is that each of its vertices is a manifold vertex. The mesh surface is closed if each of its edges is shared by exactly two triangles or equivalently if each triangle has three triangle neighbors. In a manifold surface, a pair of neighbor (adjacent) triangles share exactly one edge. Two triangles have the same orientation if their two common vertices are listed in opposite order (in the mesh data structure). We say that a surface is oriented if all of its triangles

have the same orientation. From now on, we will assume meshes to be oriented and manifold [Guez].

2.3 Mesh formalism

A mesh could be defined by its two main aspects: connectivity and geometry. Formally we introduce elegant notions of algebraic topology; a mesh M is a pair (K, V) . K is a simplicial complex representing the connectivity of the vertices, edges and triangles, thus determining the topological type of the mesh; whereas $V = \{v_1, \dots, v_n\}$, $v_i \in \mathbb{R}^3$ is a set of vertex positions defining the shape of the mesh in $\mathbb{R}^3$, its geometry, its shape.

A simplicial complex K consists of a set of vertices $\{1, \dots, n\}$, with a set of non-empty subsets of the vertices called the simplices of K . The 0-simplices $\{i\} \in K$ are single vertices, the 1-simplices $\{i, j\} \in K$ are edges and 2-simplices $\{i, j, k\} \in K$ are faces (triangles). In general n -simplices are polygons with $n+1$ vertices.

A geometric realization of a mesh M as a surface in $\mathbb{R}^3$ is obtained as follows. For a given simplicial complex K , we form a topological realization $|K|$ in $\mathbb{R}^n$ by identifying the vertices $\{1, \dots, n\}$ with the standard basis vectors $\{e_1, \dots, e_n\}$ in $\mathbb{R}^n$. Say s is a simplex of K and $|s|$ denotes the convex hull in $\mathbb{R}^n$ of the vertices of s . Hence, $|K|$ is the union of

those convex hulls, $|K| = \bigcup_{s \in K} |s|$. Let $\phi: \mathbb{R}^n \rightarrow \mathbb{R}^3$ be the linear map that assigns the i^{th} standard basis vector $e_i \in \mathbb{R}^n$ to $v_i \in \mathbb{R}^3$. Now, the geometric realization of M is the image of $\phi_V(|K|)$ (where we write ϕ_V instead of ϕ to stress that the basis vector is defined by the vertex position $V = \{v_1, \dots, v_n\}$). The map ϕ_V is an embedding if it is not self-intersecting (all vertices in V have different 3D coordinates). Only a restricted set V can make ϕ_V an embedding. If that is so, any point $p \in \phi_V(|K|)$ can be parameterized by a unique pre-image on $|K|$. This pre-image is the vector $b \in K$ (with $p = \phi_V(b)$) called the barycentric¹ coordinate vector of p with respect to the simplicial complex K . Clearly, barycentric coordinate vectors are combinations of standard basis vectors $e_i \in \mathbb{R}^n$ corresponding to the vertices of the triangles of K . Any barycentric coordinate vectors, for such point p , has at most three non-zero entries. In fact, it has exactly three non-zero entries if p lies on a mesh triangle, two if p lies on a mesh edge and one if p is one of the mesh vertices [Hop93].

1- Given three (non-colinear) points A,B,C, the "barycentric coordinates" of a point P with respect to the plane defined by A,B,C are u,v,w , such that:

$$P = u \times A + v \times B + w \times C$$

$$1.0 = u + v + w$$

And if P is inside the triangle ABC, then: $0.0 \leq u,v,w \leq 1.0$.

2.4 Mesh attributes

Vertices and faces used to represent the mesh also have additional attributes related with them. *Discrete attributes* are usually associated with the faces of the mesh. One of them, the material identifier, determines the shader function used in rendering a face of the mesh as well as some of the shader function's global parameters (a simple function might be a fixed look-up table to a specified texture map). *Scalar attributes* are more often associated with the mesh. These include diffuse color (r, g, b) , normal vectors (n_x, n_y, n_z) and texture coordinates (u, v) . More generally, these attributes specify the local parameters of shader functions defined on the mesh faces. In simple cases, these attributes are associated with vertices of the mesh. However, to represent discontinuities and because faces have different attributes, it is common to assign scalar attributes to corners of faces instead of vertices only. A corner is the pair (vertex, face). Scalar attributes at corner (v, f) specify the shading parameters for face f at vertex v . Hence, discontinuities between adjacent faces can be expressed through scalar attribute differences at adjacent corners [Hop96].

Attributes are of prime importance to meshes. However, the material provided to carry on this work did not involve them. Thus we dropped the attributes and were concerned only with geometry and topology in the rest of this thesis.

Chapter 3

Mesh Simplification

Surface simplification is a very hot topic in visualization for many sophisticated graphics applications in the industry and society in general. Huge meshes are generated in a number of fields: scientific visualization, virtual reality, surface modeling, Computer Aided Design and medical imaging to name a few [Ciam]. Due to their complexity, we encounter difficulty when presenting them in an interactive environment (typical rendering software display meshes on a polygon-by-polygon basis. Therefore, the time to render a fixed mesh is linear in the number of polygons). The volume of triangles of those meshes prevents them from being displayed at a reasonable rate for user interaction. Inevitably, the unfortunate side effect of the high resolution mesh generation methods is that the resulting mesh models are far too large to be interacted with in real-time environments. This is especially true in situations where expensive high-end computer graphics hardware is unavailable [Cort].

When users interact with these models, the computer must continually redraw the scene in conformity with changing visualization parameters (position, viewing direction...). When the model is large, the computer will not be able to keep up a sufficient frame rate and the motion will appear choppy. To allow interaction with these large data sets, we desire lower resolution models for faster dynamic viewing and the original high-resolution model for static viewing. To address these problems, researchers sought solutions that would allow them to maintain the perceived level-of-detail (LOD) of meshes while achieving interactive display rate. In the past, this problem has been solved by creating LOD approximations of the model by decimating the mesh to a constant number k of different resolutions representations of the n faces surface S ($k \ll n$) and choosing among them according to the viewing needs. This technique has the side effect of rough transitions when switching resolutions. The obvious solution to this problem was to create a scheme where the mesh display can be smoothly interpolated from its coarsest resolution to its full resolution (original mesh). This representation is called the continuous resolution form. It is a data structure that allows compact storage of k representations of S , where k is a function of the size of S . This representation contains all the relevant information to create a low-resolution model as well as a predefined series of steps used to incrementally add details to the mesh. This form provides more flexibility in the selection of the best LOD. In many cases, this choice is only relevant at runtime rather than during the mesh preprocessing phase [Ciam, Cort].

3.1 Goals in surface simplification

Reducing the complexity of meshes is therefore a must to guarantee interactivity in 3D model rendering. Such large meshes often require more than the available storage space and affect negatively the performance of graphics systems. Hence, the interest in mesh simplification has motivated considerable research by many groups. The general goals in mesh simplification are among others [Ciam]:

- *Approximation error.* The simplification procedure should provide the user with an estimate of how much the simplification has degraded/improved the mesh according to some mesh quality metrics.
- *Compression factor.* A reduction factor comparable or better than that of other approaches at the same level of approximation.
- *Multiresolution management.* Once the mesh has been simplified, its new continuous form (data structure) should offer interactive extraction of any LOD representation with the complexity of a single LOD extraction (not to confuse with rendering) being linear with respect with its output size.
- *Working domain.* The algorithm should not rely on the correctness of the surface. That is mesh anomalies such as self-intersecting, non-manifold, non-orientable surfaces, which are common in real-world data, should be accepted by the algorithm and correctly processed.

- *Space/Time efficiency.* Due to the size of large meshes, the simplification process should, like for any software, minimize its processing time and memory consumption.

3.2 Simplification methods

Due to the growing interest in the topic, research took many directions. This led to the design of many classes of methods. These methods all converge to the same operation; meshes are simplified either by merging their elements or by resampling their vertices. The methods distinguish themselves by how those operations are performed and also by how the error criteria are computed and used to measure the fitness of the simplification [Ciam]. Among the existing methods, we have:

- *Coplanar facet merging.* Coplanar or nearly coplanar adjacent polygons are searched for in the mesh, merged into larger polygons, and then retriangulated into fewer simple faces [Hink, Kalv]. This simple scheme does not provide deep simplification since it considers coplanarity as a sine-qua-non criterion to select candidate faces.
- *Retiling.* A smaller number of new vertices are inserted at random on the original mesh and then moved on the surface to be displaced on maximal curvature locations. Then, iteratively, the original vertices are removed and the mesh retiled [Turk]. This is a striking example of mesh resampling.

- *Mesh decimation.* Based on multiple filtering passes, this approach analyzes locally the geometry and topology of the mesh and removes vertices that pass a minimal distance or curvature angle criterion. Resulting holes are patched by triangulation [Sc92]. New decimation solutions that support global error control have been proposed [Baj, Coh, Kle]. In particular, the *simplification envelopes* method supports bounded error control by forcing the simplified mesh to lie between two offset surfaces (inner and outer envelopes). A local geometric optimality criterion was also paired with the definition of a tolerance volume to drive edge collapsing and maintain a bounded approximation [Guez].
- *Mesh optimization.* Not so different from mesh decimation, mesh optimization evaluates an energy function over the mesh and minimizes such a function either by removing/moving vertices or collapsing/swapping edges [Hop93]. Later on, mesh optimization has been encapsulated into algorithms that generate *Progressive Meshes* from initial input meshes. Progressive Mesh is a continuous resolution mesh format. It supports notably multiresolution management, mesh compression, and selective refinements [Hop96].
- *Multiresolution analysis.* This approach uses remeshing, resampling, and wavelet parametrization to build a multiresolution representation of the surface from which any approximated representation can be extracted [Hop95]. One such representation consists of a simple base mesh together with a sequence of local correction terms, called wavelet coefficients, capturing the details present in the mesh at different resolutions.

- *Vertex clustering*. Based on geometric proximity, the approach gathers vertices into clusters and computes a new representative vertex for each cluster. The method is fast, but neither topology nor shape is preserved [Ros].

A general comparison of these approaches is not easy because algorithm efficiency depends largely on the geometrical and topological structure of the test meshes and on the required results. Each method has its specialty. For example, the presence of sharp edges and rough angles would be better managed by a decimation approach, while on smooth surfaces mesh optimization would give better results. Furthermore, the superior results in the precision and conciseness of the output mesh given by mesh optimization and retiling techniques are counterbalanced by substantial processing times. Other approaches have been proposed for particular mesh occurrences. Some techniques are peculiar to volume rendering applications and are less general than the previous ones [Ciam]. For example, multiresolution analysis is restricted to meshes with subdivision connectivity, that is meshes obtained from a single base mesh by recursive 4-to-1 splitting [Hop95].

Nevertheless, mesh decimation and mesh optimization seem to be the most promising methods. This thesis is based on a specific use of mesh optimization; namely *Progressive Meshes*. Therefore, mesh optimization will be fully covered in the rest of this chapter. However, since mesh decimation is somewhat similar to mesh optimization, it will be presented next.

3.3 General simplification framework

Besides the wavelet, face merging, and retiling methods, most known mesh reduction methods iteratively reduce the input mesh. A sequence of topological operations is applied to the current mesh removing geometric entities in each step. Such basic operations are shown (Figure 3.1) in the following order: vertex removal and hole retriangulation, edge collapse and half-edge collapse.

Figure 3.1: Topological operations

The reader can observe that the edge collapse is the only operator that generates new vertices. That enlarges the list of vertices in the multiresolution representation of the mesh. However, if the new vertices are wisely crafted, they enhance the mesh quality. Nevertheless, it has been observed after testing different simplification methods on a variety of meshes, that the underlying topological operator does not have a significant impact on the results. The quality of the results turns out to be much more sensitive to where and when the reduction operation is applied on the mesh [Sw]. Nevertheless, the half-edge collapse is fast to optimize (none is needed) and to transmit (no extra vertices are generated in the mesh file).

In general, every simplification algorithm privileges one type of topological operation and uses it exclusively. Also, in general, the algorithm has some means to compare the cost of performing the operation on two different mesh entities (vertices, edges or faces). That is, the algorithm can evaluate the cost and associate a scalar value to it. Hence, naturally, the algorithm can rank the cost of the operation over all entities. In fact, it became common practice for most current implementations to use a priority queue to order the operations, executing the least destructive one at each iteration. Simplification algorithms have the following generic structure:

```

for all geometric entities {
    Measure costi of applying operator on entityi
    Put (entityi, costi) into priority queue
}

until queue empty, or user stopping condition reached {
    Perform the operation with least cost
    Update new cost of adjacent entities and the queue
}

```

The most demanding part for the potential application is measuring and updating the cost of the topological operation on the mesh entities. How the priority is calculated for every possible operation is intrinsic to each algorithm [Sw98].

3.4 An overview of mesh decimation

Methods could be divided into two categories: the ones that operate with global testing, and others with local testing. Mesh decimation (in its most standard definition) falls in the second category, which yields less expensive and faster algorithms. Indeed, mesh decimation uses local geometric optimality criteria. And mesh decimation is truly a greedy method of mesh simplification.

It has often been written that mesh simplification exists to reduce the size of meshes, thus it is necessary to reduce the amount of data by removing redundant information from the mesh. A precise definition of the term redundancy, in this context, depends obviously on the application for which the decimated mesh is to be used. From an optical point of view, *local flatness* of the mesh is a better indicator of redundancy. Coplanar adjacent faces (local flatness) would have the same appearance if they were all merged together [Sw]. Under this criterion, mesh decimation is known to decimate heavily flat regions while preserving faithfully other regions.

Under controlled, yet acceptable reductions, the result will meet two requirements. First the reduced mesh must preserve the original topology of the mesh. Second, it must represent a good geometric approximation of the original mesh. Technically speaking, the most important aspect is the approximation error, i.e., the modified mesh has to stay within a prescribed tolerance to the original data. Optionally, the vertices of the decimated mesh can be a subset of the original vertices. Hence, instead of creating new

vertices, relatively unimportant vertices (and associated faces) would be removed from the mesh. Although not essential to forming an accurate simplified mesh, this option has the major advantage that the mesh geometry is never modified by some new vertices (whose position has been evaluated for best fit rather than taken from the original mesh). Next is presented some pioneering work on mesh decimation from William Schroeder [Sc92].

3.4.1 A generic mesh decimation algorithm

The decimation algorithm is simple. Multiple passes are made over all vertices in the mesh. During a pass, each vertex is a candidate for removal and, if it meets the specified decimation criterion, the vertex and its star (all of its adjacent faces) are deleted. The resulting hole in the mesh is patched by rebuilding a local triangulation. The vertex removal process repeats, with possible adjustments of the decimation criterion, until some termination condition is met. Usually the termination condition is specified as a percentage reduction of the original mesh, or as some maximum decimation threshold. The repeated three steps of the algorithm are:

- 1- Characterize the local vertex geometry and topology.
- 2- Evaluate the decimation criterion.
- 3- Triangulate the resulting hole.

3.4.1.1 Characterizing the local vertex geometry/topology

The first step of the decimation algorithm characterizes the local geometry and topology for a given vertex. The outcome of this process determines whether the vertex is a potential candidate for deletion, and if so, which criteria to use. Each vertex falls into one of the following five categories:

Figure 3.2: Local mesh geometry

A *simple vertex* is surrounded by a complete cycle of triangles and each edge adjacent to the central vertex is adjacent to two triangles. If the edge is not adjacent to two triangles or if the central vertex is part of a triangle not in the cycle of triangles, then the vertex is *complex*. A vertex that is on the boundary of a mesh, within a semi-cycle of triangles, is a *boundary vertex*. A simple vertex can further be simplified in sub-categories. These classifications are based on the local mesh geometry. If the dihedral angle between two triangles is greater than a specified feature angle, then a feature edge exists. When a vertex has two of those, the central vertex is an *interior edge vertex*. With one, or more than two such edges, the central vertex is classified as a *corner vertex*.

3.4.1.2 Evaluating the decimation criteria

The characterization step produces an ordered loop of vertices (the link of the vertex star) and triangles adjacent to the candidate vertex. The evaluation step determines whether the triangles of the star can be deleted and replaced by another triangulation without the removed vertex. Although the fundamental decimation criterion used is based on the vertex distance d to the vertex star plane, others can be applied.

Figure 3.3: Plane evaluation for mesh decimation

Simple vertices are the most current class of vertices in meshes in general. In fact it is the only possible class present in our ideal, manifold-preserving mesh model. Hence, for simplicity's sake, that case only will be considered. Simple vertices use the distance to plane. An average plane is constructed using the triangles of the vertex star. All triangles yield three variable values used to compute the vertex star decimation plane. A triangle has a normal to its plane $\vec{n}_i$, a 3D center point $\vec{x}_i$, and an area size A_i . Therefore, the average plane normal $\vec{n}$ and center point $\vec{x}$ (of the vertex star) are calculated as:

$$\vec{N} = \frac{\sum \vec{n}_i A_i}{\sum A_i} \quad \vec{n} = \frac{\vec{N}}{|\vec{N}|} \quad \vec{x} = \frac{\sum \vec{x}_i A_i}{\sum A_i}$$

The distance of the vertex $\bar{v}$ to the plane is then $d = |\bar{n} \cdot (\bar{v} - \bar{x})|$. If the vertex is within a specified distance to the average plane, then it may be deleted. Otherwise, it is retained.

An algorithm may or may not take care of non-manifold vertex star cases. But if it does, it can evaluate their cost in many different ways (see Figure 3.3, boundary edge third picture).

Feature (sharp) angles may be the result of bad mesh synthesis or ‘noise’ (irrelevant to the mesh) or real geometric details of the mesh (important to preserve). In any case, if the cost of interior edge and corner vertices is calculated as it is for simple vertices, then the vertex stars with small triangles (assumed to be noise or unimportant details) will be deleted. On the contrary the big vertex stars with feature edges will be preserved since their distance will always be above the decimation criterion. Hence this simple heuristic tends to preserve surface discontinuities as the decimation is performed.

It is worthwhile to note here that the decimation criterion considers only the deviation from the previous mesh to the new one. Deviation from the original mesh is not considered. Thus there is no upper bound guarantee on the accumulated geometric error with this generic decimation strategy [Hop93].

3.4.1.3 Triangulation

Deleting a vertex, and its associated triangles, creates one loop (vertex link). Within the loop, a triangulation must be rebuilt with non-intersecting and non-degenerate triangles. It is also desirable to create individual triangles, with good aspect ratio (similar edge sizes), that approximate the original loop as well as possible.

Although other triangulation schemes can be used, a simple recursive loop splitting procedure fits naturally for triangulation. Each loop to be triangulated is split into two loops. The division is along a line (split line) defined by two non-

Figure 3.4: Triangulation

neighbouring vertices of the loop. Each loop is divided recursively down to three vertices (a triangle). The split plane is a plane than contains the split line and is orthogonal to the average star plane. Typically, each loop may be split in different ways. The best split line (or plane) must be selected. Many criteria are available, but a successful simple one is based on the aspect ratio. The aspect ratio is the distance of the closest vertex to the split plane divided by the length of the split line. The best split plane is one that yields the largest aspect ratio (ratio higher than 0.1 produce acceptable meshes).

3.5 Mesh optimization

The problem: given a set of data points scattered in 3D and an initial mesh $\hat{M}$, produce a mesh M of same topological type as $\hat{M}$ that fits the data well and has a small number of vertices [Hop93].

The pioneering work in this area was done by Hugues Hoppe whose first, yet complete, theoretical attempt has been compiled in [Hop93]. His metaphor for mesh reduction cost is encapsulated into an *energy function*.

To optimize a mesh, the algorithm must minimize the energy function that captures the competing desires of tight geometric fit and compact representation. The tradeoff between the two is controlled by a user parameter c_{rep} . The optimization process uses an input mesh $\hat{M}$ as a starting point. This non-linear process reduces the number of vertices of $\hat{M}$ and modifies their positions and connectivity.

Although mesh optimization was first intended for the surface reconstruction problem, it can also be applied to mesh simplification (in fact it became a leading method of simplification). Mesh simplification is considered here as an optimization problem with an energy function that directly measures the deviation of the final mesh from the original. As a consequence, the final mesh naturally adapts to curvature variations in the original mesh.

What will be presented next is not a simplification algorithm but rather only the optimization engine. That is, the procedures from this engine that compute the energy function and also the procedures that optimize the new vertex position (which by themselves are complicated enough).

3.5.1 Definition of the energy function

Recall that the competing goals of mesh optimization are: 1- obtain a new mesh that provides a good fit to the original mesh and 2- reduce the number of vertices. We must find a simplicial complex K and a set of vertex positions V that minimizes the energy function:

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

The first two terms correspond to the two stated goals. The distance energy E_{dist} is equal to the sum of the squared distances from the points $X = \{x_1, \dots, x_m\}$ (cropped from the original mesh $\hat{M}$) to the current approximated mesh. The representation energy E_{rep} penalizes meshes with a large number of vertices. It is set to be proportional to the number n of vertices of K .

$$E_{\text{dist}}(K, V) = \sum_{i=1}^n d^2(x_i, \phi_v(|K|)) \qquad E_{\text{rep}}(K) = c_{\text{rep}}n$$

The optimization allows vertices to be both added to and removed from the mesh. The reader however should understand that when optimization is used in the realm of simplification, then no vertices can be added to the mesh; reduction operations alone are allowed. E_{rep} acts to encourage vertex removal. The user specified parameter c_{rep} provides a controllable trade-off between fidelity of geometric fit and conciseness of representation.

But the function did not seem complete so far. That is, minimizing $E_{\text{dist}} + E_{\text{rep}}$ did not produce the desired results. After some tests, the optimized meshes had spikes in regions where there is no data. These spikes emerged from the fundamental problem that there may not be any minimum for $E_{\text{dist}} + E_{\text{rep}}$. To guarantee the existence of a minimum, E_{spring} was added to the function. It places on each edge of the mesh a spring of rest length zero and spring constant κ .

$$E_{\text{spring}}(\mathbf{K}, \mathbf{V}) = \sum_{\{j,k\} \in K} \kappa \left\| \mathbf{v}_j - \mathbf{v}_k \right\|^2$$

E_{spring} is not a smoothness penalty. It is not meant to penalize sharp angles in the mesh, since they may be present on the underlying original surface and hence, ought to be preserved. E_{spring} is rather a regularizing term that helps guide the optimization to a desirable local minimum.

3.5.2 Minimization of the energy function

The goal of the optimization is to minimize the energy function over a set of simplicial complexes, homeomorphic to the initial simplicial complex K_0 and with the vertex positions V defining the embedding. Here is an outline of the optimization algorithm:

```

OptimizeMesh( $K_0$ ,  $V_0$ ) {
     $K = K_0$ 
     $V = \text{OptimizeVertexPositions}(K_0, V_0)$ 
    repeat {
        ( $K'$ ,  $V'$ ) = GenerateLegalMove( $K$ ,  $V$ )
         $V' = \text{OptimizeVertexPosition}(K', V')$ 
        if  $E(K', V') < E(K, V)$  then
            ( $K$ ,  $V$ ) = ( $K'$ ,  $V'$ )
    } until convergence
    return( $K$ ,  $V$ )
}

OptimizeVertexPositions( $K$ ,  $V$ ) {
    repeat {
         $B = \text{ProjectPoints}(K, V)$ 
         $V = \text{ImproveVertexPositions}(K, B)$ 
    } until convergence
    return( $V$ )
}

```

```

GenerateLegalMove(K, V) {
    Select a legal move  $K \Rightarrow K'$ 
    Locally modify V to obtain V'
    return(K', V')
}

```

3.5.2.1 Optimization for fixed simplicial complex (*OptimizeVertexPositions*)

Here the problem is to find a set of vertex positions V that minimizes $E(K, V)$ for a given simplicial complex K . The energy function is simplified to $E_{\text{dist}} + E_{\text{spring}}$ since E_{rep} does not depend on V .

At the beginning, the geometry of the original mesh $\hat{M}$ is recorded by sampling from it a set of points X . At a minimum, every vertex of $\hat{M}$ is sampled. Additional points on the surface of $\hat{M}$ may also be sampled randomly. To evaluate the distance energy $E_{\text{dist}}(K, V)$, it is necessary to compute the distance of each data point x_i to $M = \phi_v(|K|)$. Each of these distances is itself the solution to the problem

$$d^2(x_i, \phi_v(|K|)) = \min_{b_i \in |K|} \|x_i - \phi_v(b_i)\|^2$$

in which the unknown is the barycentric coordinate vector $b_i \in |K| \subset \mathbb{R}^n$ of the projection of x_i onto M . Thus minimizing $E(K, V)$ for a fixed K is equivalent to minimizing the new objective function

$$E(K, V, B) = \sum_{i=1}^n \|x_i - \phi_v(b_i)\|^2 + E_{spring}(K, V) = \sum_{i=1}^n \|x_i - \phi_v(b_i)\|^2 + \sum_{\{j,k\} \in K} \kappa \|v_j - v_k\|^2$$

over the vertex positions $V = \{v_1, \dots, v_n\}$, and the barycentric coordinates $B = \{b_1, \dots, b_{|X|}\}$. To solve this optimization problem, the method alternates between two subproblems:

- 1- For fixed vertex positions, find optimal barycentric coordinate vectors by projection.
- 2- For fixed barycentric coordinate vectors, find optimal vertex positions by solving a linear least square problem.

Optimal solutions to both subproblems are always found, hence $E(K, V, B)$ never increases. And since it is bounded from below, it must converge (for more technical and theoretical details, see [Hop93]).

A modification has been brought subsequently to the equation when this optimization is used for simplification. This was necessary since the declining number of vertices in M affected E in a biased way. Instead of defining a global spring constant κ for E_{spring} , it is adapted each time an edge collapse is considered. Intuitively, the spring energy is most important when few points project onto a neighborhood of faces, since in this case finding the vertex positions minimizing E_{dist} may be an under-constrained problem. Thus κ is set as a function of the ratio of the number of points x_i to the number

of faces in the edge neighborhood (edge star) of the current mesh approximation M . With this adaptive scheme, the influence of E_{spring} decreases gradually as the mesh is simplified.

3.5.2.2 Optimization over simplicial complexes (*OptimizeMesh*)

To solve the outer minimization problem, three topology operators are considered: *edge collapse*, *edge split* and *edge swap*, taking a simplicial complex K to another K' . However, when optimization is used for mesh simplification, edge collapse only is applicable since only this operator reduces the number of faces on a mesh.

A legal move is the collapse of one edge of K that will not change the mesh topological type. Hence an edge collapse has to be tested for legality. The collapse of edge $\{i, j\} \in K$ transforming K into K' is legal if the following conditions are satisfied (extensive proof in [Hop93]):

- For all vertices $\{k\}$ adjacent to both $\{i\}$ and $\{j\}$, $\{i, j, k\}$ is a face of K .
- If $\{i\}$ and $\{j\}$ are both boundary¹ vertices, $\{i, j\}$ is a boundary¹ edge.
- K has more than four vertices if neither $\{i\}$ nor $\{j\}$ are boundary vertices, or K has more than three vertices if either $\{i\}$ or $\{j\}$ are boundary vertices.

The goal then is to find a sequence of legal moves taking the original simplicial complex K_0 to a minimum of $E(K)$. A brute-force sequencing method might be the

‘random descent’ i.e. $E(K)$ decreases by unordered decrement values. Legal moves $K \Rightarrow K'$ are randomly selected. The move is accepted if $E(K') < E(K)$, otherwise another move is selected. After many subsequent rejected moves, the algorithm terminates. Of course, much more elaborated selection strategies are possible (the priority queue being one of them). Indeed, the list of legal moves can be sorted in any way to fit specific application needs.

3.5.3 Improvements exploiting locality

This idealized algorithm is by far too computationally intensive to be useful. However, due to coherence and locality, some improvements can speed up the heuristics dramatically. These advances are based on the fact that a legal move exclusively affects the star of the collapsed edge (the neighborhood where the edge collapse occurred).

l- An edge $\{i, j\} \in K$ is a boundary edge if there is only one $\{k\} \in K$ such that the face $\{i, j, k\} \in K$. And a vertex $\{i\}$ is a boundary vertex if there is a boundary edge $\{i, j\}$.

For example, the procedure *OptimizeVertexPositions* is by far the costliest procedure, carrying heavy computations. Furthermore, it is called very often; in the order of the number of edges of (K, V) . This procedure estimates the effect of a legal move. We know very well though that edge collapses are local reductions. It is then pointless to minimize the difference in the equation E with unchanged data. Hence, a modified heuristic is applied only to a submesh, in the neighborhood of the transformation along with the subset of the data points projecting onto the submesh. The change of energy is estimated by only considering the contribution of the submesh and the corresponding point set.

Secondly, when collapsing edge $\{i, j\}$, the algorithm considers the edge star and optimizes over the new vertex $\{k\}$. For efficiency's sake, few iterations are allowed. And for optimization's sake, the initial choice of v_k is critical. Hence, to comply with both principles, three optimizations are performed, with v_k being v_i , v_j , and $\frac{1}{2}(v_i+v_j)$ and the best one is accepted.

As a final note on optimization, Figure 3.5 shows the measured accuracy of simplified meshes with respect to its resolution. The curve shows the highest expected upper bound on the ratio accuracy/conciseness for arbitrary models.

Figure 3.5: Mesh accuracy/size chart

We can observe that the geometric accuracy coincides closely with empirical measures of quality, e.g. the perceived mesh quality for a given resolution. This realization has a profound impact on LOD approximation using continuous-resolution models since it provides a statistical guideline that allows users to exercise precise control over the rendered object and the performance of the rendering engine [Lit].

3.6 Progressive Mesh representation

Mesh optimization was presented first because it is a building block of a wider application scope, continuous-resolution representations. Mesh optimization refines a mesh in regions of interest and coarsens it where data is redundant. That mesh operation is therefore very flexible and can be adapted to any specific mesh requirement. In this thesis, we use mesh optimization to simplify meshes and then store them into a continuous-resolution representation. Therefore, we consider mesh optimization as the mesh processing engine that will help produce not only one optimized representation M of a mesh $\hat{M}$ but also a whole family of n increasing LOD representations $M^0, \dots, M^n$. Once the mesh has been simplified and optimized, it can be stored into a much more useful mesh format, as a *Progressive Mesh*. The PM format addresses other mesh related problems such as *LOD approximation*, *progressive transmission*, *mesh compression* and *selective refinement*.

In a PM representation, an arbitrary mesh $\hat{M}$ is stored as a much coarser mesh M^0 together with a sequence of n detail records that indicate how to incrementally refine M^0 exactly back into the original mesh $\hat{M} = M^n$. Each of these records stores the information associated with a vertex split (the dual operation for the edge collapse), an elementary mesh transformation that fits an additional vertex to the mesh. The PM representation of $\hat{M}$ thus defines a continuous sequence of meshes $M^0, M^1, \dots, M^n$ of increasing accuracy, from which LOD approximations of any desired complexity can be efficiently retrieved.

Figure 3.6: Simplification/refinement operation

The quality of the intermediate approximations M^i depends largely on the algorithm for selecting which edges to collapse and what attributes to assign to the affected neighborhoods, especially the position of the new vertex v_i . There is a variety of such algorithms with varying tradeoffs of speed and accuracy. At one extreme, a fast brute-force method might be to select at random the candidates for collapse. More sophisticated methods make use of heuristics to improve the selection strategy like Schroder's *distance to plane* metric for example [Sc92], or the *energy function* presented earlier [Hop93].

As in mesh optimization, PM also defines an explicit energy metric $E(M)$ to measure the accuracy of simplified meshes with respect to $\hat{M}$.

$$E(M) = E_{\text{dist}}(M) + E_{\text{spring}}(M) + E_{\text{scalar}}(M) + E_{\text{disc}}(M)$$

The two new terms $E_{\text{scalar}}(\mathbf{M})$ and $E_{\text{disc}}(\mathbf{M})$ are added to preserve attributes associated with $\hat{\mathbf{M}}$. E_{scalar} measures the accuracy of the scalar attributes and E_{disc} measures the geometric accuracy of the discontinuity curves. The energy formula is calculated independently for each of the three dimensions in space. The sum of their cost becomes the total cost of performing the collapse.

The optimization engine used in PM differs from [Hop93] a bit further. It relies on the edge collapse exclusively since this topological alone can contribute to simplify a mesh. Edge swaps and splits, useful in the context of surface reconstruction and/or optimization, are not essential for simplification. In fact it has been observed that the edge collapse transformation alone can reduce a mesh. And when it is coupled with the priority queue, it produces meshes of similar quality. Moreover, the use of one transformation only simplifies the implementation (improves performance), and most importantly, paves the way for the PM representation.

Instead of randomly performing edge collapses, all candidates for collapse are inserted in the priority queue, ranked by their energy cost $E\Delta$. At each iteration, the best candidate (edge) standing on top of the queue is selected. Its collapse is performed, and the priority (energy cost) of all edges in the neighborhood are recomputed (due to local geometry transformation). As a consequence, the term c_{rep} (as well as energy term E_{rep}) is eliminated since in mesh simplification, there is no more a user choice on the

accuracy/conciseness balance. The user would rather determine the resolution of the coarsest mesh M_0 in the PM representation (how many faces to remove from $\hat{M}$).

3.6.1 Preserving Attributes

As described earlier, continuous scalar fields on meshes are represented by scalar attributes defined at every mesh corner. The E_{scalar} term is computed after E_{dist} and E_{spring} have been used to determine the position of the new vertex. For a vertex v_j having scalar attributes $\underline{v}_j \in \mathbb{R}^d$ this term is defined as:

$$E_{\text{scalar}}(\underline{V}) = (c_{\text{scalar}})^2 \sum_i \left\| \underline{x}_i - \phi_v(b_i) \right\|^2$$

where $\underline{x}_i$ is the scalar attribute of the associated mesh surface point. The c_{scalar} variable is used as a relative weight between attributes errors (E_{scalar}) and geometric errors (E_{dist}). Because the barycentric projections $\phi_v(b_i)$ have already been calculated from E_{dist} , calculating the E_{scalar} incurs little additional overhead. The overall effect of E_{scalar} is to choose attributes that blend naturally into the surrounding subgraph and to penalize collapses in proportion of how much they alter the attributes values [Cort].

E_{disc} measures the geometric accuracy of discontinuity curves formed by a set of sharp edges in the mesh. E_{disc} is defined as the sum of squared distances from a set X_{disc} of points sampled from sharp edges (on $\hat{M}$) to the discontinuity components from which

they were sampled (on M). Minimization of E_{disc} preserves the geometry of material boundaries and face normal discontinuities (creases) [Hop97a].

Appearance attributes give rise to a set of discontinuity curves on the mesh, both from the differences between discrete face attributes and scalar corner attributes. As these discontinuities curves form noticeable features, it has been found useful to preserve them both topologically and geometrically. When a candidate edge is to be collapsed, some simple tests on the presence of sharp edges in the edge star determine if the transformation would modify the topology of the discontinuity curve. If this is the case, then the transformation is either rejected or penalized. It has been found that the latter strategy is better since those discontinuities are sometimes too small to be visually relevant and generally they prevent thorough simplification. More details about E_{disc} and attribute energy costs in [Hop96].

3.6.2 Overview of the PM procedure

The construction of a progressive mesh may be divided into two steps: 1- generation of the initial set of edges to collapse 2- execution of those collapses. Because each collapse alters differently the geometry of the mesh, the order of candidate edges for collapse is important. The collapses are rated based on how much they modify the mesh (energy function) and inserted in the priority queue according to this cost value. The algorithm cycles through the following steps: an edge collapse is popped from the top of the queue, executed, and the queue is updated [Cort].

```

// Generate the initial priority queue of edge collapse.
for ( $\forall e \in M$ ) {
    // optimize new vertex position.
    for ( $v = v_s, (v_s+v_t)/2, v_t$ ) {
        // optimize over 3 different initial positions.
        improve position of  $v$  by minimizing cost of collapse.
    }
    choose  $v$  with lowest cost  $c$ .
    insert the triple  $(e, v, c)$  in queue.
}

// Collapse the mesh.
while (queue not empty) {
    delete from queue  $e$  with lowest  $c$ .
    perform collapse of  $e$ .
    recalculate cost of every edge in  $\ast(e)$ , the neighborhood of the
        transformation and update their location in queue.
}

```

Hence, after n collapses, PM has reduced a mesh $\hat{M} = M^n$ to a coarse version M^0 by applying n successive edge collapse transformations:

$$(\hat{M} = M^n) \xrightarrow{ecol_{n-1}} \dots \xrightarrow{ecol_1} M^1 \xrightarrow{ecol_0} M^0$$

Let m_0 be the number of vertices in M^0 , and let us label the vertices of mesh M^i as $V^i = \{v_1, \dots, v_{m_0+i}\}$, so that vertex v_{m_0+i+1} is removed by $ecol_i$. As vertices may have

different positions in different LOD representations of the same mesh, we denote the position of v_j in M^i as v_j^i . A key observation is that the edge collapse is reversible (see Figure 3.6). That inverse transformation is called vertex split. Therefore, we can represent an arbitrary mesh $\hat{M}$ as a simple mesh M^0 with a sequence of n vertex split records:

$$M^0 \xrightarrow{vsplit_0} M^1 \xrightarrow{vsplit_1} \dots \xrightarrow{vsplit_{n-1}} (M^n = \hat{M})$$

The resulting sequence of meshes $M^0, \dots, M^n = \hat{M}$ can be quickly traversed at runtime by applying a subsequence of vsplit or ecol transformations, and is therefore effective for real-time LOD control [Hop98].

3.6.3 Geomorphs

A very nice property of the vertex split (and edge collapse) transformation is that a smooth visual transition can be created between any two meshes M^i and M^{i+1} . Without geomorphs, instantaneous switching between two subsequent meshes would lead to a perceptible ‘popping’ effect. The transition has to be animated with more ‘frames’ than only M^i and M^{i+1} in order to look smoother. Hence, we construct a geomorph $M^G(\alpha)$ with blend parameter $0 \leq \alpha \leq 1$ such that $M^G(0)$ looks like M^i and $M^G(1) = M^{i+1}$. Mesh $M^G(\alpha)$ is defined as $(K^{i+1}, V^G(\alpha))$ whose connectivity is that of M^{i+1} and whose vertex positions linearly interpolate from $v_{x_i} \in M^i$ to the split vertices $v_{x_i}, v_{m_0+i+1} \in M^{i+1}$:

$$v_j^G(\alpha) = \begin{cases} (\alpha)v_j^{i+1} + (1-\alpha)v_{s_i}', & j \in \{s_i, m_0 + i + 1\} \\ v_j^{i+1} = v_j', & j \in \{s_i, m_0 + i + 1\} \end{cases}$$

Figure 3.7: Vertex split operation

Moreover, since single vsplit/ecol transformations can be interpolated smoothly, so can the whole sequence. Thus, given two meshes, a coarse one M^c and a fine one M^f , a geomorph $M^G(\alpha)$ is defined such that $M^G(0) = M^c$ and $M^G(1) = M^f$. To obtain M^G , we associate each vertex v_j of M^f with its ancestor M^c . The index $A^c(j)$ of the ancestor of v_j in M^c is found by recursively backtracking through the vsplit transformation that led to its creation:

$$A^c(j) = \begin{cases} j & , j \leq m_0 + c \\ A^c(s_{j-m_0-1}) & , j > m_0 + c \end{cases}$$

We have outlined the construction of geomorphs between PM meshes containing only position attributes. Construction of geomorphs for meshes containing discrete and scalar attributes is also possible. Discrete attributes by their nature cannot be interpolated. But the previous method of geomorphing automatically introduces them without any particular need for smooth transition. Scalar attributes defined on each three corners of faces can be smoothly interpolated [Hop96]. Finally, it has been observed in [Hop98] that the creation of a geomorph requires approximately twice as much time as simple iteration through the PM sequence.

3.6.4 Progressive transmission

On networked systems, applications use different servers; commonly a file server for example. The files are transferred back and forth between the workstations and servers rather than from the local disk drive on the workstation (much faster). Regardless of the communication line speed, users want to grasp interactively a rough idea at least, of what is going on in the computer within a minimum delay. They certainly do not want to wait until the end of the object transmission to see some results. Progressive Mesh is a natural representation for progressive transmission. The coarser mesh M^0 is transmitted first, followed by the stream of vsplit records. The receiving process incrementally rebuilds $\hat{M}$ as the records arrive, and animates the changing mesh. The original mesh $\hat{M}$ is recovered exactly after all records are received, since PM is a lossless representation.

3.6.5 Mesh compression

A good model should always minimize the amount of memory space it consumes to store objects. There are two approaches for achieving this. One is to use mesh simplification that reduces the number of faces in the model by processing its structure logically. The other is mesh compression, to minimize the space used to store the model at the binary level. Surface compression is an alternative that attempts to reduce the number of bits to encode the mesh (at the expense of increased computation time) rather than reducing the number of surface elements.

As for mesh simplification, we will simply compare the size of fixed and continuous-resolution models. Fixed models are stored as two sets of vertex and face records. Each vertex record contains three IEEE single-precision floating-point coordinates for a total of 12 bytes. Each face record contains three integer vertex indices, which occupy 12 bytes (we assume there are $2n$ faces for n vertices on average). It follows that the model occupies $(n \text{ vertices}) \times 12 \text{ bytes} + (2n \text{ faces}) \times 12 \text{ bytes} = 36n \text{ bytes}$. The storage of a continuous-resolution model is identical to that of the fixed model except for the n edge collapses. An edge collapse is encoded with two indices (edge end-points) and one new vertex. Hence the edge records contains two integers (8 bytes) and a vertex (12 bytes). Hence the continuous-resolution model occupies $36n + (n \text{ edges}) \times 8 \text{ bytes} + (n \text{ new vertices}) \times 12 \text{ bytes} = 56n \text{ bytes}$. And that is a steady upper bound of 56% increase for a family of n meshes! In practice it is even lower since meshes are never completely simplified because of topological reasons [Lit].

On the compression side, all kinds of computer numerical optimizations can be applied to the model (although our implementation did not explore these possibilities). For instance, a reduced number of bits for integer values can often be employed instead of a full 32-bits integer. Considering that a vertex has on average six neighbor vertices, instead of storing all three vertex indices of a vertex split (s_i, l_i, r_i) , it is possible to retrieve l_i and r_i with five bits only (the number of permutations $P_2^6 = 30$ can be encoded in $\lceil \log_2(30) \rceil = 5$). Also after a vertex split (if PM is encoded with vertex splits instead of edge collapses), it is not necessary to record the two new vertices $v_{s_i}^{i+1}$ and $v_{m_i+i+1}^{i+1}$. We

can predict those positions according to v_s^j using variable-length delta-encoding to reduce storage. Again, with this integer truncating method it is possible to cut down on face record sizes. A face record of three vertex indices can be reduced to $3 \lceil \log_2(n) \rceil$ bits of storage. Furthermore, it has been found specifically for the geometry of meshes that 3D vertex coordinates can be expressed with 16-bits fixed precision values (rather than 32-bits IEEE single-precision floating point values) without loss of significant visual quality [Deer].

Deltas in general compress better than absolute values since they tend to be much smaller in magnitude [Hop98]. Furthermore, all the delta-encodings in vsplit records can further be minimized using Huffman codes. Say that v_0 is to be split into $\{v_1, v_2\}$. The algorithm could even optimize the Huffman codes by delta-encoding using either $\{v_1 - v_0, v_2 - v_0\}$ or $\{\frac{1}{2}(v_1 + v_2) - v_0, \frac{1}{2}(v_1 - v_2)\}$ [Hop96]. Finally, as a last resort in model compression, online binary compression/decompression (such as the gzip method) has already successfully been applied to the model on top of all the other compression techniques [Hop98].

3.6.6 Selective refinement

When refining a coarse mesh M^0 to full resolution M^n , the relevance of new added details may become circumstantial. The user might have zoomed the display to a small region of the whole mesh for example. Thus, the parts of the mesh clipped outside the viewing device (screen) need not be well defined while the part displayed on the screen should be refined at the best possible resolution. For example in flight simulation, while the user is flying over a terrain (a mesh), the only refined region should be the user's visual field. Furthermore, regions far from the viewer need not to be as defined as closer ones. Same conclusion for faces oriented away (on the other side of the viewed object) from the viewer. These cases must be accounted for in order to render the smallest set of relevant faces on the object. We present here a basic real-time technique for selectively refining PM meshes according to dynamically changing view parameters (more information in [Hop97b]).

PM can support selective refinement, where details (vsplit transformations) are added to the model only in desired areas of the mesh. The application using PM has to provide a function $REFINE(v)$ that returns a boolean in real-time, indicating whether the neighborhood around v should be refined. An initial mesh M^c is selectively refined by iterating through the list $\{vsplit_c, \dots, vsplit_{n-1}\}$ and performing $vsplit_i$ if and only if

1- all three vertices $\{v_s, v_t, v_r\}$ are present in the mesh, and

2- $REFINE(v_{s_i})$ evaluates to TRUE

When exploiting selective refinement, the algorithm might stumble on a particular vsplit operation where one (or more) vertex v_j is missing due to a previous vsplit ($vsplit_{j-m_0-1}$) operation which was not allowed by selective refinement [Hop96]. Step one is verified first for this reason. Further contributions on this topic are available at [Mor98].

3.7 Summary & Discussion

Simplification of highly detailed meshes has emerged as an important issue in many computer graphics related fields. A whole library of different algorithms has been proposed in the literature (to have an overview of the breadth of the field, see [Heck]). The first generations of such algorithms are topology preserving. They use simple local mesh reduction operations on submeshes and rebuild coarser submeshes following some optimization heuristics.

Many such heuristics are available for this task. Some rely on vertex distance to surface plane [Sc92]. Others evaluate more complex energy functions to be minimized [Hop93]. But the general goals are the same: reduce the mesh to its simplest expression, preserve mesh volume and sharp edges (geometry and overall appearance). The guarantee

of an error bound after simplification is not an option anymore, but rather a frequent requirement.

In short, PM offers an efficient, lossless, continuous-resolution mesh representation. It has also proved to be of industrial strength and user friendly since its current implementation is the basis for the Progressive Mesh feature available in Microsoft's DirectX 5.0 product release [Hop98]. Moreover, PM found a superior successor, the PSC representation, a generalization of PM that permits topological changes to meshes [Hop97a]. Based on PM, PSC makes use of a more general refinement transformation allowing any input model to have an arbitrary topology (any dimension, non-orientable, non-manifold, non-regular). By allowing changes to topology, PSC approximations reach a higher fidelity. Eventually, one of the ultimate goals in continuous-resolution mesh representations would be to integrate them into animated object applications.

Mesh simplification is foreseen to have many applications due to its capabilities including transmission of 3D models on LANs and Internet, efficient storage format and continuous LOD on demand... It shall be found in a variety of computer graphics applications, scientific and domestic.

Chapter 4

Mesh Partitioning

Many problems can be represented by graphs where nodes stand for amounts of work, and edges schematize the information exchanges. The graph partitioning problem is invoked every time one needs to decompose a 'graph' problem into smaller subproblems in order to solve them simultaneously or even sequentially, but in both cases, to solve them faster than the original larger problem could be solved at once [Ci94b].

Identifying the parallelism in a problem by partitioning its data among a set of processors is the fundamental issue in parallel computing. It constitutes the first step in designing any parallel solution. The data set for some problems can be easily related to graphs (a mesh is a graph with a geometry embedding for example). Hence a parallel solution starts with partitioning the graph embedded in the problem. Several such graph partitioning algorithms have been developed in the past. We will take a look at them.

When a given problem can be modeled on graphs, graph partitioning divides the independent entities of the problem, and identifies the possible concurrency. Partitioning

a graph into subgraphs leads to a decomposition of the data, and/or tasks, associated with the computational problem. The resulting partition subgraphs can then be mapped to different processors. Graph partitioning also has an important role to play in the design of many serial algorithms by means of the divide and conquer paradigm. Two important examples of this algorithmic paradigm are the solution of partial differential equations (PDEs) by domain decomposition and the computation of nested dissection orderings for solving sparse linear systems of equations. Graph partitioning is also used extensively in other applications such as circuit layout, VLSI design and CAD.

Two main objectives are usually stated in the partitioning problem: to divide a given graph into a specified number of subgraphs p such that 1- the subgraphs have approximately an equal number of elements and 2- as few edges as possible join the subgraphs to each other (these edges are 'cut' by the partition; the set of those edges is the 'edge-cut'). In the context of parallel computations, the size of the subgraphs determines the computational load on processors, and the size of the edge-cut is a measure of the communication volume between processors in parallel programs. More specific requirements may be needed for peculiar parallel applications. For example, the work associated with a subgraph may be modeled more accurately by attaching a weight to vertices (nodes), and then create partitions of approximately equal weight. The communication costs in the algorithm might be modeled more accurately on how many subgraphs a given subgraph is adjacent to or how many boundary vertices it has. In addition, the geometrical shape of the subgraph (e.g. aspect ratio) may be an important

parameter to some algorithms. The connectivity of the subgraphs might also be a concern [Pot97].

4.1 Graph partitioning background

We will denote a graph G described by its vertex set V and edge set E . An edge $e \in E$ is a pair (u, v) ; where u and v are the vertex endpoints of e . We will refer to the number of vertices in a graph as $n = |V|$. A 2-way partition of a connected graph G is a division of its vertices in two sets A and B . The set of edges joining vertices in A to vertices in B is an edge separator (edge-cut). The removal of these edges would disconnect the graph into two components. In applications such as domain decomposition, vertices of A would be mapped to one set of processors, and vertices of B to another. The edge-cut size would be a measure of the volume of communication necessary between the two groups of processors. Hence, one goal in partitioning a graph for parallel processing is to minimize the number of edges cut by the partition to keep the communication cost low. As a second goal, we want to balance the computational work (load) between both groups of processors. This is achieved by prescribing the number of vertices in A and B to within a tolerance threshold.

Other applications call for a vertex separator. The vertex separator is a set of vertices S whose removal disconnects the graph in two parts. Such a partition is called a

dissection. Once again, it is important that the separator be as small as possible and that both subsets be approximately of equal size.

The graph partitioning problem is NP-hard, i.e. it is unlikely that vertex separators or edge separators can be computed efficiently (in polynomial time) for arbitrary graphs. Consequently many researchers have designed heuristic methods to approximate the problem for general and particular graphs. The existing heuristics can be organized into two classes: recursive methods and greedy methods.

4.2 Recursive methods

All these recursive methods make use of a bisection framework. We begin with a quick presentation of the recursive nature of bisection methods and then review these methods.

4.2.1 Recursive bisection

Most common partitioning algorithms are bisection oriented. They must be applied recursively when they are used to derive arbitrary p -way partitions (p subsets). All bisection algorithms have a recursive version: it consists of the bisection algorithm itself mounted on a recursive bisection (RB) framework. RB first divides a graph into two

approximately equal-sized subgraphs, using any bisection algorithm, and then recursively divides the subgraphs until it generated p subgraphs of approximately n/p vertices. Ideally, we would choose an optimal bisection algorithm. But we know that such an algorithm is NP-complete. Practical RB algorithms rather use more efficient heuristics which are, for the most part, designed to find the best possible bisection within allowed time. Some extended heuristics even use quadsection and octsection instead of bisection. Explanations on these along with mathematical analysis on bisection accuracy, speed and guaranties can be found in [Hor].

```

Recursive_Bisection_Scheme( $G, p$ )
{
    Apply function Bisection to find a bisection  $G_L$  and  $G_R$  of  $G$ 
    if  $|G_L| > n/p$  then
        Recursive_Bisection_Scheme( $G_L, p/2$ )
        Recursive_Bisection_Scheme( $G_R, p/2$ )
    return the subgraphs  $(G_1, \dots, G_p)$ 
}

```

There might be some inconveniences with the use of recursive bisection. Obviously, it will fail to produce approximately equal sized subsets when p is not a power of two. Secondly, the graph partitioning problem is not most efficiently solved with bisection: for example, an optimal 4-way partition is not the result of recursively bisecting the graph twice, even if bisections are optimal. In spite of these observations, most partitioning algorithms are bisection based.

4.2.2 Spectral methods

A fairly new heuristic appeared in [Si91]. It paved the way for an important class of partitioning algorithms called spectral methods using eigenvectors of a matrix associated with a graph to create a partition. It bisects the graph by considering an eigenvector of an associated matrix to gain an understanding of global properties of the graph. This method has received much attention because it offers a good balance of generality, quality and efficiency.

Various formulations of spectral bisection can be found in different papers [Si91, Ci94b, He93]. Here we assign a variable x_i to every vertex v_i such that $x_i = \pm 1$ depending on which of the two subsets it belongs to; and $\sum x_i = 0$ (assuming an even number of vertices). Then, notice that the function $f(x) = \frac{1}{4} \sum (x_i - x_j)^2$, for $\forall e_{ij} \in E$, counts the number of edge crossing between two subsets since $(x_i - x_j)^2$ is 0 with x 's of same sign and 4 otherwise. This function, proportional to the edge-cut size, must be minimized. Hence $f(x)$ is converted to a $n \times n$ matrix to make the solution more apparent.

First we define the adjacency matrix $A_{i,j}$ to be 1 if $(v_i, v_j) \in E$, and 0 otherwise. The degree matrix $D_{i,j}$ is defined as $d(i)$ (degree of v_i) if $i = j$ and 0 otherwise. Next, the function is transposed into matrix algebra terms:

$$\sum_{E_{ij}} (x_i - x_j)^2 = \sum_{E_{ij}} (x_i^2 + x_j^2) - \sum_{E_{ij}} 2x_i x_j$$

and the two terms are refined to :

$$\sum_{E_{ij}} 2x_i x_j = \sum_{v_i} \sum_{v_j} x_i A_{ij} x_j = \sum_{v_i} x_i \sum_{v_j} A_{ij} x_j = x^T A x$$

$$\sum_{E_{ij}} (x_i^2 + x_j^2) = \sum_{E_{ij}} 2 = 2|E| = \sum_{v_i} x_i^2 d_i = x^T D x$$

Finally, we define the Laplacian matrix L of graph G , $L(G) = D - A$. And finally, $f(x) = \frac{1}{4} x^T L x$. Coupling this with the constraints on x , we define the discrete bisection problem:

$$\text{Minimize } \frac{1}{4} x^T L x \text{ subject to } x^T \vec{1} = 0 \text{ and } x_i = \pm 1$$

where $\vec{1}$ is the n -vector $(1, 1, 1, \dots)^T$. But bisection is NP-complete. We cannot expect to solve this problem exactly. However we can approximate this intractable problem with a tractable one if we relax the discreteness constraint that $x_i = \pm 1$ and let it vary continuously between $\sqrt{n}$ and $-\sqrt{n}$ (where $n = |V|$) to define the continuous bisection problem:

$$\text{Minimize } \frac{1}{4} x^T L x \text{ subject to } x^T \vec{1} = 0 \text{ and } x^T x = n$$

in which the elements of vector x may take on any values satisfying the norm constraint. This continuous problem is only an approximation of the discrete problem, and the values defining its solution must be mapped back to ± 1 by some appropriate scheme to define a partition. Let us emphasize that relaxing of the discreteness constraint is a crucial step. Otherwise, spectral methods could not adapt efficiently to graph partitioning. Ideally, the solution to the continuous problem will have entries clustered near ± 1 , showing a good approximation to the discrete problem [He93].

This solution is theoretically the nearest discrete point to the continuous optimum or else a lower bound on the edge-cut size produced by any balanced partitioning of the graph. That is because the solution space of the continuous problem contains the solution space of the discrete problem [He93]. The dominant cost of this algorithm is the calculation of the eigenvector of L . An efficient approach is the Lanczos algorithm [Gol]. More theoretical results are available in [Fi73, Fi75, He92].

Since then, this method has stimulated the work of many at the design, specialization, refinement and analysis level. Work on eigenvalues and eigenvectors can be found in [Cv80, Cv88, Mo91, Mo92]. Analyses were conducted in [Ci97, Spi]. Other authors also developed the idea [Bop, Pot90, Pow, Re90, Re94, Mi95, Ro].

4.2.3 Geometric methods

Graphs from large-scale problems in scientific computing are often defined geometrically. They are meshes in d -dimensional Euclidian space (typically 2D or 3D). A mesh embedded in space contains geometric information about its vertices (coordinates). Algorithms for partitioning meshes by bisecting along coordinate axes have already been considered in the past. Coordinate bisection is a simple heuristic that chooses a partitioning plane perpendicular to one of the coordinate axis. Inertial bisection tries to do better by choosing a plane perpendicular to some version of a moment of inertia of the mesh points. Those early algorithms are fast. However, the quality of the separator obtained by such straight-line cuts is not good relatively to other algorithms.

This section covers the most well known geometric mesh partitioner. That method computes a separator by using a circle instead of a line to cut the mesh. The method sees the mesh as an edgeless graph, a collection of vertices. It partitions the d -dimensional mesh by finding a suitable sphere in $d+1$ -space, and dividing the vertices which are inside and outside of the sphere. The cutting circle is found by a randomized algorithm that involves a conformal mapping of the points on the surface of the sphere in $d+1$ -space. Let us review the work of the fathers of geometric partitioning [Th98, Th93]. In the following algorithm outline, the mesh vertices are scaled and translated to a $[-1..1]$ system.

1-*Project up*. Project the input points stereographically from R^d to the surface of the unit sphere centered at the origin in R^{d+1} . Point $p \in R^d$ is projected to the surface of the sphere along the line through p and the north pole $(0,0,\dots,0,1)$.

2-*Find the centerpoint*. Compute a centerpoint of the points projected on the surface of the sphere in R^{d+1} . This is a special point in the interior of the unit sphere (as described below).

3-*Conformal map: rotate&dilate*. Move the centerpoint to the origin of the sphere (and all projected points in R^{d+1} as well) in 2 steps. First rotate the projected points about the origin in R^{d+1} so that the centerpoint becomes $(0,0,\dots,0,r)$ on the $d+1^{\text{th}}$ axis. Second, dilate the points on the sphere so that the center point becomes the origin. The dilatation can be described as a scaling in R^d : project the rotated points stereographically down to R^d ; scale the points in R^d by a factor $((1-r)/(1+r))^{1/d}$; and project the scaled points up to the unit sphere in R^{d+1} again.

4-*Find a great circle*. Choose a random great circle (e.g. d -dimensional unit sphere) on the unit sphere in R^{d+1} .

5-*Unmap and project down*. Transform the great circle in R^{d+1} to a circle in R^d by undoing the dilation, rotation, and projecting back from R^{d+1} to R^d .

6-*Convert circle to separator*. For the edge separator version of this method, the 2 sets A and B are the vertices that lie inside and outside the circle respectively (This last point could be rephrased to represent a vertex separator).

The centerpoint of a given set of points is a point such that every hyperplane through it divides the set of points approximately evenly into two subsets, which means in this case that the worst-case ratio of the sizes of the two subsets is $d:1$. It was proved that every finite point set in $\mathbb{R}^d$ has a centerpoint, and this proof yields a polynomial time algorithm that uses linear programming to compute the centerpoint. But this solution is much too slow to be useful and heuristics are used instead. After projection and conformal mapping, the centerpoint of the mesh points has moved to the origin of the sphere. Therefore the mapped points are divided approximately evenly by every plane through the origin, that is by every great circle on the unit sphere in $\mathbb{R}^{d+1}$.

4.2.3.1 In practice

The algorithm can run on any mesh with no requirements on its geometry and topology. It has proved to generate good partitions even for badly shaped meshes. The theoretical foundation (theorem) of this algorithm makes use of mesh classification (overlap graphs) and a mesh model (neighborhood system) that define meshes [Gi94]. This framework was necessary to evaluate theoretical guarantees on the algorithm results. But any practical implementation of geometric methods takes a simpler approach that does not require this neighborhood system. It simply divides the vertices into those inside and outside the separating circle (edge separator). This is simpler than a vertex separator since we do not have to identify a third subset of vertices (or edges) as the partition separator.

Also, the separating circle does not necessarily split the mesh exactly in half. In theory, the centerpoint construction guarantees a splitting ratio no worse than $d+1:1$. And common implementations actually use an approximate centerpoint construction with a weaker guarantee [Th93, Ep93, Ep96]. But in practice, they lead to much better splits than theory predicts (most splits are less than 20% uneven). This ratio does not sound like much of an improvement. But it does not pose a problem: one has only to shift the separating circle on its normal direction and stop it where it will evenly split the mesh [Gi94].

4.2.3.2 Discussion

The geometric algorithm has some advantages. It examines only the vertices of the mesh, and makes no use of the edges except to compute the quality of the generated separator. And although the theory behind the geometric partitioner is fairly complicated the algorithm is simple. Its computations are local, simple and linear in the number of vertices. The drawback, however, is that it cannot be directly applied to graphs with no coordinate information. Most recently, researchers have tried to amalgamate spectral and geometric methods together [Gi95], which apparently yields better results than each individually.

4.3 Other partition-related algorithms

4.3.1 Multilevel method

In search of faster solutions to complex computational problems P , modern algorithms sometimes combine a basic algorithm p that solves P with another algorithm x which has nothing to do with problem P . An example of that is the use of merge operations within a sorting algorithm when the sorting algorithm functions in the divide-and-conquer fashion. Subparts of the set are sorted individually, merged, resorted, ... etc.

Hence, it is not surprising that graph partitioning does not differ from this duality. For speed's sake, some researchers looked for a way to reduce the size of the graph, partition it and somehow in the last step, map the partition of the reduced graph to the original graph [Si93, He95]. What is surprising however is the irony of the situation: graph simplification is used to speed up partitioning; whereas we want to use partitioning to speed up mesh simplification.

This method looks at the mesh with a large number of vertices as the finest graph in a sequence of coarser graphs to be computed. A series of shrinking operations are performed until a coarsening threshold is met. Then basically, any partitioning algorithm can be run on the coarsest graph. That partition is associated to the fine graph (mesh) by reversing the series of shrinking operations.

```

Multilevel_Partition(graph Go)
    G = Go
    until (G is small enough) do
        G = coarsen(G)
    Partition = Partition_Graph (G)
    until (G == Go) do
        G = uncoarsen(G)
        Partition = uncoarsen(Partition)
    return (Partition)

```

4.3.1.1 Coarsening step

The authors of [Si93] coarsen a graph by finding a subset of non-adjacent vertices S , and then ‘growing’ neighborhoods around each vertex in S using the graph connectivity until all vertices have been included in at least one neighborhood. The coarse graph is the edge-less vertex set S with a new connectivity: a virtual new edge joins two vertices in S if their neighborhoods intersect, e.g. if they have common vertices in their neighborhoods. The set of vertices S can be chosen to be a maximal independent set of the original graph [Fab].

A more popular method is to induce a series of edge collapse operations to the graph (see Chapter 3). In order to have an even contraction of the graph, we need somehow to capture the local coarsening information (at each edge and vertex). The partitioner also needs this information to derive better partitions of the fine graph through

partitioning the coarse graph. We choose a simple weighting system. For example, say that the edges and vertices of the fine graph all have weight one. When the two endpoints of a contracted edge have a common neighbor, the new edge joining the neighbor to the new vertex has a weight equal to the sum of the weight of the two replaced edges. The weights of all other edges remain unchanged. The weight of the new vertex is the sum of the weight of the endpoints.

4.3.1.2 Uncoarsening step

Vertex and edge weights were recorded during the coarsening step. A vertex in a coarse graph corresponds to a unique set of merged vertices in the fine graph and hence, it is possible to compute a partition of the latter from a partition of the former. The coarsening procedure has three important properties:

- 1- The total weight of the edges cut by a partition in the coarse graph is equal to the number of the edges cut in the fine graph when that partition is projected from the former to the latter.
- 2- The sum of the vertex weights is the same in the fine and coarse subgraphs. Hence, constraints on the subset sizes are preserved in the coarse graph in the form of weight sums.
- 3- Any partition of the coarse graph corresponds unambiguously to a partition of the fine graph.

The uncoarsening of a partition is trivial. Each vertex in a coarse graph is simply the union of one or more vertices from the original graph. We simply assign vertices from the original graph to the same subset its coarse graph counterpart belongs to. Since the weight of a coarse graph vertex is the sum of the weights of its constituents, the uncoarsening procedure preserves the sum of the vertex weights in each subset, and the sum of the edges as well.

The partitioning algorithm used maintains a load balance between the generated subsets by ensuring that the sum of the node weights in each subset of the coarse graph, once partitioned, is approximately the same. The edge-cut is minimized by keeping the sum of the weights of the cut edges low. The invariance of the total vertex weights in each subset, and the sum of the weights of the cut edges, under the coarsening/uncoarsening steps, ensure that a good partition of the coarse graph is also a reasonable initial partition of the fine graph [Pot97].

4.3.1.3 Discussion

The details of the partitioning of the coarsest graph are not central to the multilevel technique. So we will not dwell upon them. However, it is important to note that the partitioning algorithm must be able to handle edge and vertex weights, even if the original graph is not weighted.

The multilevel technique operations (computing the maximal independent set of G , graph coarsening and uncoarsening) all run in $O(|E|)$. Thus, the multilevel technique can greatly speed up any algorithm whose complexity is higher than linear (most of them).

The uncoarsened graph partition has more degrees of freedom than the coarse graph partition. Consequently, the best partition, optimal for the coarse graph, might not be optimal when mapped on the fine graph. Then one possibility (standard procedure in most implementations) would be to apply a local refinement scheme to the final partition (see Section 4.3.2). The multilevel technique has shown particularly good results when coupled with a spectral partitioning method in terms of execution time and partition quality.

4.3.2 Optimization methods

Whatever partitioning method may be used, one can use a post-processing optimizer to improve the load balancing or the edge-cut of partitions. Without a doubt, the most famous local optimization method for improving a given partition is due to Kernighan and Lin [KL]. Fundamentally, the method is based on a given bisection (V_1, V_2) of G , and tries to improve it by exchanging subsets of vertices from both subgraphs. The subset selection criterion is determined from the following gain function:

$$\begin{aligned} \text{for } v \in V_1, \quad g_v &= d_{V_2}(v) - d_{V_1}(v) \\ \text{for } w \in V_2, \quad g_w &= d_{V_1}(w) - d_{V_2}(w) \\ g_{vw} &= g_v + g_w - 2\delta(v, w); \quad \delta(v, w) = \begin{cases} 1 & \text{if } (v, w) \in E \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

where $d_A(v)$ is the number of neighbors of v that belongs to subset A . So after computing g_v and g_w for $(\forall v \in V_1 \text{ and } \forall w \in V_2)$, the algorithm chooses a pair of vertices (v_1, w_1) which maximizes the gain $g_{v_1 w_1}$. It exchanges the pair of vertices. Then the gain values of all neighbors of v_1 and w_1 are updated. The optimizing process is iterated n times (until the gain of the last pair $g_{v_n w_n}$ is computed). Finally, the algorithm chooses all pairs $\{(v_1, w_1), \dots, (v_k, w_k)\}$ for $k < n$ with positive gain and exchange them. The process is repeated until there is no better improvement to the mesh [Ci94b].

KL(V_1, V_2)

Compute g_v, g_w for each $v \in V_1$ and each $w \in V_2$

do

$Q_{V_1} = \emptyset, Q_{V_2} = \emptyset$

for $i = 1..n$ **do**

Choose $v_i \in V_1 - Q_{V_1}$, and $w_i \in V_2 - Q_{V_2}$, such that $g_{v_i w_i}$ is maximal.

Set $Q_{V_1} = Q_{V_1} \cup \{v_i\}$ and $Q_{V_2} = Q_{V_2} \cup \{w_i\}$

// update of neighbor vertices

For $\forall v \in V_1 - Q_{V_1}$, do $g_v = g_v + 2\delta(v, v_i) - 2\delta(v, w_i)$

For $\forall w \in V_2 - Q_{V_2}$, do $g_w = g_w + 2\delta(w, w_i) - 2\delta(w, v_i)$

Choose $k \in \{1, \dots, n-1\}$ to maximize $\sum_{i=1}^k g_{v_i w_i}$

Interchange subsets $\{v_1, \dots, v_k\}$ and $\{w_1, \dots, w_k\}$

until (no more subset interchanges)

Inside the outer loop, the pairs are coupled one by one from the highest gain to the lowest (which may be negative). For each new pair, the chosen vertices are inserted in the exchanged set, and the remaining candidates have their gain recomputed. Once all n pairs are built, the list of pairs is traversed from the beginning till the first null gain. The vertices from all these traversed pairs are then exchanged to the other set. The complexity of the outer loop is $O(n^2 \log n)$. As for how many times the outer loop is executed, this is all depends on how good the initial partition is. The same conclusion applies to KL's efficiency. In any case though, the outer loop is necessarily bounded by $O(n)$.

Obviously, there are many improvements that could be brought to this persistent algorithm (KL influenced many other local optimization methods). For example in [Fid], vertices are exchanged individually (not in pairs) alternating from both subsets. This version runs in $\mathcal{O}(n^2)$ due to a special sorting technique, a nice improvement. In [He93] the previous method is further generalized to an arbitrary partition, the k -way partitioning problem. Hence this one runs in $\mathcal{O}(kn)$. Finally, a more accurate method, C-L-O, was brought to combine local search methods with simulated annealing [Otto].

4.4 Greedy methods

We will now discuss the partitioning method used in this thesis for the purpose of parallel mesh simplification.

Imagined partitions evolving on the graph in a way similar to bacteria in a Petri dish; in an uncontrolled greedy manner. Greedy algorithms are a natural and naive way to look at some problems. In some cases, they give reasonable results, especially here for the graph partitioning problem.

4.4.1 An intuitive start

My first attempt was an intuitive one inspired by [Lee]. Below is the idea in pseudocode:

algorithm Intuitive_Partition(graph $G(V, T)$, p)

```

N = |T| // the number of triangles in graph
remaining_triangles = N;
// number of triangles that do not belong to any partition

taken[1..N] = FALSE;
// If taken[TriId] is TRUE, the triangle TriId belongs to a
// partition. If FALSE, the triangle has not yet been assigned
// to a partition.

for i = 1..p do
    Capturedi =  $\emptyset$  // Triangles in partitioni
    Candidatesi =  $\emptyset$  // Possible triangles to add to partitioni

// Initialize Nbr_part partitions with one triangle each.
for PartitionId = 1..p do
    repeat
        TriId = random number in [1..N]
    until (taken[TriId] == FALSE)
    add_triangle_to_partition(PartitionId, TriId)

while (remaining_triangles > 0) do
    for PartitionId = 1..p

```

```

    repeat
        TriId = pop triId off queue Candidates
    until (taken[TriId] == FALSE)
    add_triangle_to_partition(PartitionId, TriId)

return(Captured)

procedure add_triangle_to_partition(PartitionId, TriId)
    if (taken[TriId] == FALSE)
        CapturedPartitionId = CapturedPartitionId  $\cup$  {TriId}
        taken[TriId] = TRUE;
        //append the 3 neighbor triangles of TriId
        insert {neighbors(TriId)} in CandidatePartitionId
        remaining_triangles = remaining_triangles - 1

```

The algorithm will correctly and completely partition the triangular mesh into p subsets provided that the underlying graph is one connected component only. The rest is self-explanatory: it first selects a random triangle seed for each subset of the partition. Then, in round robin, it grows the subsets one triangle at a time selecting the oldest candidate triangle from the appropriate candidate triangle queue. Then the three triangles adjacent to the new 'captured' triangle are inserted in the subset's candidate triangle queue. The algorithm stops when all triangles have been partitioned. The figure below shows the Duck (4K faces) partitioned into eight subsets.

Figure 4.1: An exploded view of a 8-way partition of the NRC Duck

This simple greedy heuristic will yield acceptable partitions in much less time than the previous more complicated partitioning methods. There is a catch though: the initialization of partitions with random seeds. The element of randomness makes this algorithm not only greedy but also classifies it in the probabilistic algorithm category.

Optimal seeds would allow the rest of the algorithm to yield very even partition sizes. However, finding such a set of seeds is prohibitively expensive. When an algorithm is faced with a choice, it is sometimes preferable to choose a course of action at random,

rather than spending time working out which alternative is best. The main characteristic of probabilistic algorithms is that the same algorithm may behave differently when applied twice to the same problem instance. Its execution time, and even the result obtained, may vary considerably from one use to the next. The algorithm can thus generate different correct solutions for one instance unlike deterministic algorithms. Probabilistic algorithms are mostly used in numerical problems where the algorithms are initialized with a rough initial approximation of the solution and then iteratively converge towards a more exact solution [Bra].

In the algorithm above, the seeds are simply chosen at random without any concern other than verifying their availability status. Of course this should be considered as a basic seeding guideline and does not have to be implemented as such. Clearly, the algorithm grows the subsets around those seeds (say in concentric manner). And whenever two subsets reach each other, they mutually block each other in this area. If a subset is blocked all around by other subsets before it reaches full size, then it is locked out and will remain as such. Hence the algorithm would generate more or less balanced partitions. However, there is a way to improve the quality of those partitions by improving the seeds. From those premature blockings, we understand that the best seeds are those with maximum closest pairwise distance. With such a set of seeds, the algorithm can grow its subsets the most before they reach each other (when the last remaining triangle faces are to be partitioned) and interlock each other.

In fact this growth operation is p concurrent controlled Breadth-First-Searches (BFS) performed on the mesh and originating from the p seeds. This graph search technique simulates how the mesh is partitioned. But it can also be applied to find the seeds. Say for example that the above seeding technique was improved by performing a BFS on each new chosen seed. A certain small region R_1 could be grown (using BFS) around this first randomly chosen seed. Then a region of same size could be grown around the second randomly chosen seed. The decision criterion as to whether to keep the last created seed or not is based on whether or not its region collided a region from a previous seed. In case of collision, the last created region is deleted and another seed is randomly chosen. Otherwise the seed is accepted. Obviously, this scheme will function only if the sum of triangles in the p regions is smaller than the number of triangles in the mesh. Also, this sum has to be adapted to the topology and size of the mesh. Say that the seeding algorithm will start with this sum equivalent to an arbitrary 60% of the mesh triangles. It will try an arbitrary number of times to seed with this sum. If it fails it will keep trying with lower and lower sums until it succeeds.

This last approach tries to find sufficiently remote seeds pairwise and in acceptable time. It has been tested as one of our implementations. It generates good seeds, but these are however still not optimal. There exist greedy methods (not probabilistic) to find better seeds. Those methods would also be based on BFS.

4.4.2 Ciarlet's algorithm

It turns out that my attempt with the previous algorithm was not far off track. In the course of my research, I found a very similar algorithm by Ciarlet [Ci94a].

Transposed to a greedy approach, the graph partitioning problem can be solved by an algorithm that computes the subsets V_i one after the other by accumulating vertices (or triangles) when traveling through the graph. But how to start and stop? The way to accumulate vertices in each subset is obvious from the graph structure of the problem (meshes). A starting vertex v_s is chosen and marked. The accretion process is done by selecting the neighbors of v_s , then the neighbors of the neighbors and so on until the subset has reached the required number of vertices. This can be abstracted as building fronts of vertices around v_s one layer of vertices at the time.

The method used to choose the starting vertex v_s will affect the shape of the final partition. The manner in which one chooses the prescribed number of vertices (among all candidates of the last front) affects the quality of the final partition. Hence a greedy heuristic for solving the graph partitioning problem can be defined roughly by iterating the next three steps for every subset:

- 1- Choose v_s .
- 2- Accumulate enough descendants of v_s .
- 3- Stop according to some tie-break strategy in case of multiple choice.

There are unfortunately no theoretical results on the ‘goodness’ of the starting node. Neither are the results on how good a tie-break strategy is. Hence we will have to rely on educated intuitive guesses. Nevertheless, an obvious justification for using this greedy heuristic is its unsurpassed speed.

The algorithm derived from this heuristic is a general purpose graph partitioner for graphs that come from physical meshes (2D or 3D). As greedy algorithms do, it grows the solution step by step choosing always the best immediate decision. It builds iteratively the different subsets of the partition accumulating vertices in each subset and marks them (as partitioned) once they have been visited. In the following algorithm we define the boundary vertices as the set of unmarked vertices adjacent to marked ones (this algorithm also assumes that the input graph is one connected component only). We define d as the degree of a vertex and the update degree of a vertex as its number of neighbors which are unmarked. Recall that $n=|V|$, p is the number of subsets and the expected number of vertices per subset is $N = n/p$.

Algorithm greedy($G(V, E)$, p)

currentBoundary = random vertex in $[1..n]$

for $i = 1..p-1$ **do**

$$N_i = \left(n - \sum_{j=1}^{i-1} N_j \right) / (p - (i-1))$$

(a) choose an unmarked vertex v_i such that

$1-v_i$ belongs to currentBoundary


```

    2-if currentBoundary is not new, then  $v_i$  is an unmarked
       vertex adjacent to subset  $V_{i-1}$ 
    3- $v_i$  has minimal updated degree
 $V_i = \{v_i\}$ 
    mark  $v_i$ 

    (b)select the  $k$  unmarked vertices, neighbors of  $V_i$ 
while ( $|V_i| + k < N$ )
    -mark the  $k$  vertices
    -add them to  $V_i$ 
    -select all ( $k$ ) unmarked vertices, neighbors of  $V_i$ 
    -update their updated degree

    (c)mark ( $N - |V_i|$ ) of the  $k$  vertices with minimal updated degrees
    update currentBoundary

    (d)Mark all the remaining nodes and add them to  $V_p$ 

```

By choosing a node on the current boundary that verifies condition (2), one can be convinced that this will provide the overall regularity of the partition. As a matter of fact, because of the definition of the current boundary, the subsets will be built around the boundary in a concentric way. The tiebreak strategy in step (c) which dictates the selection of minimal updated degree vertices, also ensures the minimization of the edge-cut size.

The complexity of the algorithm is $O(|E|)$. In step (a), looking at neighbor vertices of the previous subset satisfying all three conditions is done in $O(dN)$. Otherwise, if there are no unmarked neighbors of subset V_{i-1} , then under conditions (1) and (2) only, the

operation takes $O(dN_{\text{boundary}})$. Step (b) takes $O(dN)$. Step (c) requires sorting less than N vertices on their degree values in $O(N)$ using a heap. Updating the boundary is done in $O(N)$. Hence the algorithm complexity is $p \times O(dN) = O(dn) = O(|E|)$.

4.4.2.1 Discussion on connectivity

The algorithm looks very nice and simple. However, it does not work as such even if we assume that the input mesh is one connected component only. What could be the problem? In block (b), there might be no more unmarked vertices. Variable k would be stuck to zero, driving the loop into infinity. Some precautionary steps (necessary due to the random and greedy nature of the method) were left out. Indeed, it could happen during the construction of one subset, that there are no more unmarked vertices on its boundary while the subset in progress has not reached its full size. Actually, it will happen many times (proportionally to $|V|$ and p) before the algorithm is started with the right seed that will allow it to grow the first $p-1$ subsets without being locked. One way to complete the locked subset is to choose a new starting vertex and grow an additional component to the current subset until n vertices are attained. This would of course lead to multiconnected subsets.

To avoid multiconnectivity, we could reassign the vertices of the incomplete subset to the neighboring subsets, and rebuild it. But that would create unacceptable imbalance between subsets sizes. Furthermore, the last subset is trivially built by assigning to it all the remaining (unpartitioned) vertices. Of course the connectivity of this last subset is far

from being guaranteed. A last multiconnected subset may be acceptable depending on the application that will use the partition. Otherwise one way to avoid it would be to keep the biggest unpartitioned component as the last subset and distribute the other unpartitioned components to other adjacent subsets, which will unfortunately again affect the partition balance.

In the light of those observations, I decided to implement two different versions of this algorithm. The first would refuse any compromise on multiconnectivity. It starts with a random seed and grows the subsets one after the other. If any subset locks before termination, then the partitioning is restarted from the beginning. And it will be restarted over and over until the algorithm has not built the first $p-1$ subsets. Then it would detect the remaining components of unmarked vertices, assign the biggest to subset p , and assign all others to the other first $p-1$ subsets (breaking the balance between them). It is obvious that if all first $p-1$ subsets are grown to n/p vertices and that the last subset sees exactly n/p unmarked vertices but not all connected, then the last subset will pick the biggest remaining component (smaller than n/p) and the other unmarked vertices will be assigned to adjacent subsets. So while the first $p-1$ subsets maintain the balance, the additional vertices will deteriorate this balance. As the reader will see in the next section, this first algorithm is slow in spite of its underlying fast technique (BFS) and creates unbalanced partitions of exactly p subsets.

The second version builds partitions under no connectivity constraints. The only difference is that it does not restart the partition when one subset happens to lock in

progress. It just chooses another seed and keeps growing the current subset on another component. This algorithm works at the speed of light and creates well balanced partitions. The drawback is that when asked for a p -way partition, it generates multiconnected subsets (especially the last one which could be considered as a garbage collector, picking up the vertices that the other subsets left out).

4.4.3 Analysis

When testing an algorithm, one needs to run it on good case, average case and worst case data objects. For this algorithm, we can anticipate that the goodness of an object is related to how much its topology could trigger subset locks. In that sense, a good object would be simply a sphere (with an even distribution of the vertices) on which a lock could hardly occur. A worst case would be an object with many spikes on it like a star with a small center and many branches. The average case would be meshes created from common physical objects, which is what our software has been designed for.

Unfortunately this complete testing is impossible because meshes with those special topology constraints could not be created on demand. However, we have managed to gather a good set of objects from different sources, some of them at different resolutions. We used the Grapple, the Elephant, the Nefertiti and the Duck from the National Research Council of Canada [NRCb], the Bunny and the Dragon from Stanford University [Stan].

Figure 4.2: Bunny (Surfaces)

Figure 4.3: Bunny (Full Wireframe)

Figure 4.4: Duck (Surfaces)

Figure 4.5: Duck (Full Wireframe)

Figure 4.6: Dragon (Surfaces)

Figure 4.7: Dragon (Full Wireframe)

Figure 4.8: Elephant (Surfaces)

Figure 4.9: Elephant (Full Wireframe)

Figure 4.10: Grapple (Surfaces)

Figure 4.11: Grapple (Full Wireframe)

Figure 4.12: Nefertiti (Surfaces)

Figure 4.13: Nefertiti (Full Wireframe)

It is important to remember that those algorithms are randomized. Hence every individual test had to be sampled sufficiently (250 times) to yield trustworthy average and standard deviation results on every metric that we tried to measure (recall that for a set $\{x_1, \dots, x_n\}$ the average χ is $\sum(x_i)/n$ and standard deviation is $(\sum(x_i - \chi)^2/n)^{1/2}$). We tested every object, on p-way partition tests (where $p = 2, 4, 8$, and 16) and with each algorithm (3). Our first metric is the average execution time T_a and its standard deviation T_s . Our second metric is the standard deviation S of the subset sizes. The averages S_a and standard deviation S_s were computed on it. Our third one is a standard metric on partitioning algorithm: the edge-cut size. We derived a similar metric, the triangle-cut size: the average number of triangles between subsets C_a and standard deviation C_s . Finally, for the last algorithm, which generates multiconnected partitions, metrics CT_a , CT_s , CP_a , CP_s indicate the average and standard deviation for the total component count of all the p subsets, and the component count for the first $p-1$ subsets. Finally, a last metric I found meaningful is B , the size of the biggest subset. It will be denoted by its average B_a and standard deviation B_s . This new metric complements metric S since it indicates the maximum workload applied to a processor during the parallel execution of the parallel program (the upper time limit) rather than the general imbalance between subsets. All tests were conducted on a Pentium-90/Windows95 platform with 40Mb of RAM.

4.4.3.1 First (intuitive) algorithm tests

	V	T	p	T _a (s)	T _s	S _a	S _s	C _a (t)	C _s	B _a (v)	B _s
1	453	948	2	0.019	0.026	20.6	9.41	78.02	8.42	232.7	3.57
2			4	0.039	0.025	23.4	6.46	136.43	7.17	124.9	5.50
3			8	0.067	0.024	17.9	5.42	212.30	9.64	68.0	3.55
4			16	0.126	0.035	11.4	1.90	297.27	10.22	38.5	2.57
5	1.9K	3.85K	2	0.073	0.027	92.3	59.83	150.16	20.29	972.5	28.16
6			4	0.125	0.051	80.7	44.92	299.37	21.43	515.4	33.26
7			8	0.175	0.044	54.6	20.71	472.05	31.31	278.8	22.77
8			16	0.172	0.039	29.2	7.46	649.11	16.96	141.7	10.61
9	8.1K	16.3K	2	0.343	0.081	173.5	246.16	305.66	43.80	4144	122.4
10			4	0.696	0.263	243.1	169.89	626.72	43.65	2208	135.5
11			8	0.984	0.206	168.3	87.26	1024.2	61.77	1127	91.04
12			16	1.16	0.233	94.9	27.61	1438.2	49.36	572.6	34.44
13	34.8K	69.4K	2	1.43	0.242	511.1	799.73	617.89	119.28	17647	402.4
14			4	2.38	0.858	1013.9	665.87	1328.0	114.42	9413	552.3
15			8	3.56	1.09	614.0	349.21	2191.4	162.60	4753	351.8
16			16	4.43	1.19	370.1	129.04	3100.3	147.26	2445	162.8

Table 4.1: Partitioning test results on the Bunny models

	V	T	p	T _a (s)	T _s	S _a	S _s	C _a (t)	C _s	B _a (v)	B _s
1	2K	4K	2	0.070	0.025	159.1	75.40	149.9	36.38	1088	37.70
2			4	0.091	0.032	63.7	45.12	305.0	18.47	548	38.29
3			8	0.183	0.054	54.3	19.38	470.9	20.82	296	22.88
4			16	0.246	0.056	30.2	8.46	711.5	14.80	154	11.92
5	12.3K	24.5K	2	0.449	0.089	525.6	669.4	378.1	96.34	6403	334.7
6			4	0.516	0.135	156.2	196.9	807.2	83.61	3147	81.0
7			8	0.840	0.290	191.2	122.0	1247	75.87	1669	147.4
8			16	1.13	0.332	109.7	44.3	1883	60.93	850	51.3
9	50K	100K	2	2.40	0.761	163.4	756.6	984	100.0	25105	378.3
10			4	4.25	1.55	2390	1331.2	1855	253.5	13423	533.2
11			8	8.69	1.59	2269	691.4	3091	258.1	7408	411.7
12			16	11.4	1.22	1756	269.5	4639	269.4	4334	258.2

Table 4.2: Partitioning test results on the Duck models

	 V 	 T 	p	T_a (s)	T_s	S_a	S_s	C_a (t)	C_s	B_a (v)	B_s
1	5.2K	11.1K	2	0.195	0.061	456.4	384.3	154.5	62.4	2818	183.2
2			4	0.302	0.099	228.5	100.2	337.7	46.1	1423	64.7
3			8	0.622	0.164	136.4	70.4	816.3	57.0	745	57.1
4			16	0.798	0.124	90.4	24.0	1234.3	50.3	402	29.39
5	23K	48K	2	0.864	0.085	1546.8	2035	352.4	132.0	12233	980.4
6			4	1.32	0.450	968.8	559.9	754.8	107.7	6288	368.0
7			8	2.75	0.806	561.4	284.6	1803	170.6	3233	217.2
8			16	3.64	0.744	303.6	111.2	2707	130.3	1688	122.1
9	100K	202K	2	3.69	0.438	6978	8457	712.5	275.7	53532	4186
10			4	6.60	2.47	3374	2197	1557	240.3	27243	1511
11			8	24.9	3.37	3025	1293	4298	351.0	14816	1123
12			16	32.5	2.60	1979	569.2	7215	360.9	8005	627.8

Table 4.3: Partitioning test results on the Dragon models

	 V 	 T 	p	T_a (s)	T_s	S_a	S_s	C_a (t)	C_s	B_a (v)	B_s
1	15.7K	31.5K	2	0.58	0.10	114.5	146.3	479	72.1	7920	73.1
2			4	1.07	0.38	212.5	174.5	962	106.0	4040	82.9
3			8	1.13	0.40	233.6	128.5	1461	102.1	2115	91.0
4			16	1.61	0.42	156.4	58.85	2148	89.3	1099	53.5
5	21.4K	42.8K	2	0.73	0.078	174.8	269.9	430	135.1	10785	135.0
6			4	1.32	0.49	571.3	521.0	1017	213.5	5663	314.4
7			8	1.85	0.67	316.1	191.3	1666	146.8	2871	122.4
8			16	2.66	0.63	227.6	88.9	2612	107.5	1502	83.9

Table 4.4: Partitioning test results on the Elephant [1-4], Grapple [5-6] models

These tables allow us to analyze the algorithm under four parameters. The first of them is the execution time (ET). The reader will recall that this first algorithm works in two phases: 1-find good seeds, 2-grow the subsets till partition is finished. And while the second phase seems to be the core part of the algorithm, ironically, the first part consumes most of ET. Looking at the T_a columns, we clearly see that ET is function of p (the number of subsets in the partition). While phase two spends a constant time for an object regardless of p , phase one accounts for the differences of ET. Hence, as it was stated

earlier, this situation is unacceptable. And although this algorithm is very good in its nature, it should be provided with a more efficient phase one (deterministic rather than random). Other than p , we also notice that ET is proportional to $|T|$ (the number of triangles, equivalent to $|E|$). That fact was taken for granted since the algorithm is a special case of BFS which behaves linearly in the size of the graph it traverses. In the second column, there is a little positive note however. The standard deviation remains more or less the same regardless of varying ET (for different values of p).

The second metric S_a is the average standard deviation of the sizes of the subsets of each partition test. With the exception of abnormal values in table 4.2 (line 9), S_a seems to decrease with p . For some objects it increases first. But it eventually decreases with p . This is normal since when p increases, the average number of vertices per subsets n/p decreases and the variations of size from subset to subset is believed to converge to a small value. This is true although the work of phase two gets complicated when p increases, since the seeding phase is not optimal (in this implementation).

The third metric is the edge-cut size of the partition (for commodity we rather accounted C_a , the average number inter-subset triangles). The complexity of C_a seems to behave as $O(|T|)$ and $O(p^{1/2})$. This comes with no surprise. Consider for example a mesh which is a sphere of radius r . Its surface is proportional to r^2 and contour to r . Thus, since bisection a sphere is equivalent to draw a line on its contour (say across a triangle strip around the sphere), it would intersect proportionally r edges, or $(|E|)^{1/2}$. The edge-cut grows linearly with the number of bisections γ . And since $\gamma^2 = p$ the edge-cut grows

linearly with $p^{1/2}$. On the other hand, collapsing half of the triangles will leave the sphere with half of its triangles, on and off the triangle strip. Hence C is proportional to $O(|T|)$. But those results are of course not specific to this algorithm...

I created the fourth metric B for the sake of parallelism. Since the goal of partitioning is to distribute the work load equally on each processor, and this work load is proportional to the amount of data to process; I figured that the time of computations of the parallel software will be directly in relation to the biggest subset of data to process. Indeed, not only is it important to have approximately an even partition but especially to make sure to minimize the biggest subset. In the case of a random partitioner, subsets are rarely even. Two different partitions might have the same standard deviation of subset sizes. But the best one will have the smallest B , which will allow the parallel software to finish faster. This metric is not very useful here. We will use it to compare the different partitioning algorithms instead. Let us just observe that the difference between B and n/p increases with p . This is normal since the algorithm produces more even partitions with smaller values of p .

4.4.3.2 Second algorithm (greedy) tests

	 V 	 T 	p	T_a (s)	T_s	S_a	S_s	C_a (t)	C_s	B_a (v)	B_s
1	453	948	2	0.024	0.027	14.6	11.6	88.4	13.7	232.0	7.04
2			4	0.020	0.026	35.6	19.2	155.8	4.63	130.8	9.90
3			8	0.057	0.055	27.0	4.3	228.5	6.58	68.9	4.92
4			16	1.78	2.15	13.3	0.72	325.7	0.95	33.2	0.95
5	1.9K	3.85K	2	0.078	0.028	82.6	95.5	170.0	36.0	971.6	45.58
6			4	0.099	0.064	137.9	113	316.9	26.2	541.6	63.77
7			8	0.190	0.18	99.3	30.4	495.8	27.3	305.6	25.66
8			16	2.62	3.06	30.9	8.6	724.0	26.8	138.8	11.38
9	8.1K	16.3K	2	0.449	0.23	260.3	464	353	64.8	4189	229.2
10			4	0.695	0.61	542.1	418	695	56.6	2353	242.4
11			8	2.99	3.43	410.3	141	1067	54.1	1339	129.0
12			16	49.3	51.6	108.7	40.8	1530	41.7	591	51.9
13	34.8K	69.4K	2	1.51	0.18	919	1683	778	203	17842	850.5
14			4	1.64	1.01	1678	1557	1459	125	9654	844.4
15			8	3.33	6.39	1510	748	2290	139	5478	639.2
16			16	19.4	22.5	524	158	3277	105	2658	254.7

Table 4.5: Partitioning test results on the Bunny models

	 V 	 T 	p	T_a (s)	T_s	S_a	S_s	C_a (t)	C_s	B_a (v)	B_s
1	2K	4K	2	0.077	0.027	138.8	220.4	164.5	40.0	1077	110.2
2			4	0.071	0.026	140.9	141.2	329.4	36.0	585	89.0
3			8	0.082	0.038	44.9	39.84	509.6	23.1	281	32.7
4			16	0.11	0.065	26.1	11.83	758.3	24.7	141	9.8
5	12.3K	24.5K	2	0.47	0.024	621.7	973.6	430.9	95.4	6451	486
6			4	0.45	0.227	717.2	755.9	847.2	91.3	3513	486
7			8	0.74	1.024	249.2	215.1	1304	65.2	1690	175
8			16	0.90	0.794	113.1	78.8	1975	58.2	852	71.0
9	50K	100K	2	1.95	0.026	2359	4641	1141	218.0	26203	2320
10			4	2.29	2.590	2650	3050	2001	183.3	13948	1839
11			8	9.83	16.47	1037	874.6	3156	179.9	6892	690.5
12			16	9.56	11.44	462	299.1	4772	175.6	3409	226.1

Table 4.6 Partitioning test results on the Duck models

	V	T	p	T _a (s)	T _s	S _a	S _s	C _a (t)	C _s	B _a (v)	B _s
1	5.2K	11.1K	2	0.39	0.25	1251	895.2	161.0	67.4	3189	419.6
2			4	2.89	3.7	377.3	285.7	441.2	54.6	1506	166.1
3			8	48.6	51.0	307.2	72.51	898.9	66.7	770	39.0
4			16	-	-	-	-	-	-	-	-
5	23K	48K	2	1.93	1.07	2337	3601	256.8	134.1	12620	1733
6			4	129.6	118	2093	1308	970.9	137.4	6883	804
7			8	-	-	-	-	-	-	-	-
8			16	-	-	-	-	-	-	-	-

Table 4.7: Partitioning test results on the Dragon models

	V	T	p	T _a (s)	T _s	S _a	S _s	C _a (t)	C _s	B _a (v)	B _s
1	15.7K	31.5K	2	0.619	0.025	137.9	243	577.1	129.5	7932	121.5
2			4	0.560	0.101	462.1	536	1100	125.0	4120	187.9
3			8	1.008	1.86	489.4	300	1678	117.6	2179	119.7
4			16	3.30	4.60	219.3	95	2355	80.1	1096	51.8
5	21.4K	42.8K	2	0.857	0.057	408.3	641	479.2	128.4	10902	320.6
6			4	0.796	0.257	757.9	824	1116	176.8	5725	382.4
7			8	2.37	3.73	690.4	392	1940	158.7	3015	185.6
8			16	7.40	8.54	301.5	118	2817	99.7	1517	71.1

Table 4.8: Partitioning test results on the Elephant [1-4], Grapple [5-6] models

Column T_a shows that ET increases with p and $|E|$. We also observe that it becomes impractical for $p > 16$. Furthermore, it seems to be very sensitive to mesh topology (see Table 4.7). For each object, there are threshold pair values (p , $|E|$) over which the algorithm's performance drops miserably. Thus two objects of the same size might have significantly different ET. And this had to be expected. Recall that the algorithm tries to form connected subsets and whenever the progression of one subset is locked, then the whole partition is rejected and a new one is started. For small p , the partitioning is less likely to lock; same conclusion for small $|E|$. Clearly, this rejection strategy is not acceptable. Furthermore, not only T_a can grow rapidly but the standard deviation T_s is

most of the time bigger than T_a ! This time, modifying the random part of the algorithm (seeding) to an optimal seed method would not make any difference. Only a relaxation of the subset connectivity constraint could improve the algorithm (see next algorithm).

The average standard deviation between subsets S_a increases from $p = 2$ to 4, peaks, and decreases from $p = 8$ to 16. This is normal since, as for the previous algorithm, building bisections is easy. The work gets complicated as p increases but S_a will eventually decrease as p increases since the average subset size n/p also decreases. Hence we can conclude that S_a converges with the inverse of p . As for the cut size C_a , it seems to behave exactly in $O(|E|)$ and more or less in $O(\log_2(p))$. Finally, interestingly enough, the difference (in %) between the average biggest subset size B_a and n/p stabilizes as p increases (around 10% to 15% depending on the objects). Therefore the algorithm maintains a fixed good subset balance with high p values.

4.4.3.3 Third algorithm (revised greedy) tests

p	T_a (s)	T_s	S_a	S_s	C_a (t)	C_s	B_a (v)	B_s	Ct_a	CT_s	CP_a	CP_s
2	0.0112	0.022	6.08	4.50	111	13.2	227.3	0.48	8.1	2.9	1.0	0.0
4	0.0105	0.021	9.85	2.03	196	34.1	115.2	1.31	13.5	6.4	3.2	0.44
8	0.0090	0.020	10.39	1.91	279	20.4	59.9	1.02	25.0	2.6	14.2	1.5
16	0.010	0.021	11.26	2.25	372	11.1	33.0	1.07	35.7	2.7	27.4	3.3
2	0.049	0.017	30.07	13.66	205	46.4	944.2	0.57	8.7	3.8	1.0	0.0
4	0.046	0.020	22.44	5.60	389	35.3	473.3	0.77	16.0	3.5	4.5	2.0
8	0.047	0.019	16.88	3.49	572	31.9	238.4	1.56	29.2	3.7	15.3	3.3
16	0.045	0.021	10.82	1.59	799	26.2	120.9	1.16	45.0	5.5	32.7	5.3
2	0.217	0.012	39.82	15.58	421	109.5	4073	0.94	12.9	5.3	1.0	0.0
4	0.209	0.021	32.29	8.86	805	86.3	2037	0.78	27.4	6.0	4.5	1.6
8	0.215	0.016	24.78	5.12	1220	60.7	1020	0.98	44.1	7.2	20.5	4.9
16	0.200	0.026	17.72	3.05	1718	58.7	511	0.91	67.4	8.9	49.1	7.2
2	0.983	0.017	76.30	32.35	847	227.3	17417	0.57	4.0	2.5	1.0	0.0
4	0.972	0.025	63.84	15.87	1616	165.2	8709	0.54	9.01	3.1	3.5	1.0
8	0.966	0.027	42.54	8.59	2463	132.4	4355	0.58	17.6	4.8	10.6	3.3
16	0.967	0.026	27.54	3.37	3488	123.0	2178	0.74	30.5	6.2	24.0	5.2

Table 4.9: Partitioning test results on the Bunny models

p	T_a (s)	T_s	S_a	S_s	C_a (t)	C_s	B_a (v)	B_s	Ct_a	CT_s	CP_a	CP_s
2	0.0515	0.014	1.59	1.15	183.9	52.8	1009	0.22	3.20	1.27	1.0	0.0
4	0.0484	0.018	3.80	1.81	366.2	37.9	505	0.86	8.52	3.65	3.4	0.8
8	0.0499	0.016	3.92	1.75	554.2	28.0	253	0.60	13.98	3.07	8.8	1.7
16	0.0494	0.017	5.42	1.60	791.5	14.6	128	1.14	26.54	3.72	22.9	3.2
2	0.317	0.023	1.46	1.13	475.0	98.2	6141	0.21	3.87	1.67	1.0	0.0
4	0.316	0.023	3.65	1.90	916.5	87.0	3071	0.60	9.48	3.13	3.5	0.9
8	0.316	0.023	4.95	1.67	1373	69.1	1536	0.79	16.87	3.80	10.4	2.4
16	0.310	0.026	4.14	0.84	2042	58.6	770	1.51	31.20	5.14	25.5	4.8
2	1.29	0.028	3.17	2.78	1245	224.8	25024	0.55	14.50	4.48	1.0	0.0
4	1.28	0.026	10.39	3.65	2284	234.2	12513	0.74	28.65	6.90	5.2	2.3
8	1.28	0.025	11.14	2.94	3489	200.1	6258	1.31	47.61	7.56	23.4	6.5
16	1.27	0.025	9.03	1.60	5073	180.2	3133	2.57	74.73	9.90	54.8	10.1

Table 4.10: Partitioning test results on the Duck models

p	T_a (s)	T_s	S_a	S_s	C_a (t)	C_s	B_a (v)	B_s	Ct_a	CT_s	CP_a	CP_s
2	0.153	0.023	16.9	16.1	275	114.6	2602	1.09	13.8	6.0	1.0	0.0
4	0.149	0.024	23.9	7.7	644	93.6	1302	0.93	29.8	6.2	13.3	5.1
8	0.149	0.025	23.1	4.4	1171	68.6	652	1.25	57.6	8.2	37.6	7.8
16	0.143	0.027	19.6	3.2	1656	78.3	328	1.16	84.7	8.5	65.0	6.7
2	0.664	0.018	19.6	21.6	634	244.0	11491	0.94	25.9	10.8	1.0	0.0
4	0.660	0.018	29.1	9.2	1364	239.6	5746	1.01	54.1	10.9	18.2	9.5
8	0.661	0.018	28.5	6.1	2505	182.9	2875	1.01	95.6	12.0	58.9	12.7
16	0.666	0.023	26.2	3.9	3578	136.6	1438	1.01	141.2	12.8	107.6	12.4
2	2.791	0.036	51.4	35.1	1360	570.8	50104	0.70	49.3	18.4	1.0	0.0
4	2.786	0.035	64.4	18.5	2969	493.1	25053	0.86	100.8	18.5	28.6	17.7
8	2.792	0.030	61.3	10.9	5310	372.6	12528	1.23	171.9	18.5	101.8	19.9
16	2.784	0.028	49.0	5.7	7674	323.3	6266	2.06	253.2	20.8	184.1	21.2

Table 4.11: Partitioning test results on the Dragon models

p	T_a (s)	T_s	S_a	S_s	C_a (t)	C_s	B_a (v)	B_s	Ct_a	CT_s	CP_a	CP_s
2	0.411	0.027	2.15	1.84	614	134.9	7864	0.42	4.8	2.7	1.0	0.0
4	0.406	0.027	3.54	1.56	1208	133.1	3933	0.64	11.6	4.1	3.7	1.1
8	0.405	0.026	4.16	1.02	1789	109.5	1969	1.42	20.4	5.5	11.6	3.1
16	0.406	0.027	4.80	0.94	2501	86.6	990	2.81	35.8	5.4	27.1	4.4
2	0.567	0.026	2.61	1.93	578	139.1	10698	0.64	5.6	3.7	1.0	0.0
4	0.561	0.023	3.75	1.83	1252	150.9	5349	1.06	9.8	4.5	3.8	1.8
8	0.561	0.023	4.02	1.19	2081	144.0	2676	1.03	19.3	5.7	12.9	4.4
16	0.560	0.022	4.69	1.32	2953	92.6	1339	1.50	34.1	6.9	27.6	5.7
2	1.28	0.025	17.75	14.6	1233	441.9	25043	0.44	15.9	6.8	1.0	0.0
4	1.26	0.017	27.90	5.74	2636	279.4	12522	0.92	35.4	9.3	9.9	6.0
8	1.25	0.023	20.14	5.04	4158	265.1	6262	1.51	61.8	10.5	31.5	9.2
16	1.25	0.023	18.05	3.52	5810	217.9	3132	1.52	96.7	11.3	62.0	10.7

Table 4.12: Partitioning test results on the Elephant [1-4], Grapple [5-6], Nefertiti [8-12] models

This version of the greedy algorithm has a totally different timing behavior. Because it never has to restart again. When a subset is locked in its progression, another new seed is chosen and the rest of the subset is grown from this seed as a new component of the current subset. This algorithm executes only one graph traversal of the mesh, and it is not affected at all by parameter p . When we consider T_a , we see that timings are all

constant for each object regardless of parameter p . We even observe that T_a tends to decrease slightly as p increases! This is due to the algorithm's internals. The inner growth loop in the algorithm is exited faster with higher values of p . During the subset progression phase, the loop is exited when vertices from other subsets are encountered (subsets colliding with each other). And this phenomena is more likely to happen when the number of subsets p is high (not to mention that this algorithm produces multiconnected subsets). A very striking result is that the timing standard deviation T_s , is fixed not only for different p , not only for different resolutions, but also for every different objects!

We recall that the last subset is a special case: it is built by collecting all the remaining unpartitioned vertices. Some might be part of a big component. Others might be isolated, stuck inside another subset or between many subsets. Those small components, composing the last subset, deteriorate the partition efficiency. But for the sake of subset balance, they have to be assigned to the last subset rather than to any other. Our metric S_a measures the average standard deviation between subset sizes of each partition. This metric is based on the size of the subsets, in triangles rather than in vertices, despite the fact that the partitioner tries to optimize the subset sizes in terms of vertices. We chose this metric (triangle count) since eventually, in our parallel software, the processor's workloads will be measured as the number of triangles sent to it, not the number of vertices (our software collapses edges, not vertices). Now, in the light of those explanations, we observe that S_a is not nearly equal to zero, although very small. This is

all due to the last subset. Its vertex count is equal to those of other subsets, but its triangle count is lower since its components are not always triangulations. They might even be as small as one or a few vertices; but connected in lines rather than in triangles... Nevertheless, these subset size deviations are very small and do not vary a lot (see S_s).

The cut size varies as for the previous algorithms in function of p and $|E|$. It should be understood however that since subsets are multiconnected, the proportion of triangles between subsets should be slightly higher than for previous algorithms.

Now follows the proof that the algorithm creates well-balanced subsets. The biggest subset size B_a is always within at most a dozen vertices more than the average subset size n/p . For sufficiently big meshes, the difference becomes meaningless. From column B_s , we observe that the deviation of the biggest subset is fairly small and constant for any mesh regardless of $n=|V|$ and p .

Finally, due to the connectivity of the partitions generated here, it was interesting to include a metric CX which analyses the component count of each partition. Due to the nature of the algorithm, we also split the metric into two specific ones: CP , which counts the number of components related to the first $p-1$ subsets and CT , which counts all components (CP + components of last subset). The first observation is that for any mesh, the algorithm yields $CP_a = 1.0$ and $CP_s = 0.0$ when $p = 2$. This means that the first subset ever built on a connected mesh is connected. There is no way this first subset could get locked by mesh topology or other subsets. There are not many observations possible for

CP_a . It seems to behave more or less the same for all objects except for the Dragon (Table 4.11) which we can now identify as our worst case (under all metrics). CP_s seems to be proportional to CP_a . CT truly becomes significant when considered under the difference $CP-CT$, the number of components of the last subset. We observe that this difference increases with p but converges to a constant. Furthermore, the standard deviation of this difference converges to zero as p increases. In other words, CT_s tends towards CP_s as p increases.

4.4.3.4 Comparison of algorithms

At first glance, it is quite obvious that the revised version (third algorithm) of the greedy algorithm (G) is more efficient than (second algorithm) its ancestor. So we reject the ancestor right from the start and compare the last two contenders, P (first algorithm) and G.

We can easily see that the difference of timing between the two varies in function of $|V|$ whereas P's timing also vary with p . For small objects, P takes on average five times more to compute a partition. And that ratio can go up to ten for bigger meshes when $p = 16$. We could expect that difference to decrease to less than two if the seeding phase of P was solved in a deterministic way.

In terms of subset size deviation, G wins handily. G builds well-balanced partitions all the time. The balance of the partitions built by P is a function of the quality of the seed

set provided to it. Once again, if the seeding phase was improved, the balance of partitions of P would improve (although never as good as G). Metric B is in direct relation with metric S . It accounts for the average biggest subset of a partition. Here again, G wins by producing the smallest B (its subsets are approximately all even). Whereas P builds biggest subsets up to 25% bigger than the average n/p (when $p = 16$).

As for edge-cut size, P yields the best results. But the reassuring fact is that the difference of C_a (between P and G) decreases to 10% when $p = 16$. We have to mention though that when $p = 2$, the difference of ratio can be as high as 100% depending on the object's topology (worst case: Dragon). Nevertheless, the analysis of this difference becomes meaningful when p is sufficiently high since partitioning for parallel software is rarely lower than four (even for coarse grain implementations). In that sense, G 's edge-cuts are not far off track if we consider integrating it into a parallel software. It all depends on how much the program is edge-cut size sensitive; in our case, not at all (see Chapter 5 and 6).

In light of this analysis, algorithm G is better than P under execution time and subset balance. P beats G on the cut size metric. But this is a meagre benefit to us since our parallel algorithm is insensitive to this metric (note that most parallel algorithms are though). Furthermore, G is superior to P in the sense that it can partition unconnected graphs (which P , as is, cannot). This particularity broadens the variety of meshes compliant for parallel processing or eliminates the need to check for multiple component

input meshes. For all these reasons, G has been chosen as the partitioning basis for our parallel algorithm.

4.5 Conclusion

In this chapter, we have presented many of the graph partitioning alternatives. We have seen the spectral methods using eigenvectors and Laplacian matrix of the graph to partition it. Geometric methods rely on mapping points to a higher dimension, and separating them with hyperplanes. Those were two of the three main classes of partitioning algorithms. Additionally, there exist more general but less practical partitioning schemes such as in [Fa98].

Then we examined optimization methods. Multilevel partitioning involves shrinking the graph, partitioning it (with any algorithm) and mapping the partition back to the original graph. Post processing methods such as the Kernigan-Lin algorithm are used to improve the quality of a partition.

Although we did not implement any other method than the greedy ones, [Ci94b] did compare the spectral and greedy methods. It turns out that the greedy method presented in the last section yields partitions of equal quality (even subsets, low edge-cut size) to those of spectral methods, but at a fraction of the execution time! But are the evenness of subsets and minimum edge-cuts synonymous with partition quality?

Although our parallel software is not at all concerned with communication costs, parallel algorithms are in general. Thus, this topic deserves a discussion. In a recent paper [He98], Bruce pushed further the issue of graph partitioning quality. Apparently, the standard partitioning approach suffers two shortcomings: faulty metrics and unnecessary constraints. Clearly the current scheme is advantageous for limiting the communication (in the underlying parallel application), but the edge-cut size is not the most important thing to minimize.

For example, contrary to what is often assumed, the total communication volume is not proportional to the edge-cut size, but to the number of vertices on the border of subsets. Moreover, on a more technical level, sending a message on a parallel computer requires a fixed latency time and the delivery time proportional to the message length. Graph partitioning tries to minimize the total volume, but not the total number of messages. And as we can expect, for all but very large graphs, latency delays can overcome the communication volume.

Another technical aspect of the problem springs from heterogeneous environments where each processor has distinct characteristics (in terms of CPU power, communication speed and size of local memory). An even data distribution may not mean even workload. Likewise, a minimum total communication cost may not mean an even communication cost between processors. Hence ideally there is a compromise work/communication load to be addressed. Ultimately, optimizing one will degrade the other. But algorithms of the

future will take this fact into account and derive adapted partitions. This 'intelligent' partitioner remains however quite a challenge to design.

Chapter 5

Parallel Mesh Simplification

Although the computational power of computers is continuously increasing, it is unlikely that they will ever reach the level of performance required to execute the complex procedures applied to constantly enlarging meshes. The scientific and industrial communities continuously come up with needs that surpass the current state-of-the-art computer systems. As we are about to discover, mesh simplification is such an operation, very demanding in CPU time. Its execution does not have to be real-time since it is a one-time offline operation. However, nobody would complain if the execution were cut from days to hours. This is where parallelism comes into play. But first, there are many parallelism issues to explore. A background introduction on parallelism is quite appropriate. Hence before we tackle the issue of designing a parallel mesh simplifier, here are some parallelism highlights beneficial to any parallel algorithm designers.

5.1 Parallelism at large

This shall serve the purpose of a quick introduction to parallelism issues. We will introduce the reader to the different kinds of parallelism. We will then explore the many related concepts. Finally, based on these observations, we will face the design choices that have to be dealt with when designing parallel software [Croc].

5.1.1 Different kinds

There exist three main classes of parallelism. Their distinction depends on which entity is to be distributed among processors. This entity might be functional, temporal or data related. The choice of which to parallelize is the first issue to address. How would a specific problem be parallelized best? Each problem has a stronger entity to parallelize, or one that parallelizes better than others. Furthermore, hybrid solutions can combine multiple forms of parallelism.

5.1.1.1 Functional parallelism

In this paradigm, the whole process applied to data is represented as an ordered series of distinct tasks. It suffices to assign those tasks approximately evenly (in terms of computation cost) to the available processing units and we get a pipeline-style application. The data is continuously fed piece by piece to the first unit. It processes the pieces and forwards them in order to the next processing unit. Basically, pieces of data

are passed from units to direct downstream neighbor units until they come out of the pipeline fully processed. The number of steps in the pipeline characterizes the degree of parallelism of this approach. Hence this method does not scale at all to an arbitrary number of available processors. Furthermore, the speed of the pipeline is limited by its slowest step, which can lead to a serious waste of CPU time at other steps (other units). Nevertheless, this method has proven worthy for a few specific applications such as graphics rendering among others.

5.1.1.2 Temporal parallelism

This parallelism is involved when continuous sequenced output has to be generated, piece by piece. Those resulting pieces are then associated by the application to distinct time frames (an obvious example is video playback). In this case, parallelism is obtained by decomposing the problem in the time domain. Processors are assigned data related to one or more time frames.

5.1.1.3 Data parallelism

This is the most common form of parallelism: data is split into a number of streams, which are in turn assigned to distinct processors. This method scales perfectly well in input data size and number of processors. Its only limitation is economic and technical in nature: how many processors can be incorporated in the system? Moreover, of critical importance is the communication network which routes data between processors.

Network characteristics have a significant influence on the application design decisions, as we will see.

5.1.2 Parallel algorithm concepts

Some algorithms parallelize trivially, requiring little communication or additional computations. Most parallel algorithms, however, introduce overheads, which are not present in their sequential counterpart. These overheads arise from many sources:

- communication pitfalls between processors
- uneven workloads
- redundant computations
- increased memory requirements for replicated or auxiliary data

To understand how these occur, we need to examine some key concepts of parallel algorithms.

5.1.2.1 Coherence

Coherence refers to the tendency for space or time neighboring features to have similar properties. Computer graphics for example relies on coherence to reduce computational load. Parallel algorithms must exploit coherence to reduce communication

costs and/or improve load balance. Otherwise, they will give rise to overhead not present in the sequential version.

5.1.2.2 Task/data decomposition

Data parallel algorithms are distinguished by how the problem is decomposed into workloads or tasks. The primary goal is to distribute approximately evenly the workload among the processors. The distribution of the workload has a subtler incidence on the communication: the choice of task decomposition has a direct impact on data access patterns. For example, in distributed-memory architectures, where remote memory accesses are usually expensive, task and data distribution are bound together. That is, data distribution must be optimized to reduce the communication flow between processors. This is less of an issue for shared-memory architectures. Nevertheless, good data locality achieves efficient caching on all architectures.

5.1.2.3 Granularity

Related to the concept of task and data decomposition is the notion of granularity. An algorithm's granularity is the qualification of the complexity of its most atomic unit of work. A computation (a task sprung from a program) is fine-grained if the workload unit is small and coarse-grained if it involves substantial processing. Granularity can also refer to data decomposition. Fine-grained data decompositions include few data items in many partition subsets. Coarse-grained decomposition exploits bulky data blocks. Granularity

has an impact on the efficiency of a parallel application. Fine-grained computations involve more scheduling and communication overhead, but enforce sharp load balancing. Coarse-grained computations tend to minimize overheads but increase the risk of load imbalance and may restrict the amount of available parallelism.

5.1.2.4 Scalability

Scalability of a parallel system refers to the ability of the application to adapt to any problem instance and any system size (number of processors). There are two kinds of scalability. Performance scalability is the ability to achieve higher levels of performance on a fixed sized problem (more processors). Data scalability is the ability to accommodate larger problem instances on a fixed system. Traditional shared-memory systems offer the potential for low overhead, but their performance scalability is limited by the contention of the communication system (which links processors to memory) and the number of processors themselves. In the distributed-memory architecture, processors and memory are tightly coupled. The system can be augmented at will, connected by a scalable network (although the risk of contention increases with the number of workstations). The CPU power and total memory scale linearly with the number of processors on the network. However, remote memory access remains very expensive.

5.1.3 Design & implementation issues

Taking those concepts into account, one has to consider the application at hand, evaluate its requirements, and resolve tradeoffs inherent to parallelism, in the most suitable way. We will consider those issues here.

How does the problem decompose itself? Fine-grained problems are best carried out on shared-memory systems. This system provides a global address space so there is no need for complex partitioning of the data. However, its architecture does not scale. An increase in the number of processors will cause higher memory latencies and communication contention. Therefore, to support additional processors, shared-memory algorithms must stress data locality (modern systems have cached processors), avoid memory hot spots and reduce the number of synchronization operations, as is done with distributed-memory algorithms. On the other hand, distributed-memory systems scale well. The drawback remains the heavy cost of remote memory access.

Communication between processors is a perpetual concern when designing parallel applications. The choice of algorithm inevitably has an impact on the volume and patterns of communication. When sending a message on the network, there are three timing aspects to consider: latency, bandwidth and contention. Latency is the time required to setup the communication (communication protocol stack traversal). Bandwidth is the nominal data flow on the network hardware per unit of time. Contention occurs when an application tries to inject more data on the network than it can absorb. In this case, a

bottleneck happens and contention is the time needed to ‘unplug’ the network. The sum of those three variables is the time required to send data from one processor to another. The value of these variables differs depending on the system in use. Hardware latencies are below the μ -second but the software communication layer’s are a few orders higher. Hardware bandwidth also can span from one Mb/sec to many Gb/sec (in dedicated graphics hardware). Contention is, however, unpredictable. It depends on dynamic traffic patterns of the application, the algorithm in use, the specific data input...

5.2 Different alternatives

As we have seen, parallelism can follow many paths in algorithm design depending on which decisions guided the hardware selection and software implementation. Two distinct granularities of parallelization can be considered for our problem: 1- fine-grain implemented on a collapse by collapse basis or 2- coarse-grain implemented on a whole mesh basis [Cort].

5.2.1 Fine-grain

Before collapsing, the algorithm evaluates the cost of every edge and ranks them in a priority queue. When the first collapse is performed on edge X, it will necessarily affect the neighborhood, its neighboring edges. The next collapse in the queue cannot be

performed until the new costs of the neighboring edges of X are recomputed. This step is necessary since the cost of a collapse is a function of its neighborhood, and has to be recalculated when that neighborhood changes. The queue must be reordered since the edges with updated costs would be inconsistently positioned in the queue. One of them might migrate to the top of the queue and eclipse the current best collapse. For this reason, collapses must be performed one after the other, sequentially, with queue reordering after each collapse.

However, collapses themselves can be parallelized. For example, once the collapse is done (a simple operation on the data structure), the computationally most intensive part of the collapse is in fact to recompute the cost of its neighboring edges. Different neighboring edges could be assigned to different processors for parallel cost recomputation (although the cost of communicating these edges on a network might weaken this motivation). Similarly, there are many points in the algorithm which could benefit from this fine-grain optimization. For example, an edge is optimized (for best vertex position) three times over three different vertex positions (both edge-ends and edge center). These three optimizations could be performed by three different processors.

5.2.2 Coarse-grain

By considering the fact that the impact of an edge collapse is limited to the edge star, it should be possible to perform collapses in parallel. Consequently, a more complicated approach would segment the mesh into a number of disjoint pieces, and

allow the collapses to be performed independently and then integrate the collapses back into the mesh. Here again the collapses can be distributed to processors in broadly two different ways.

An optimistic method begins by building the priority queue as usual and then assigns the edge collapses (and their neighborhoods *exclusively*) to available processors. Once the collapses are returned to the master processor, the queue would be updated with the recomputed neighboring edge costs. If there were a conflict between two concurrent collapses, one would be kept upon some criterion and the other thrown away (the collapses could be chosen from the queue to avoid collisions though, skipping the conflicting pairs). There are some minor difficulties with this method. First of all, when p slave processors are available, p non-intersecting collapses must be chosen from the top of the queue, and sent for processing. The master processor will not send another round of collapses until it receives the previous ones and updates the priority queue. Hence, all available processors will wait idle until the last one finishes its collapse. Furthermore, as stated earlier, collapses are simple data manipulations. Other intensive operations would benefit more from parallelization. Also, such a scheme requires a lot of communication, two messages per edge collapse, decreasing its efficiency. And finally, after each round of collapses, a broadcast message containing mesh updates should be sent to all processors, further delaying the program termination. This method works, but is not optimally efficient. Nevertheless, it paves the way to a coarser, more efficient scheme.

The galaxy method divides the mesh into *groups of adjacent* edge stars that cover the mesh completely. Each processor would then be assigned a group (of edge stars). Because every processor is aware of which processor any vertex, edge and face belong to, conflicts can be managed before performing collapses. The processors would all be given a big piece of the mesh. After simplification, the processors would all synchronize with each other and solve conflicting collapses. Then other rounds of simplification can be initiated over and over until all possible collapses have been executed. The advantage of this method is that data dependencies are easily resolved since most of the edge collapses have their entire neighborhood within their processor's mesh subset, exclusively local in the processor's memory. Few edge collapses involve updating neighborhoods split between many mesh subsets. Also, there is little communication needed since processors work independently and report to the master processor only. The incentive for that algorithm are enhanced scalability and execution speed inversely proportional to the number of mesh groups (or the number of slave processors).

5.3 Our version

As it might be inferred from the previous chapters, the design we opted for is mainly based on the galaxy method. We chose to privilege a simple brute-force method, e.g. greedy coarse-grain parallelism. We were looking for a solution where each slave processor simplifies a distinct piece of the mesh independently and returns its set of edge

collapses back to the master processor for merging. Here is the basic outline of our parallel algorithm:

```

Parallel_Simplification(Mesh M, PartitionSize p)

    if (ProcID == 0)                                //Master section
        (M1, ..., Mp) = Partition(M, p)
        for i=1..p
            send Mi to Proci
        for i=1..p
            receive PMi from Proci
        merge  $\forall$ PMi into PM
        return PM
    else                                            //Slave section
        receive MProcID from Proc0
        PMProcID = Simplify(MProcID)
        send PMProcID to Proc0

```

The first step is to partition the mesh. This task is carried out sequentially by the master processor. We wanted to make our algorithm (and implementation) as general as possible. In that sense, any partitioning algorithm can be used, and there are no restrictions regarding the partitions themselves other than optimality criteria (such as approximately even subsets and minimal edge-cut) the ability to derive arbitrary p-way partitions (for generality's sake) and the partition format specific to our algorithm (edge-separator partition).

Then the master processor sends to each slave its part of the mesh. The slaves consider these mesh parts as whole *open* meshes. They will process them transparently and send back their collapse sets to the master for merging of the sub-results. One must remember that this sketch is only the algorithm backbone and that the actual implementation is much less transparent (see next chapter).

5.3.1 How does it meet the parallel paradigm?

This algorithm works relatively well. It achieves the goals it has been designed for. That is, faster computations. But how well does it meet the different aspects of parallel efficiency?

We expected coherence to be a benefit of computer graphics for this application. Coherence played a major role in the design of this application. It has been employed notably to resolve partition border problems (see next section). Moreover, it is quite obvious that this algorithm is coarse grained and fully scalable, in data and processor space and provides an approximately even workload to all processors. We chose a distributed-memory architecture as the building ground of our application. How well did we adapt the application to this architecture?

We already know that this algorithm is a data parallel algorithm. The only separable entity here is the 3D data objects themselves. Mesh simplification is a one step offline operation that has to be performed thoroughly before the resulting Progressive Mesh can

be used. Thus, the pipeline model is just as irrelevant to our problem as time frame model. Data parallelism is generally the kind of parallelism employed to solve common problems.

The previous question could be rephrased as follows: How much overhead results from the parallel version? The first obvious overhead is the partitioning step, inherent to the parallel version. This step divides the data among the slave processors. And although it generates little overhead, it has been optimized to reduce the overhead in the rest of the algorithm. That is, its greedy nature quickly generates low quality partitions; optimized on only one requirement of interest, evenness of the partition subsets. That way, each processor will process a distinct, approximately equal, part of the mesh. Therefore, at the micro level, each processor will compute distinct edge collapses and edge costs. Although this statement may seem insignificant, it validates the fact that our parallel algorithm does not perform redundant calculations; calculations performed more often than in the sequential version. Redundant computations are hurdles that parallel algorithms can hardly overcome. They are a frequent source of overhead. Moreover, since our partitioner derives arbitrary p -way partitions, the algorithm can be run on an arbitrary number of slave processors to accommodate faster computations or lower memory consumption.

As we all know, there is a constant tradeoff (speed/memory usage) to consider when designing software. It is easy to minimize one at the expense of the other's growth. Redundancy can be found in computations and data storage. And wherever it exists, it is a weakness the designer has to minimize. Data redundancy is to be avoided as much as

possible since it will induce higher memory usage, possibly clog the virtual memory and degenerate the system performance causing excessive disk swapping. Our algorithm is absolutely free of redundant computations. Our implementation, however, has not been optimized for data redundancy. Useless data storage could be reduced in future versions (see next chapter).

The network is also thought of as a form of memory. It differs from conventional memory since it is shared between many processors and accessed through a distributed management scheme established by the algorithm structure, the necessary exchange of information between processors dictated by the problem. Technically, only a pair of processors (sender/receiver) can access the network at any time. Therefore, when designing an algorithm, one must make sure to avoid concurrent communications, or network contention, the cause of network bottlenecks. Since most algorithms rely on synchronous communications, bottlenecks dramatically slow down parallel program executions. On that issue, our algorithm is particularly efficient since it necessitates only two waves of communication; before and after the slave's task. On the first wave, the master sends the mesh partition to each slave. This step engenders intensive communication. However, it cannot lead to network contention since the master is the only sender; all slaves are receivers. Thus, all slaves receive their share of the mesh in order and start processing immediately after. The second wave is just the opposite: slaves all send back their edge collapse sets to the master. In this situation, there are p senders and one receiver. This could lead to a network bottleneck. But it does not because slaves do not send their collapses all at once. Although we provide approximately equal

workloads to every slave, a difference of a dozen collapses between two slaves might just be enough for the first one to finish transmit its collapses before the next one starts. In fact, our experiments never showed any concurrent communications.

5.3.2 Border problem

In spite of its slight additional overhead, the parallel version of the algorithm brought its share of algorithmic problems. The partition border is one of them. As with any parallel graph processing algorithms, the partition border management was our prime difficulty. Remember that the mesh is partitioned into p subsets so that: $\bigcup_{i=1}^p V_i = V$ and $V_i \cap V_j = \emptyset$ for $i, j \in [1..p]$ and $i \neq j$. Therefore, there are edges (and faces) between partition subsets (part of the edge-cut). In parallel algorithms, those edge-cut edges need to be dealt with just like subset edges. That is, during the parallel execution, at synchronization points, there must be an exchange of information between slaves (potentially through the master). Then either of the neighboring slaves will process shared edges using edge information from the other neighbors. This interdependence management scheme is costly but most applications are border-sensitive and require it. So economically, there is no way around this problem.

Fortunately, the graphs we are dealing with here do not represent electronic circuits or airplane routing systems. Our graphs are 3D graphics objects. The human eye accuracy sets the required processing level. Hence, *invisible* degradation of the optimal result is

allowed. In other words we chose to avoid the border problem and not to deal with edges in the edge-cut. We do not process them, we do not collapse them. After all, this is acceptable since many edges from the original mesh will not be processed as well, for various reasons. For example, some of them might be on the border of a hole in the mesh (missing triangles create a hole) or they might be non-manifold due to bad mesh filtering after the mesh generation step. By not collapsing the edge-cut, one might expect to see the mesh separator in full resolution when the progressive mesh is displayed at a coarse resolution. But the beauty of it is that none of that will happen, and the edge-cut will be indirectly simplified along with any other edge. This phenomenon is shown in the figure below:

Figure 5.1: Edge-cut face deletion

This partition snapshot represents the border between two subsets at some point in the simplification process. The thick lines represent the link of each subset (the edge envelopes of subsets). In this situation, edge e from subset A will be collapsed. The reader might say: “This edge has part of its neighborhood in subset B! You cannot collapse it” to which we would reply “Why not?”. The collapse would only affect the structure of subset A. The neighborhood part outside of A (vertices and edges of B) is only used to compute the best vertex position and the edge collapse cost but is never

modified by the collapse. Following the collapse, two faces from A are deleted, including one from the edge-cut (between subsets). This way, slaves processors can also simplify the edge-cut around their border. The parallel algorithm can simplify meshes as much as the sequential version does without the need for synchronization steps or communications between slave processors!

5.4 Conclusion

In this chapter, we discovered many new issues to consider when designing parallel algorithms absent from their sequential counterpart. Most of those issues are technical by nature. Moreover, parallel implementations remain platform dependent even though parallel libraries are standardized. Since parallel processing is still a young discipline evolving towards maturity, no unique standard has been proposed yet. At the logical level, parallel libraries are developed. But many architectures are available and the choice of architecture has a strong influence on the parallel algorithms, especially on the communication patterns.

When designing a parallel algorithm, one has to think about what will be divided among the processors (time slotted results, functions or data). Then what granularity applies to this entity. The answer to that question will essentially determine the algorithm's structure and the architecture on which to implement it. The decision was quite obvious for our mesh simplifier. Its functionality is hardly separable, so the mesh

(data) is the entity to divide. Also, the mesh cannot be divided into edges but rather as groups of adjacent edges. Those observations led us to a coarse-grained data parallel algorithm.

Finally, after the design phase, parallel pitfalls such as communication overload, redundancy, memory abuse and workload evenness were checked. This simple algorithm easily met most of them. The rest is left to optimize. Moreover the partition border problem was addressed in a very elegant way. This first attempt yielded a firm algorithm as accurate as the sequential one and optimally parallel (see tests in next chapter).

Chapter 6

Implementation, Tests & Analysis

Recall that what triggered this thesis was a need for a parallel mesh simplifier. A working sequential version had been implemented from Mr. Hoppe's research, and produced excellent results. But its appetite for CPU time and memory indicated that it could never service bigger *real world* meshes. The sequential simplifier submitted to me was written in C++ and contained many objects, related to either meshes or more common data structures. It is well encapsulated into objects and stands on a firm C backbone.

6.1 Implementation

The parallel implementation contains two parts. There is first, all the material related to mesh simplification, i.e. the sequential mesh simplification implementation.

Secondly, there are the parallel additions to this simplifier. In order to understand the parallel simplifier, we will first present a minimum technical overview of the sequential simplifier.

6.1.1 Sequential implementation

The mesh objects are divided into two groups: common data containers and specialized simplification objects. The former contains mesh information only and the latter contains additional features used during the simplification process, such as links to adjacent mesh elements. The objects are displayed with their data members (methods are not necessary for a clear understanding):

```
// Vertex Class
// Description:  A mesh vertex in 3D space.
class Vertex {
    double x, y, z;    // 3D coordinates
    double r, g, b;    // RGB color
};

// Face Class
// Description:  A triangular mesh face.
class Face {
    int    v[3];        // 3 indices to a mesh vertex array
};
```

```

// Edge Class
// Description:  An edge collapse record.
class Edge {
    int    s, t;        // 2 indices to a mesh vertex array
};

// Mesh Class Declaration
// Description:  A multi-resolution polygonal mesh.
class Mesh {
    int m;              // number of vertices in mesh
    int n;              // number of edges currently collapsed
    Vector<Vertex> v;    // vertex array
    Vector<Face> f;      // face array
    Vector<Edge> e;      // edge collapse array
};

```

The purpose of these objects is to keep the mesh information, that is to transfer it from file into memory. Then the application will build a working data structure of the mesh with the following objects:

```

// ConnectedFace Class
// Description:  A mesh face record containing connectivity data.
class ConnectedFace {
    Face* face;        // reference to the mesh face
    ConnectedEdge* edges[3]; // reference to 3 adjacent edges
};

```

```

// ConnectedEdge Class
// Description:  A weighted edge record containing connectivity data.
class ConnectedEdge {
    int v[2];                // indices of endpoints in the mesh array
    ConnectedFace* faces[2]; // reference to 2 adjacent faces
    int legal;               // non-zero if edge can be collapsed
    double cost;             // cost of performing edge collapse
    Vertex* vertex;          // optimal vertex position for collapse
};

```

These data structures are mutually intertwined with pointers to each other. After reading the mesh, the application will build an array of pointers to dynamically created `ConnectedFaces` objects (one for every mesh face). In the process, dynamic `ConnectedEdges` objects are also created. At the end, the working *connected* data structure is consistent with the original mesh. Then the program will rank all of the *legal* `ConnectedEdges` into the priority queue and enter the loop that will collapse them one by one until a stopping criterion is met (a user specified simplification threshold). Finally, the most complex object in the sequential implementation is probably the `Neighborhood` object:

```

// Neighbourhood Class
// Description:  The collapsing edge star: set of faces in a
//              connectivity graph which are affected by an edge
//              collapse.

```

```

class Neighbourhood {
    ConnectedEdge* collapsingEdge; // edge considered for collapse
    Vector<ConnectedFace*> faces;

    // set of faces sharing an endpoint with collapsingEdge
    Vector<ConnectedEdge*> edges;

    // set of edges sharing an endpoint with collapsingEdge
    Vector<Vertex*> perimeter;

    // set of vertices on the link of the edge star
    Vector<Vertex*> points;

    // set of points on original surface covered by the neighborhood
    double kappa;                // edge spring constant
};

```

This is a big object. There is one Neighborhood related to every ConnectedEdge, as the first data member implies. Fortunately, it is not a persistent object, as the Neighborhood object instances do not remain in memory very long. A dynamic Neighborhood object is created when an edge has to be evaluated for collapse. Many procedures are performed on it: legality tests, optimized vertex position computations and collapse cost evaluation. Once this is done, the results (cost and vertex position) are written into the related ConnectedEdge object and the Neighborhood object is deleted.

The Neighborhood constructor begins by visiting the faces around the collapsing edge, in the edge neighborhood. These faces are inserted in the face array along with the visited edges in the edge array. The different endpoints of those edges represent the link of the collapsing edge neighborhood (or edge star). These points are all inserted in the Perimeter array. Then comes the Points array which contains more points than the

Perimeter array. Recall that those neighborhoods are built throughout the simplification process and the current neighborhood might contain vertices which are in fact the result of previous edge collapses. The Neighborhood constructor has to retrieve all of the points from the original mesh which are covered by the current neighborhood. To achieve this, the constructor traverses the collapse set backwards, expands all the collapses which occurred within the neighborhood boundaries, and retrieves all those vertices covered by the neighborhood. When an edge neighborhood is built before any edge was collapsed, then the Perimeter and the Points arrays contain the same points. But as the simplification progresses, the size ratio Points/Perimeter grows. In fact Kappa is the spring factor based on this ratio. All the computations applied on the neighborhood object are at the core of the edge collapse optimization (energy function evaluation). They are computationally intensive especially at the end of the simplification when the Kappa ratio peaks. However, the implementation details will be skipped since they are not pertinent to this chapter and the theory has already been explained in Chapter 2.

6.1.2 Parallel extension

The source code additions to the sequential version are essentially in C language (although some methods have been added to existing objects). In the execution flow of the parallel version, the first lines are, just like for any parallel program, the parallel package initialization:


```

int main(int argc, char **argv)
{
    int    my_rank;

    int    cluster_size;

    // MPI standard setting up
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
    MPI_Comm_size(MPI_COMM_WORLD,&cluster_size);

    ...
}

```

As far as parallel packages are concerned, our available options for distributed memory architectures are either MPI and PVM. We chose MPI since it is the successor of PVM. More importantly, it has set the standard for message passing libraries. Consequently, it enforces the portability paradigm for our application (there is a plethora of efficient MPI for many different parallel platforms). Indeed, MPI, the Message Passing Interface, is a standardized and portable library of communications subroutines for parallel programming designed to function on a wide variety of parallel computers and network of workstations [MPI].

The MPI package we use is the free LAM-MPI [LAM]. Before launching our parallel program, the package must be pre-initialized via a command line in the console. This command uses as input a user file containing all the processor names available in the system. It creates the MPI processor pool. Then we can execute the parallel program. The

first source line initializes the MPI library. The second identifies the processor ID in variable *my_rank* (recall that the same code is submitted to every processor). The execution flow of the parallel program is strictly based on this ID number (especially to distinguish the master from the slaves). The third line reveals how many processors are executing the program (are available in the pool).

Following this preparation step, the program is ready to run. While the slaves are waiting, the master proceeds to partition the mesh into p subsets. The partitioner will return to the master a size $|V|$ integer array. Each array cell corresponds to a mesh vertex and contains the subset ID $\in [1..p]$ to which the vertex belongs. The next logical step is to send¹ that partition array to the slaves (abstractly to send each slave its subset of the mesh). Each slave receives the partition array. Then they all read the same mesh file into their Mesh object exactly like in the sequential version. Next they build their working mesh structures independently yielding dynamic ConnectedFaces and ConnectedEdges objects.

¹ - note that for any MPI communication, the receiver must allocate sufficient memory to receive the incoming message. Failure to do so will result in a program crash. We satisfied that condition in an efficient but inelegant way. Before sending any variable length message, we send a fixed length message (one integer) informing the receiver of how many bytes to allocate for the next message.

That step constitutes the major difference between the sequential and parallel versions. Recall that the slaves would simplify only their part of the mesh as if it were a mesh on its own. With the partition array in hand, the slave will build a working mesh structure of edges and faces which are either part of its partition subset or adjacent to it (surrounding edge-cut). Therefore, the slave's working structure will contain edges and faces whose vertex set intersects the slave's vertex subset. Next each slave will insert those edges in its edge collapse priority queue on one extra condition: each edge must have both end-points in the slave's vertex subset.

From that point, the slave's task remains the same as in the sequential program: it is left with a priority queue of edges to collapse and so it does. The working mesh structures and the priority queues have been cautiously prepared so that the simplification engine processes them as it would for a whole mesh.

After simplification, say a slave performed c edge collapses, it is left with four data containers. One is a Face Hashbag containing the faces which were not deleted during the simplification. Its counterpart is the face Stack which contains c pairs of deleted faces in reverse order of deletion (last on top). And finally, there are two arrays of c collapsed edges and c new replacing vertices, in order of edge collapse. This data must be transmitted to the master. However, LAM-MPI does not encapsulate complex data structures in communication streams. Moreover, it allows us to send messages of only a single data type at a time (arrays). Hence, a data conversion scheme had to be written to

break down our objects into streams of basic data types and back into objects again. Fortunately, our object classes have data members of one data type only. So we simply created a pair of conversion functions for every object class. One function for the *marshalling* of the object and one for the *unmarshalling*. The slaves would marshall the four containers into four single type arrays, and send them to the master.

The master does just the opposite. It will unmarshall the four arrays back into the four containers (hashbag, stack and arrays). Then the master is left with p sets of those four containers. The goal is to merge them into one set of four containers just as if the master had executed the simplification sequentially and returned from the simplification engine with those four containers. This task is not as simple as it seems.

First of all we should recall every slave is provided with a distinct vertex subset. The hull of that vertex subset (the link) encloses the set of faces owned exclusively by that slave, i.e. the faces whose three vertices all belong to the slave subset. But slaves also use the faces from the neighboring edge-cut, i.e. faces which have one or two of their vertices in the vertex subset. Hence, neighboring slaves have common faces in their working structures and will submit redundant faces to the master.

The slaves delete their edge-cut faces indirectly, so all of the face stacks generated by the slaves contain distinct faces. But what about the edge-cut faces they do not delete? Consider two neighboring slaves with their subsets X and Y . Consider also a face f in their common edge-cut, which has an edge in X 's subset (f has two vertices in X , and one

in Y). The possible scenarios could be the following: 1- $f \in \text{HashBag}_X$ and $f \in \text{HashBag}_Y$ (X did not delete f) or 2- $f \in \text{Stack}_X$ and $f \in \text{HashBag}_Y$ (X deleted f). The pattern depicted here involves the fact that every face from the edge-cut, shared between two subsets, will occur in the Hashbag of one slave and in the HashBag or Stack of the other.

The master's first task is to merge the HashBags into one master HashBag; which will be emptied in the cleared mesh face array (see Mesh class, Section 6.1.1). This structure is filled first with the faces which were not deleted in the simplification process by any slave (in accordance with the PM format). This basic set of mesh faces represents the Progressive Mesh in its coarsest representation. The order in which they are moved from the HashBags to the face array is irrelevant. Hence, we simply transfer the faces into the master HashBag, one HashBag after the other. We avoid face duplicates in the array by looking it up before inserting any new face (the HashBag is quite efficient on look-up operations). We finally get a duplicate-free HashBag of not deleted faces. Then the same array is once again searched for matching faces against all the faces in every face Stacks. When a face is found in both the master HashBag and a Stack, it means that one slave deleted that face and another saved it as 'not deleted'. Therefore, we choose to save the collapse by deleting the face in the master HashBag. Then all the faces in the master HashBag are copied in the empty mesh face array, and all HashBags destroyed.

Next, come the three other sets of containers (face Stacks, edge and vertex arrays) which must be extracted together since their items are all related by the same edge collapses (one edge collapse $\rightarrow$ two faces deleted $\rightarrow$ one new vertex). The sequential

algorithm implies a strict ordering relationship between those containers. That is after n collapses, the cell i in the edge and vertex arrays should correspond to the i^{th} collapse and cells $2(n-i)$ and $2(n-i)+1$ in the stack should correspond to the i^{th} pair of deleted faces. This ordering has been respected by each slave. But how do we maintain it when merging them in the mesh object?

We know the two arrays (vertex and edge) share the same exact ordering. Hence, we extract edges and vertices from this pair of slave arrays together. This operation enforces the ordering relationship between edges and vertices in the mesh object. We repeat this extraction operation, on collapses from the slave containers, in round robin. This way, we induce a cycle of edge collapses in PM, from all the slave collapse sets, in round robin. Traversal of the Progressive Mesh, from coarse to fine resolution, will continuously involve edge collapses from the same slave sequence. That is, we distribute equally the collapses of every slave over the whole resolution span on the Progressive Mesh. That traversal appears more natural than if the collapses of each subset were written contiguously in the PM collapse array. The 3D object would appear as if it were refined/coarsened by patches (slave subsets) rather than by individual edges.

There still remains the task of synchronizing the face Stacks with the mesh edge and vertex arrays. The slaves store deleted faces in stacks. The pair of faces on top of a stack represents the last collapse performed by a slave. In the PM format, the deleted faces must be inserted in the face array in reverse order of deletion, i.e. last deleted, first inserted. But the face insertion order is made more complex, since we also have to

observe the order of slaves from which the pairs (edge, vertex) have already been inserted in the Progressive Mesh.

We address that problem by recording that insertion order in a stack. The method is shown in Figure 6.1. The numbers in boxes identify the different collapses (vertices, edges and face pairs). Slave A has a face stack [1,2 with 2 being on top for this example. The vertex and edge arrays have already been merged in round robin order (1, 3, 6, 2, 4, 5) and written to PM. The slave order of collapse insertion is ABCABB (in terms of slave ID). Hence the order stack is [ABCABB. To determine the stack insertion order, the master pops the slave IDs from the order stack. It suffices to pop the faces in that order from the different face stacks to build the PM face array (5, 4, 2, 6, 3, 1) which is the exact reverse order of edge and vertex arrays in PM.

Figure 6.1: Merging of collapsed faces in PM

At this point, the mesh object is completely augmented to its PM representation. The last step is to dump it to a PM file.

6.2 Tests & analysis

To evaluate the quality and performance of our implementation, we performed a series of tests on a network of Linux-Intel workstations, each equipped with Pentium 120Mhz processors and 32Mb of RAM (some were PentiumMMX 166Mhz). The processor P_0 was the master processor responsible for managing the process, i.e. partitioning the mesh, distributing the tasks to every slave processor, collecting the collapse sets and merging them into one Progressive Mesh. We tested our program with 2, 4, 8, and 16 slave workstations (the network was limited to 19 workstations) connected by a 10Mbs Ethernet network. We ran these tests when most of the machines were idle but there is no guarantee that the timings were not influenced by other users on the network. Other than workstation characteristics, network specifications are also valuable when analyzing a parallel program execution. We could have further evaluated network timings in terms of latency and transmission speed with simple MPI tests programs. Also, our tests were conducted on two basic meshes: the Duck in 4K, 25K and 100K faces [NRCb], and the Dragon in 11K, 48K, 202K and 400K faces [Stan].

6.2.1 Timing analysis

As outlined in the previous section, the parallel program can be segmented into a series of sequential steps. We retained only the most relevant steps (file I/O was disregarded for example). Other than the timing metric, we might also want to verify our assumption that every slave workload is linear with the size of the mesh part submitted to

it. Hence we will verify that the approximately even mesh subsets match an approximately even number of edge collapses by slaves.

During our tests, each object (at each resolution) was processed five times in parallel. More than one sample execution is needed since the timing is likely to be influenced by the random nature of the partitioner. This many executions are a small sample indeed but due to the available time, no more experiments were possible. Therefore, in the following tables, the indices A and S stand for average and standard deviation on the current operation timing for those five tests. The next six timing check points were considered interesting to crop from the executions for display.

P: Partitioning time by master (includes mesh reading and partitioning).

PC: Partition transfer time to slaves.

S: Slaves processing time and sub-results transfer to master.

ΔS : time difference between slowest and fastest slave.

M: Merging time of sub-results from slaves by master.

T: Total execution time.

	 T_I/V_I 	p	PC_a / (PC_s)	S_a / (S_s)	ΔS_a / (ΔS_s)	T_a / (T_s)
1	4040 2031	1	-	602	-	618.0
2		2	0.0 0.0	283.2 42.6	71.0 57.8	295.6 41.1
3		4	0.2 0.4	133.0 4.1	51.6 16.2	150.4 4.2
4		8	0.2 0.4	66.8 8.6	35.0 7.8	84.2 8.2
5		16	1.2 1.5	30.4 1.3	18.0 2.0	49.2 2.4
6	24558 12281	1	-	4676	-	4774
7		2	2.0 0.0	2244 70.7	640 117.5	2350 72.1
8		4	2.4 0.5	1368 243.6	539 304.9	1473 242.6
9		8	3.8 3.1	642 79.1	319 91.4	747 80.6
10		16	8.4 6.3	322 30.8	178 40.5	445 36.4
11	100084 50044	1	-	25951	-	26345
12		2	13.4 0.5	11966 422	2792 619	12376 382
13		4	12.8 5.1	6920 136	2686 181	7321 145
14		8	15.2 12.0	3325 246	1553 311	3764 272
15		16	67.3 17.6	1813 119	979 128	2321 167

Table 6.1: Parallel Duck simplification statistics

	 T_I/V_I 	p	PC_a / (PC_s)	S_a / (S_s)	ΔS_a / (ΔS_s)	T_a / (T_s)
1	11102 5205	1	-	533	-	572
2		2	0.8 0.4	265.4 2.3	79.2 2.9	307.2 3.1
3		4	1.0 0.0	132.2 7.8	42.2 15.0	171.6 7.4
4		8	1.4 0.8	66.0 7.6	31.0 11.1	107.6 6.2
5		16	2.6 2.7	39.0 2.6	25.4 2.5	80.8 5.2
6	47794 22998	1	-	3639	-	3813
7		2	5.4 1.2	1878 310	560 362	2079 308
8		4	6.0 2.4	949 38	328 71	1141 39
9		8	6.8 3.6	462 22	178 36	657 28
10		16	11.2 12.4	233 13	108 17	425 12
11	202520 100250	1	-	32795	-	34811
12		2	52.8 28.4	15617 452	4057 161	16577 429
13		4	68.5 56.5	7158 70	2086 126	8211 174
14		8	82.8 1.8	3449 68	1260 127	4427 96
15		16	162.8 3.1	1776 75	709 99	2874 130
16	400000 198318	1	-	-	-	-
17		2	-	-	-	-
18		4	103 12	29640 1367	9483 2768	32103 1638
19		8	184 20	11338 58	3749 307	13714 59
20		16	369 73	5620 13	1873 131	8315 353

Table 6.2: Parallel Dragon simplification statistics

We decided to show only PC , S , ΔS and T since partitioning has been fully analysed in Chapter 4, and merging the sub-results is insignificant compared to the total time (except for our biggest mesh). Tables 6.1 and 6.2 are the results collected on the Duck and the Dragon models respectively.

As we can see (in the PC column), the partition communication time increases with the number of processors p and the number n of mesh vertices. It was predicted that this communication wave would not degenerate into network contention; there is one sender only, therefore no message collisions. As a consequence, those timings behave linearly, exempt of irregularities. Accordingly we obtained a partition communication time equivalent to $O(pn)$. Indeed, PC is cut in half when the program is executed from 16 to 8 processors. However, the values converge rapidly to a floor value when using fewer processors. This may be explained by the fact that sending the same message more times amortizes the communication setup time. The standard deviation is irrelevant. It is tight here and wide there, avoiding any logical pattern. This only means that the network was busy during some tests and more available at other times.

The next column S is of greater interest. It represents the time spent by processors from when they start reading the mesh to when they finish the simplification. More specifically, that timer is started when the master has finished sending the partition to slaves, and stopped when the master has received the last collapse set. Essentially, that metric determines the level of parallelism achieved in our application. What is troubling

here is that simplification by two processors takes on average less than half the time by the sequential program (one processor); this is impossible.

We must admit that the program timings were measured between breakpoints in the program. Therefore those timings possibly include system management, disk trashing and time slices from other users. So those numbers should be considered as *floating*. Furthermore, there was evidence that the workstations themselves were not tuned properly for maximum user throughput. But the best explanation for this mysterious speedup most likely comes from the fact that the parallel implementation never collapses as many edges as the sequential one (as opposed to what was stated in Chapter 5). Recall that the edge-cut edges are eliminated indirectly, not collapsed. There must be an adjacent edge, on the subset border, legal for collapse, for the edge-cut edges to be eliminated. Otherwise, they will remain in the mesh, contributing to a lesser simplification. Furthermore, due to how edge collapse costs are computed (see section 6.2.2), the last collapses always take much longer to compute (sequential complexity is $O(|E|^2)$). Since the parallel implementation does not perform those specific end-of-simplification expensive collapse cost calculations, it finishes faster than it should.

Again, the standard deviation is chaotic and meaningless since the slaves all have approximately the same even workload at every test. The theoretical deviation should be close to nil. But the circumstances of experimentation were not optimal. Recall that it takes only one workstation deprived of optimal conditions to derange the values of that column (average and standard deviation). The next column (ΔS) accounts for that

anomaly. It records the time between the first and the last slave to send their collapse sets to the master. The results show clearly that this difference can easily represent more than 50% of the total simplification time even though the workload is approximately even on each slave. But there is nothing alarming about that since like for most distributed memory environments, processors have different characteristics. Such parallel systems are called heterogeneous environments (120Mhz and MMX 166Mhz processors in our case).

The last column stands for the final test. It shows the total execution time including every step of the process (for some unknown reason, the file server on the system spent ten times more time writing than reading the same file. Hence, writing the PM files was, by far, the second longest step in the executions). The complexity of the sequential implementation evaluates to $O(|E|^2)$. The parallel implementation gracefully follows a linear timing pattern when compared to the sequential one; in spite of the extra partitioning step and inter-processor communications. Our parallel implementation is optimal. Its complexity is $O(|E|^2/p)$.

The reader probably noticed the blanks left in Table 6.2 in Line 16 and 17. Those tests constantly caused system crashes. So why do The tests with the same object succeeded in Line 18-20? This questions opens another aspect of parallelism which balances the well known compromise time/memory consumption. The system crashed for those tests because the memory requirements (at each machine) for sequential and 2-slaves parallel runs overcame the machine's capacities. The machines had enough

memory to process the mesh only then it was split in 4 or more subsets. This underlines the fact that parallelism not only speeds up computations but also reduces the memory demand on single machines by distributing the data over the whole aggregated memory in the system. This aspect of parallelism relieves memory-bound algorithms such as ours, allowing processing of bigger objects.

6.2.2 Quality analysis

As was stated earlier, due to the fact how edge collapse costs are computed, the last collapses always take much longer to compute. In the Neighborhood class (Section 6.1.1), the data member `Points` represents all the points from the original mesh covered by the current neighborhood. At the end of the collapse process, neighborhoods cover large parts of the original mesh (200 vertices or more is usual). The cost computation is proportional to that Point set size. The parallel program does not perform those last few expensive collapses. The number of these avoided collapses is a direct function of the edge-cut size. For example, on our biggest mesh, with 16 subsets, the parallel implementation collapsed 0.5% fewer edges than its sequential implementation (0.01% for two subsets). Notice that for p -way partitions, where $p > 2$, our partitioner produces a number of faces (proportional to p^2) which have their three points in three different subsets. Hence their three edges are never collapsed or eliminated in any way.

The partition border problem impacts again on the quality of the Progressive Meshes generated by the parallel implementation. Remember that the edge collapse cost

is evaluated on the Point data member in the Neighborhood class. Edges on the subset borders have such neighborhood points in the other adjacent subsets. This situation raises some efficiency concerns. We discuss how the problem evolves along with the simplification. At the beginning of the simplification process, before any edge is collapsed, each slave has in memory the whole mesh (which contains their exclusive subset). Say slave A has mesh subset M_a adjacent to M_b among others. Then in A's memory, M_b is the same as in B's memory. So when A computes the cost of a border edge with neighborhood points in B, then the computation is consistent with the vertices in M_b in B. However, when B happens to collapse some of its border edges, then M_b (initial mesh) in A's memory is no longer consistent with M_b in B's memory since collapses are not communicated between slaves. When A builds a neighborhood for one of its border edges, the neighborhood points from A's memory are all inserted in the Neighborhood Point data member. However, only the points from the initial mesh, in M_b , are included in Point (since A cannot access B's collapses). Therefore, slaves mutually bias their border edge cost computations. Bias might be an overstatement though since the points from the initial mesh are a sufficiently strong basis (use of coherence) to compute an edge cost and, anyway, all additional vertices are derived from those initial mesh points. Figure 6.2 shows snapshots of a Progressive Mesh which supports those claims. The border problem does not really prevent our parallel simplifier from producing meshes of very good visual quality.

100071 faces

63160 faces

36474 faces

19316 faces

10094 faces

Figure 6.2: Duck in Progressive Mesh version at different resolutions

6.3 Improvements

This first implementation generates very good results, but has many minor weaknesses. The parallel extension to the sequential simplification engine could be improved to handle larger meshes and produce even better PM quality.

The parallel implementation is based on the sequential simplification engine. Therefore, unless the engine is optimized for faster execution, there is nothing we can do about the parallel implementation speed. The workload is already optimized to be approximately even. Really? Recall that our implementation is meant to be portable on any parallel platform. As we have seen, heterogeneous environments are the most common systems these days. Therefore, the processors capabilities are not always equal (mostly in terms of CPU clock speed). One improvement to our implementation would be somehow to inform our partitioner of the different processors capabilities and load them with a task proportional to their capabilities in order to try to minimize the values in the ΔS column. Furthermore, there is also a much more obvious improvement to bring better speedup to the application: minimize disk trashing by minimizing memory usage at every processor. Those issues were well addressed in [Mo98].

Each slave loads the entire mesh in its memory even though the working data structure is composed only of faces and edges related to only its subset. Thus, the

background mesh structure in memory could be further downsized. For example, a mesh with 500K faces (12 bytes/face) and ~250K vertices (48 bytes/vertex) requires 18Mb (6Mb+12Mb) for that structure (not to mention the arrays of pointers, 500Kb×4 + 250Kb×4). However, when the mesh is partitioned into 16 subsets, the memory requirement would drop to less than 1.2Mb. This is very suitable on a simple 32Mb workstation since the working data structure consumes even much more memory, and other processes may also be running on the workstation. All there would be to do is to augment the Vertex and Face classes with an index data member corresponding to the index of the face and vertex in the master data structures. The working data structure could not lookup vertices and faces in constant time anymore, but lookup operations could be facilitated using hashbags rather than arrays.

On the quality issue, other improvements are possible. For example, we could toss away coherence in border edge cost computations and implement a real communication scheme between slaves so that whenever a slave needs updates on outside points from another slave, the former could probe the latter and receive a full collapse update. There is a strong assumption that the execution time of such a parallel implementation would crumble dramatically, though.

In the sequential implementation, the user specifies the desired number n of edge collapses in the resulting PM. Then the program generates the edge collapse priority queue and selects the top n edges (the less expensive ones). Consequently, the generated PM remains as faithful (at any resolution) to the original mesh. In our parallel

implementation, however, each slave receives $1/p$ of the mesh. Hence, with a well distributed workload, each slave should perform the n/p less expensive collapses. This is not the case with the current implementation: we force a full simplification (collapse all legal edges). However, we ultimately want each slave to receive a subset containing approximately n/p of those less expensive edges. Hence edges should first all be weighted by the master and then the partitioner should be fed this information along with the mesh so that it could derive partitions with n/p of those desired edges in each subset. This remains hard to achieve since edge costs change when neighboring edges are collapsed. One solution again would be to implement a slave-master synchronisation scheme. The master would accumulate all the collapses in an ordered array (sorted on the collapse cost). Then when the n collapse mark is reached, the master would stop those slaves whose top next collapse is more expensive than the n^{th} accumulated one. That way, only the slaves containing potentially less expensive collapses would keep collapsing. But then again, the workload would not be even. We can anticipate that if the partitioner derives partitions with an equal distribution of the edge cost in every subset, then the subsets will remain balanced on this cost criterion throughout the simplification. Although the cost of an edge changes when its neighborhood changes, we observed that the costs of close edges tend to converge toward local averages as the simplification proceeds. The cost of edges neighboring a collapse changes, but it all evens out in general.

Let us consider for a moment that only the n less expensive collapses are returned to the master once the simplification step is completed. There remains the problem of merging them. Each slave returns its collapse set into an ordered array, from the least to

the most expensive collapse. So far we just merge them in round-robin fashion assuming that collapses of same position in each array have more or less the same cost. But this is not true; the average cost in every subset is strongly determined by the geometry of the mesh subset. For example, a subset embedded in a flat mesh region is most likely to have a lower average collapse cost than subsets embedded into rough bumpy regions. Therefore, to be consistent with the PM format, collapses should be merged into the final collapse array, and then sorted on their cost value. That problem is minor, since once again, the human eye could not tell the difference. The degradation of the PM as the resolution decreases is not worse than what would be produced by the sequential simplifier. Besides, fixing that problem is easy. It suffices to add to the collapse sets (from the slaves) the cost of each collapse so that the master could merge them correctly. The slaves could as well send to the master a fifth data container, a cost array.

Finally, to perfectly simulate the sequential implementation, after receiving the collapse sets and merging them, the master should rebuild the priority queue for those remaining legal edges which could not be collapsed and collapse them.

6.4 Summary & conclusion

We derived a parallel implementation of the Progressive Mesh simplifier based on the sequential implementation. The general idea was quite simple: shred the mesh into slabs the slaves are fed with for parallel processing. So we moved on with the most

obvious way to parallelize graph processing applications: we implemented a general graph partitioning scheme and assembled it with the mesh simplifier. We wrote it in C and used the standard MPI parallel package for the implementation.

We soon realized that there was more to parallel programming than nice algorithms. A lot of experimentation was necessary before we could devise a program that worked and was of industrial strength. We also realized that the partition border problem can sometimes be avoided at the cost of a slight degradation of the quality or exactness of the result.

Nevertheless, our implementation has proved to yield the same results as the sequential implementation, and at an optimal speedup. Similar work can be found in [Fa98, Mo98].

Chapter 7

Conclusion

In this thesis, we presented contributions in the area of parallel mesh processing. We specifically studied, designed and implemented a parallel mesh simplifier. Our study first led us to believe that parallel mesh simplification necessitates a general graph partitioning method.

We first explored the different alternatives in graph partitioning and mesh simplification theoretically. We found many general partitioning algorithms, which could all have filled our needs. Those partitioning needs being modest, we opted for a simple and fast greedy mesh partitioner. On the mesh simplification side, many methods are available, focusing on different goals or different mesh quality aspects. But our choice of method was set right from the start since this project was first initiated from a previous sequential implementation of the mesh optimization method.

Then we derived an algorithm to integrate both into a parallel mesh simplification algorithm. Naturally, an implementation sprung out of this algorithm. The runs we performed on many mesh models, mesh sizes, and number of processors confirmed our assumptions. In theory, our algorithm is optimal and yields simplified meshes just as accurate as the sequential simplifier does. In practice though, the program's behavior (timing) is more or less 'moody', irregular, due to the dynamic state of the network of workstations (current network and CPU load).

There are, however, still many possible improvements. One which is outside of the scope of this work is to slightly modify the simplification engine to embrace arbitrary meshes of any dimension, manifold or not.

Advances in our ability to process large complex meshes are likely to come as much from increases in algorithmic efficiency as from hardware capability over the next decade. Of particular interest is the development of algorithms which are well suited to the current trends in computer hardware. Parallelism is the corner stone of this vault. It grasps all those optimization aspects at once.

Bibliography

- [Baj] CL. Bajaj, D. Schikore. *Error bounded reduction of triangle meshes with multivariate data*. SPIE 2656 :34-45.
- [Bop] R. Boppana. *Eigenvalues and Graph Bisection: An Average Case Analysis*. 28th Annual Symposium on Foundations of Computer Science, IEEE, pp. 280-285, 1987.
- [Bra] Gilles Brassard, Paul Bratley. *Fundamentals of Algorithms*. Prentice-Hall, Chapter 10, 1996.
- [Chaco] Bruce Hendrickson, Robert Leland. *Chaco User's Guide Version 2.0*. Report SAND95-2344. Sandia National Laboratories, 1995.
- [Ci94a] P. Ciarlet, F. Lamour. *An Efficient Low Cost Greedy Graph Partitioning Heuristic*. CAM Report 94-1, UCLA, Department of Mathematics, 1994.
- [Ci94b] P. Ciarlet, F. Lamour. *Recursive Partitioning Methods and Greedy Partitioning Methods: a Comparaison on Finite Element Graphs*. UCLA, Department of Mathematics, 1994.
- [Ci97] Tony F. Chan, P. Ciarlet, W.K. Szeto. *On the Optimality of the Median Cut Spectral Bisection Graph Partitioning Method*. SIAM J. Sci. Comput., vol 18, no. 3, pp.943-948, 1997.
- [Ciam] Andrea Ciampalini, Paolo Cignoni, Claudio Montani, Roberto Scopigno. *Multiresolution Decimation Based on Global Error*. The Visual Computer 13 pp. 228-246, Springer-Verlag, 1997.
- [Coh] J. Cohen, A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F. Brooks, W. Wright. *Simplification envelopes*. Computer Graphics Proceedings Annual Conference Series, SIGGRAPH 96, ACM Press, pp. 119-128, 1996.
- [Cort] Brian Cort. *An Implementation of a Progressive Mesh Simplification Algorithm*. Institute for Information Technology Group, NRC Canada, 1997.
- [Croc] Thomas W Crockett. *Parallel Rendering*. Technical Report ICASE-95-31, Institute for Computer in Science and Engineering, NASA Langley Research Center, 1995.
- [Cv80] D. Cvetkovic, M. Doob, H. Sachs. *Spectra of Graphs*. Academic Press, New York, 1980.
- [Cv88] D. Cvetkovic, M. Doob, I. Gutman, A. Torgasev. *Recent Results in the Theory of Graph Spectra*. Anals of Discrete Math., vol. 26, North Holland, 1988.

- [Deer] Michael Deering. *Geometry Compression*. Computer Graphics, SIGGRAPH 95 proceedings, pp. 13-20, 1995.
- [Ep93] K.L. Clarkson, D. Eppstein, G.L. Miller, C. Sturtivant, S.H. Teng. *Approximating Center Points With and Without Linear Programming*. Proceedings of 9th ACM Symposium on Computational Geometry, pp. 91-98, 1993.
- [Ep96] K.L. Clarkson, D. Eppstein, G.L. Miller, C. Sturtivant, S.H. Teng. *Approximating Center Points with Iterated Radon Points*. Internat. J. Comput. Geom. Appl., #6, (1996), pp. 357-377.
- [Fa98] Lamis M. Farrag. *Application of Graph Partitioning Algorithms to Terrain Visibility and Shortest Path Problems*. MCS thesis, Carleton University, 1998.
- [Fab] Fabio Guerinoni. *Mesh Partitioning Techniques and New observations for 3-Regular Graphs*. Technical Report #2623, Institut National de Recherche en Informatique et en Automatique, 1995.
- [Fi73] M. Fiedler. *Algebraic Connectivity of Graphs*. Czechoslovak Math. Journal, #23 (1973), pp. 298-305.
- [Fi75] M. Fiedler, *A Property of Eigenvectors of Nonnegative Symmetric matrices and its Applications to Graph Theory*. Czechoslovak Math. Journal, #25 (1975), pp. 619-633.
- [Fid] C.M. Fiduccia, R.M. Mattheyses. *A Linear-Time Heuristic for Improving Network Partitions*. Proceedings of the 19th IEEE Design Automation Conference, IEEE, pp.175-181, 1982.
- [GE] *Visualizing Large Geometry Models*. GE Research&Development Center, GE Corporate Research & Development.
- [Gi] Feng Cao, John R. Gilbert, Shang-Hua Teng. *Partitioning Meshes with Lines and Planes*.
- [Gi94] John R. Gilbert, Gary L. Miller, Shang-Hua Teng. *Geometric Mesh Partitioning: Implementation & Experiments*. SIAM Journal on Scientific Computing, Vol. 19, #6, pp. 2091-2110 and Technical Report CSL-94-13, Xerox Palo Alto Research Center, 1994.
- [Gi95] Tony F. Chan, John R. Gilbert, Shang-Hua Teng. *Geometric Spectral Partitioning*. <ftp://ftp.math.ucla.edu/pub/camreport/cam95-5.ps.gz>.
- [Gol] G. Golub, C. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 1989.
- [Guez] André Guézic. *Surface Simplification Inside a Tolerance Volume*. Technical Report RC 20440, IBM T.J. Watson Research Center, 1996.
- [He92] Bruce Hendrickson, R. Leland. *An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations*. Technical Report Sand 92-1460, Sandia National Laboratories, 1992.

- [He93] B. Hendrickson, R. Leland. *Multidimensional spectral load balancing*. Technical Report 93-0074, Sandia National Laboratories, 1993.
- [He95] Bruce Hendrickson, Robert Leland. *A Multilevel Algorithm for Partitioning Graphs*. Proceedings of the 1995 Supercomputing Conference, ACM, IEEE, 1995.
- [He98] Bruce Hendrickson. *Graph Partitioning and Parallel Solvers: Has the Emperor No Clothes?*. Proc. Irregular 98, Springer-Verlag, pp. 218-225 and <http://www.cs.sandia.gov/~bahendr/partitioning.html>.
- [Heck] P. Heckbert, M. Garland. *Survey of polygonal surface simplification algorithms*. Technical Report CMU-CS-95-194, Carnegie Mellon University, 1995
- [Hink] P. Hinker, C Hansen. *Geometric Optimization*. Proceedings of IEEE Visualization 93, IEEE Computer Society Press, CA:189-195, 1993.
- [Hop93] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, Werner Stuetzle. *Mesh Optimization*. Technical Report 93-01-01, Dept. Of Computer Science & Engineering, University of Washington and SIGGRAPH 93 Proceedings, pp. 19-26. 1993.
- [Hop95] M. Eck, TD. Rose, T. Duchamp, H. Hoppe, M. Lounsbery, W. Stuetzle. *Multiresolution analysis of arbitrary meshes*. Computer Graphics Proceedings Annual Conference Series, SIGGRAPH 95, ACM Press, pp.173-181, 1995.
- [Hop96] Hugues Hoppe. *Progressive Meshes*, Microsoft Corporation and SIGGRAPH 96, 1996.
- [Hop97a] Hugues Hoppe, Jovan Popovic. *Progressive Simplicial Complexes*. Computer Graphics, SIGGRAPH 97 Proceedings, pp. 217-224, 1997.
- [Hop97b] Hugues Hoppe. *View-Dependent Refinement of Progressive Meshes*. Computer Graphics, SIGGRAPH 97 Proceedings, pp. 189-198, 1997.
- [Hop98] Hugues Hoppe. *Efficient Implementation of Progressive Meshes*. Microsoft Corporation Report MSR-TR-98-02 and Computer & Graphics, 1998.
- [Hor] Simon H. Horst, Teng Shang-Hua. *How Good is Recursive Bisection?*. SIAM, Journal of Scientific Computing, Vol. 18, No 5, pp.1436-1445, 1997.
- [Kalv] AD. Kalvin, R. Taylor. *Surfaces: polygonal mesh simplification with bounded error*. IEEE Computer Graphics Applications, 16:64-77, 1996.
- [KL] B.W. Kernighan, S. Lin. *An efficient heuristic for partitioning graphs*. The Bell System Technical Journal, volume 49 #2, 1970.
- [Kle] R. Klein, G. Liebich, W. Strasser. *Mesh Reduction with Error Control*. Proceedings of Visualization 96, IEEE Computer Society Press, CA:311-318.
- [LAM] <http://www.osc.edu/search/> moved to <http://www.mpi.nd.edu/lam/>
- [Lee] Lee Willis, Virtual Landscape Dermatologist, employee for Terrex, a 3D Terrain generation software company, <http://www.terrex.com/>.

- [Lit] Nathan J. Litke. *A Continuous-Resolution Model for LOD Approximation*. NRC Canada, Visual Information Technology, 1997
- [Mi95] S. Guattery, G. Miller. *On the Performance of Spectral Graph Partitioning Methods*. Proc. 6th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 233-242, 1995.
- [Mi98] Stephen Guattery, Gary L. Miller. *Graph Embeddings and Laplacian Eigenvalues*. ICASE Report #98-23, Institute for Computer Applications in Science and Engineering, NASA Langley research center, 1998.
- [Mo91] B. Mohar. *The Laplacian Spectrum of Graphs*. J. Wiley, New York, pp. 871-898, 1991 and 6th Intl. Conf. Theory and Applications of Graphs, Kalamazoo, 1988.
- [Mo92] B. Mohar, S. Poljak. *Eigenvalues in Combinatorial Optimization*. preprint, 1992.
- [Mor98] Patrick R. Morin. *Two Topics in Applied Algorithmics*. MCS thesis, Carleton University, 1998.
- [MPI] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, Jack Dongarra. *MPI: The Complete Reference*. The MIT Press, 1997.
- [NRCa] http://www.vit.iit.nrc.ca/Pages_Html/English/Sensing.html
- [NRCb] Dr. Gerhard Roth, Visual Information Technology Group, Institute for Information Technology, National Research Council of Canada
- [Nu88] Doron Nussbaum. *Directional Separability in 2D & 3D Space*. MCS thesis, Carleton University, 1988.
- [Otto] Olivier C. Martin, Steve W. Otto. *Combining Simulated Annealing with Local Search Heuristics*. In G. Laporte and I. Osman, editors, *Metaheuristics in Combinatorial Optimization*.
- [Pot90] A. Pothens, H. Simon, K. Liou. *Partitioning sparse matrices with eigenvectors of graphs*. SIAM J. Matrix Anal., #11(1990), pp. 430-452.
- [Pot97] Alex Pothens. *Graph Partitioning Algorithms with Applications to Scientific Computing*. Technical Report TR_97_03, Old Dominion University, Department of Computer Science, 1997.
- [Pow] D. Powers. *Graph Partitioning by Eigenvectors*. Linear Algebra Applications, #101(1988), pp. 121-133.
- [Re90] F. Rendl, H. Wolkowicz. *A Projection Technique for Partitioning the Nodes of a Graph*. Technical Report CORR 90-20, University of Waterloo, Faculty of Mathematics, 1990 and Ann. Oper. Res., #58 (1995), pp. 155-180.
- [Re94] J. Faulkner, F. Rendl, H. Wolkowicz. *A Computational Study of Graph Partitioning*. Math. Programming, #66 (1994), pp. 211-240.
- [Ro] R. Van Driessche, D. Roose. *An Improved Spectral Bisection Algorithm and its Application to Dynamic Load Balancing*. Parallel Computing, #21, pp. 29-48, 1995.

- [Ros] J. Rossignac, P. Borrel. *Multi-resolution 3D approximation for rendering complex scenes*. Geometric modeling in computer graphics, Springer, pp.455-465.
- [Roth] Gerhard Roth and Eko Wibowoo. *An efficient volumetric method for building closed triangular meshes from 3-D image and point data*. Report NRC 41544 and Proceedings of Graphics Interface 97, pp. 173-180, May 1997.
- [Sav] José G. Castaños, John E. Savage. *The Dynamic Adaptation of Parallel Mesh-Based Computation*. Technical Report CS-96-31, Department of Computer Science, Brown University, 1996.
- [Sc92] WJ. Schroeder, JA. Zarge, WE. Lorensen. *Decimation of triangle meshes*. ACM Computer Graphics, proceedings SIGGRAPH 92, 1992.
- [Si91] H. D. Simon. *Partitioning of Unstructured Problems for Parallel Processing*. Conference on Parallel Methods on Large Scale Structural Analysis and Physics Applications, Pergamon Press and Computing Systems in Engineering, #2 pp. 135-148, 1991.
- [Si93] T. Barnard, H.D. Simon. *A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems*. Proceedings of the 6th SIAM Conference on Parallel Processing for Scientific Computing, pp.711-718, 1993.
- [Spi] D.A. Spielman, S.H. Teng. *Spectral Partitioning Works: Planar Graphs and Finite Elements Meshes*. Technical Report UCB CSD-96-898, University of California, Berkeley, 1996 and Proc. 37th Annual IEEE Symposium on Foundations of Computer Science, 1996, pp. 96-105.
- [Stan] <http://www-graphics.stanford.edu/data/3Dscanrep/>
- [Sw] Swen Campagna, Leif Kobbelt, Hans-Peter Seidel. *A General Framework for Mesh Decimation*. University Erlangen-Nürnberg.
- [Sw98] Swen Campagna, Leif Kobbelt, Hans-Peter Seidel. *Efficient Decimation of Complex Triangle Meshes*. Technical Report 3/98, University Erlangen-Nürnberg, 1998.
- [Th93] G. L. Miller, S. H. Teng, W. Thurston, S. A. Vavasis. *Automatic Mesh Partitioning, in Sparse Matrix Computations: Graph Theory Issues and Algorithms*. IMA Volume in Mathematics and its Applications, Springer-Verlag, Vol. 56, pp. 57-84, 1993.
- [Th98] G. L. Miller, S. H. Teng, W. Thurston, S. A. Vavasis. *Geometric Separators for Finite Element Meshes*. SIAM Journal of Scientific Computing, Vol. 19, pp. 430-452, 1998.
- [Turk] G. Turk, M. Leroy. *Zippered polygon meshes from range image*. ACM Computer Graphics, 28:311-318, 1994.

* Mr. Hoppe's publications and more are available at <http://www.research.microsoft.com/~hoppe/>