UNIVERSITÄT SIEGEN

Computergraphik und Multimediasysteme

Efficient and High Quality Clustering

Effiziente und Hochqualitative Clusterbildung

vom Fachbereich Elektrotechnik und Informatik
der Universität Siegen

zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)

genehmigte Dissertation

von

Iurie Chiosa
Siegen, Juli 2010

1. Gutachter:	Prof. Dr. Andreas Kolb
2. Gutachter:	Prof. Dr. Mario Botsch

Tag der mündlichen Prüfung: 23 September 2010

Gedruckt auf alterungsbeständigem holz- und säurefreiem Papier.

Revision 1.2.4

Abstract

Clustering, as a process of partitioning data elements with similar properties, is an essential task in many application areas. Due to technological advances, the amount as well as the dimensionality of data sets in general is steadily growing. This is especially the case for large polygonal surface meshes since existing 3D geometry acquisition systems can nowadays provide models with up to several million triangles. Thus, fast and high-quality data and polygonal mesh processing becomes more demanding.

To deal with such a huge and diverse heap of clustering problems efficient algorithms are required. At the same time the resulting clustering quality is of highest importance in almost all situations. Thus, identifying an optimal tradeoff between efficiency and quality is crucial in many clustering tasks.

For data clustering tasks in general as well as mesh clustering applications in particular, *k-means* like techniques or *hierarchical methods* are used most often. Nonetheless, these approaches are deficient in many respects thus a considerable amount of work is still required to improve them.

This dissertation describes new feasible solutions for efficient and high quality mesh and data clustering. It addresses both the algorithm and the theoretical part of many clustering problems, and new clustering strategies are proposed to overcome inherent problems of the standard algorithms.

With the advent of general-purpose computing on Graphics Processing Units (GPU), which allows the usage of a highly parallel processing power, new tendencies emerged to solve clustering tasks on the GPU.

In this work, a first GPU-based mesh clustering approach, which employs mesh connectivity in the clustering process, is described. The technique is designed as parallel algorithm and it is solely based on the GPU resources. It is free from any global data structure, thus allowing an efficient GPU implementation and a further step in parallelization.

Based on the original concepts a GPU-based data clustering framework is also proposed. The formulation uses the spatial coherence present in the cluster optimization and in the hierarchical merging of clusters to significantly reduce the number of comparisons in both parts. The approach solves the problem of the missing topological information, which is inherent to the general data clustering, by performing a dynamic cluster neighborhood tracking. Compared to classical approaches, our techniques generate results with at least the same clustering quality. Furthermore, our technique proofs to scale very well, currently being limited only by the available amount of graphics memory.

Zusammenfassung

Clusterbildung, als Prozess der Gruppierung von Datenelementen mit ähnlichen Eigenschaften, ist eine grundlegende Aufgabe in vielen Anwendungsbereichen. Aufgrund technologischer Fortschritte nimmt die Menge sowie die Dimensionalität der Daten stetig zu. Dies ist insbesondere für große Polygonnetze der Fall, da aktuelle 3D-Scanner Modelle mit bis zu mehreren Millionen Dreiecken liefern können. Entsprechend steigt die Nachfrage nach einer schnellen und hochqualitativen Verarbeitung von Daten und Polygonnetzen.

Um die große Menge an vielfältigen Problemen im Bereich der Clusterbildung lösen zu können, sind effiziente Algorithmen erforderlich. In den meisten Fällen ist hierbei die Qualität der Ergebnisse sehr wichtig. Für viele Anwendungen ist es daher entscheidend, den optimalen Kompromiss zwischen Geschwindigkeit und Qualität zu finden.

Für Daten-Clusterbildung im Allgemeinen sowie für Polygonnetz-Clusterbildung im Speziellen, wurden bisher primär k-means-ähnliche sowie hierarchische Ansätze verwendet. In vielen Fällen sind die Ergebnisse dieser Methoden jedoch unzureichend.

Die vorliegende Dissertation beschreibt neue Ansätze zur effizienten und hochqualitativen Clusterbildung von Polygonnetzen und allgemeinen Daten. Sie befasst sich sowohl mit dem algorithmischen als auch mit dem theoretischen Hintergrund von Clusterbildungsproblemen und stellt neue Ansätze zur Überwindung der Nachteile klassischer Verfahren vor.

Mit der Einführung moderner Grafik-Hardware, welche die massiv-parallele Verarbeitung von Daten ermöglicht, zeichnet sich ein neuer Trend zur Lösung von Clusterbildungsproblemen auf Grafikprozessoren (Graphics Processing Units, GPU) ab.

In dieser Arbeit wird der erste GPU-basierte Clusterbildungsansatz für Polygonnetze beschrieben, der die Konnektivität des Polygonnetzes berücksichtigt. Das Verfahren läuft dabei ausschließlich auf der GPU und verzichtet gleichzeitig auf jegliche globale Datenstruktur.

Basierend auf dem zuvor vorgestellten Verfahren, wird weiterhin ein GPU-basiertes Daten-Clusterbildungs-Framework vorgeschlagen. Dieses nutzt die räumliche Kohärenz im Kontext der Clusteroptimierung und des hierarchischen Mergings zur signifikanten Reduktion nötiger Vergleiche. Gleichzeitig umgeht der Ansatz das Problem der fehlenden Topologie-Information durch dynamisches Tracking von Nachbarschaften. Im Vergleich zu klassischen Verfahren erzielt das vorgestellte Verfahren Ergebnisse mit mindestens gleicher Qualität, skaliert dabei sehr gut und ist derzeit nur durch die verfügbare Menge an Grafikspeicher begrenzt.

Contents

List of Abbreviations

BB	Boundary-based
BIC	Bayesian Information Criterion
BL	Boundary Loop
CA	Cluster Array
CVD	Centroidal Voronoi Diagram
DE	Dual Edge
DG	Dual Graph
EMLO	Energy Minimization by Local Optimization
GPU	Graphics Processing Unit
HC	Hierarchical Clustering
HE	Half Edge
HFC	Hierarchical Face Clustering
ML	Multilevel
MWCVD	Multiplicatively Weighted Centroidal Voronoi Diagram
ODE	Optimal Dual Edge
PML	Parallel Multilevel
PQ	Priority Queue
VC	Variational Clustering

Chapter 1

Introduction

Due to technological advances, the amount as well as the dimensionality of data sets in general are steadily growing. This is especially the case for large polygonal surface meshes since existing 3D geometry acquisition systems can nowadays provide models with up to several million triangles. Thus, fast and high-quality data and polygonal mesh processing becomes more demanding.

Clustering, as a process of partitioning data elements with similar properties, is an essential task in many application areas [XW08]: computer science (web mining, information retrieval, mesh and image segmentation), engineering (machine learning, pattern recognition), medicine (genetics, pathology), economics (marketing, customer and stock analysis) and many other fields. To deal with such a huge and diverse heap of clustering problems efficient algorithms are required. At the same time the resulting clustering quality is of highest importance in almost all situations. Thus, identifying an optimal tradeoff between efficiency and quality is crucial in many clustering tasks.

For data clustering tasks in general as well as mesh clustering applications in particular, *k-means* like techniques or *hierarchical methods* are used most often. In both categories, a problem-dependent energy functional is present, which drives an optimization process (k-means) or the order of cluster merging (hierarchical methods). Nonetheless, these approaches are deficient in many respects thus a considerable amount of work is still required to improve them.

With the advent of general-purpose computing on Graphics Processing Units (GPU), which allows the usage of a highly parallel processing power, new tendencies emerged to solve clustering tasks on the GPU. The pioneering work of Hall and Hart [HH04] was the first attempt to accelerate the k-means algorithm. Although the approach applies also to surface polygonal meshes, it does not actually utilize the mesh connectivity in the clustering process. Thus, the demand for new GPU-based mesh clustering techniques is still an open problem.

In this dissertation new feasible solutions for efficient and high quality mesh and data clustering are introduced. We address both the algorithm and the theoretical part of many clustering problems. In all cases we are driven by the aim to fulfill the major general clustering goals.

1.1 Overall Goals

In this dissertation we investigate the clustering for two data types: polygonal surface meshes, which have geometry and connectivity (topology) information, and general data, which misses the topology and usually tends to be high-dimensional. We usually differentiate between these two data types and correspondingly apply different clustering algorithms.

However, from a general clustering point of view similar goals can be identified in both cases:

Automatic clustering: Given a clustering criterion, it is always desirable to have an automatic clustering. This implies no user interaction for driving the clustering process, defining or adjusting different clustering parameters, or any other user related decisions.

Efficient clustering: This generally implies that the algorithms must have low computational and memory requirements. Both can be achieved through:

- Better algorithm design.
- Efficient data structures.
- Parallelization of the clustering algorithms. Thus parallel architectures, e.g. multi-core CPU or GPU, can be used for acceleration.
- Intelligent energy functionals. Indisputably these are essential for efficient clustering, since the more cumbersome the energy functional, the more resources and computational effort it requires.

High quality results: It is always desirable to have the best possible clustering results regarding a given clustering criterion. If an energy functional is present the result must be as close as possible to the global optimum.

Generic clustering: This implies that the same clustering technique applies regardless of the problem or data specific details.

1.2 My Contributions

This dissertation describes a new generic clustering framework for efficient and high quality clustering. Most parts of this work have been published in different scientific articles [CK06], [CK08], [CKCL09], [CK11]. The major contributions of my work are:

The Energy Minimization by Local Optimization approach. This approach is proposed as a generalization of the Valette approach [VC04]. It is a formulation which is free from any global data structure and which can perform efficiently if a special energy functional formulation exists. In this context new energy functionals are introduced:

- The notion of *Multiplicatively Weighted Centroidal Voronoi Diagram* is generalized in the context of mesh coarsening. Different cluster's weights are suggested to capture the mesh features as good as possible.
- A new spherical mesh approximation energy functional has been developed, which can be represented in an incremental formulation.

The Multilevel (ML) clustering strategy. This strategy is proposed to solve inherent problems of the standard Variational and hierarchical algorithms, such as initialization dependency or greediness of clustering results. The algorithm neither uses any heuristics nor any a-priori user-specified parameters. It is generic and performs a complete mesh analysis, providing a complete set of solutions, and yielding results of at least the same clustering quality compared to standard techniques. The approach proved to be a powerful and reliable tool for clustering. In this context different new elements were developed:

- *A discrete data structure* for storing and reconstructing the multilevel construction.
- *Different variants of ML clustering* which allows the user to choose between faster execution or higher quality.

A GPU-based mesh clustering framework. It is the first GPU-based approach which employs mesh connectivity in the clustering process. It is solely based on GPU resources and allows for a considerable speedup. In this context many new elements were proposed:

- *A new mesh connectivity encoding* is proposed which allows performing all necessary mesh clustering tasks on the GPU.
- *The Boundary-based mesh clustering approach* is proposed. It is parallelizable and provides all necessary ingredients for a GPU-based implementation, without introducing any special energy functional requirements.
- *The Parallel Multilevel technique (PML)* which is an important strategy for an efficient GPU-based multilevel (hierarchical) clustering, allowing a further step towards parallelization.

A GPU-based data clustering framework. Like the mesh clustering framework, it is solely based on GPU resources. It generalizes the Multilevel clustering idea to data clustering, exploiting the spatial coherence present in the optimization and the cluster merging steps. Besides the fact that the approach generates results with at least the same clustering quality, it proved to be a better strategy at revealing the number of clusters present in the data set. In this context new elements were developed:

- *A dynamic cluster neighborhood tracking* to resolve the missing topological information. It allows required cluster merging and optimization.

- *The Local Neighbors k-means algorithm* as a counterpart to the classical k-means algorithm. The approach strongly incorporates the spatial coherence present in the optimization, and thus is faster and scales much better.

1.3 Outline

The remainder of the dissertation is organized as follows.

In **Chapter 2** we review most of the work done in the field of polygonal surface mesh and data clustering. We describe major algorithms and theoretical aspects in both cases. In the last section of the chapter we also describe the Graphics Processing Unit (GPU) and its application in the context of general-purpose computation.

In **Chapter 3** we describe a new clustering approach, i.e. Energy Minimization by Local Optimization, in the context of different application areas, such as mesh coarsening and mesh approximation. Many related improvements are also proposed in this chapter.

In **Chapter 4** we introduce the Multilevel mesh clustering approach. Here we describe all related algorithm and implementation details, together with many evaluation results.

In **Chapter 5** we describe a GPU-based solution for mesh clustering. We show how the mesh must be encoded to allow for mesh clustering on the GPU. Here we also propose new algorithms which can be implemented on the GPU. Additionally, we provide most of the GPU-specific implementation details and some non-trivial OpenGL-specific solutions.

In **Chapter 6** we demonstrate the GPU-based Multilevel solution for data clustering. We describe in detail all new elements and techniques. Here we extensively evaluate the proposed approaches. We also test the approach for revealing the number of clusters present in the data set.

Chapter 2

Mesh and Data Clustering Algorithms

In this dissertation the term *clustering* refers to the process of grouping the data elements in different groups, i.e. clusters, according to some criterion. A *cluster* is then a collection of data elements with similar properties, while the elements in different clusters are dissimilar from one another.

In this work we apply clustering to two data types: surface polygonal meshes and general data. In contrast to general data polygonal meshes possess additional connectivity information which must be taken into account during clustering. In both cases a clustering criterion is defined which drives the clustering process. The clustering criterion is usually problem dependent – there is no universal criterion that could be equally applied to any clustering problem. Nonetheless, the clustering techniques are usually similar, although applied to different data types. Thus, it is important to analyze advantages as well as the inherent problems which exist for this type of clustering approaches, with the aim to identify new and better solutions.

This chapter is intended to draw an overall picture of the most significant aspects of mesh and data clustering algorithms. Here, we give all necessary information for understanding of the later chapters. It must be recognized that there is a huge amount of work done in both fields. Thus, we focus on techniques that are closely related to our work.

In Section 2.1 we describe surface polygonal meshes and the state-of-the-art clustering algorithms which apply in this field. In Section 2.2, data clustering techniques are described. In both sections the k-means-like and hierarchical-like methods are considered in detail.

In the second part of this dissertation we propose approaches for performing clustering tasks on the Graphics Processing Unit (GPU). Thus, in Section 2.3 we describe in more detail the main GPU stages and the processing concepts in the context of general-purpose computations on graphics hardware. We also review the previous work related to GPU-based clustering. This gives the reader a first idea of GPU processing concepts and the underlying difficulties of using graphics hardware for clustering purposes.

2.1 Mesh Clustering

In this section we describe different aspects of surface polygonal mesh clustering (short: *mesh clustering*). First, in Section 2.1.1 we review some mesh and clustering terminology. In Sections 2.1.2 - 2.1.3 we describe in more detail Variational and hierarchical mesh clustering approaches.

2.1.1 Introduction

Polygonal meshes are the most often used representation for 3D models in computer graphics. This representation is a *de facto* standard geometric representation for interactive 3D graphics. Their popularity is mostly due to their flexibility to approximate any 3D shape with arbitrary accuracy. Additionally, polygonal meshes can be processed and rendered by current graphics hardware very efficiently.

According to [Sha08], a surface polygonal mesh M is defined as a tuple $\{V, E, F\}$ with:

- V a set of vertices $V = \{V_i = \mathbf{p}_i \mid \mathbf{p}_i \in \mathbb{R}^3,\ 1 \leq i \leq m\}$
- E a set of edges $E = \{e_{ij} = (\mathbf{p}_i, \mathbf{p}_j) \mid \mathbf{p}_i, \mathbf{p}_j \in V,\ i \neq j\}$
- F a set of faces $F = \{F_j\}$, which are represented by planar polygons.

The use of the term *surface mesh* is required in order to distinguish between 3D volumetric meshes and the actual 2D surface mesh that is embedded in 3D. Despite this, for the rest of this work we usually use only the term *mesh* for referring to surface polygonal meshes.

Although the faces F_j can be represented by arbitrary planar polygons, most often triangular faces are considered, i.e. $\{F_j = (\mathbf{p}_r, \mathbf{p}_s, \mathbf{p}_t) \mid \mathbf{p}_r, \mathbf{p}_s, \mathbf{p}_t \in V,\ r \neq s,\ r \neq t,\ s \neq t\}$, which are usually called *triangle meshes* or simply *triangulations*, see Figure 2.1. All the models used in this dissertation are represented by triangular meshes. Nonetheless, the proposed mesh clustering algorithms equally apply to any polygonal meshes.

An important topological characteristic of a mesh is whether or not it is a 2-manifold. This implies that for each mesh point the surface is locally homeomorphic to a disk, or half disk at the mesh boundary [BPK*08]. Note that, only in this case the local mesh neighborhood is well defined – a prerequisite for all mesh clustering algorithms proposed in this work. Thus, we always assume and require the mesh to be an orientable 2-manifold.

There are different mesh data structures which can be employed to store the geometry and the topological (connectivity) mesh information, see [BPK*08]. In general this depends on the application requirements, i.e. fast mesh queries or traversal, which might be necessary during different mesh processing tasks. The *winged-edge* [Bau72], *half-edge* [Män88] and *directed-edge* [CKS98] data structures are most often employed. In our work we use a variant of the half-edge data structure, as depicted in Figure 2.1. For each halfedge the following references are stored: next halfedge (`next_halfedge`); previous halfedge (`prev_halfedge`); the halfedge in the opposite direction (`twin`); its adjacent face

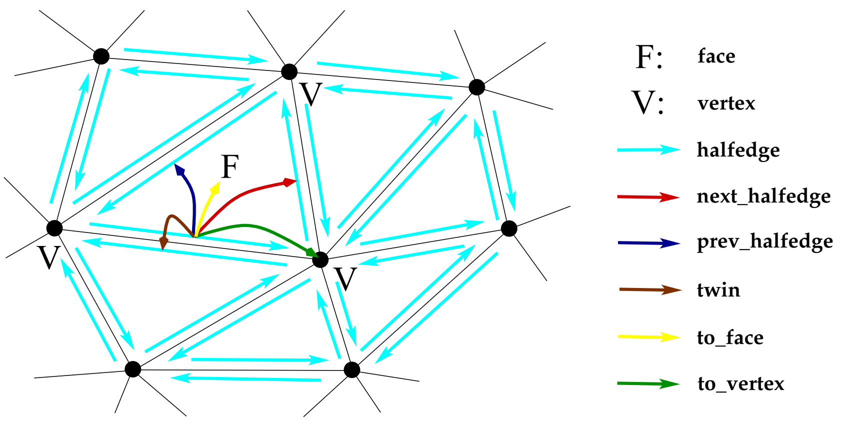


Figure 2.1: Example of stored references for each halfedge in a half-edge data structure.

(`to_face`); the vertex it points to (`to_vertex`). Additionally, for each face a reference to one of its halfedges and for each vertex a reference to one of its outgoing halfedges is stored. This allows an efficient retrieval of any required mesh information during the clustering process.

Now, given a clustering criterion, *clustering* a mesh means grouping the mesh elements, such as faces or vertices or edges, into clusters according to a given similarity measure. Generally, an energy functional is given (sometimes named error or cost functional), which calculates the costs for assigning an element to a specific cluster. This energy must be minimized when clustering the mesh elements.

Mesh clustering is used in many areas of computer graphics. Later on we will describe some of the applications. For an overview in the context of mesh segmentation see [Sha04], [Sha06], [AKM*06], [APP*07], [Sha08], [CGF09].

Note that, due to a variety of application areas, different terminology is sometimes used regarding the clustering process. It is sometimes referred to as partitioning or tessellation and the clusters as regions. In the field of mesh segmentation these two notions are sometimes interleaved and cannot be well separated. Still, it must be recognized that segmenting a model can be done without employing a clustering process.

Due to a huge amount of different mesh clustering problems, there are plenty of algorithms to accomplish this. However, there are two classes of algorithms that prevail and are *de facto* standard for performing mesh clustering. These are Variational and hierarchical methods. In both categories, an energy functional is present, which drives the optimization (Variational) or the order of the cluster merging (hierarchical methods). In Sections 2.1.2-2.1.3 we describe these algorithms in more detail. At the end of each section we also provide examples of their applicability in different areas of computer graphics.

2.1.2 Variational Mesh Clustering

Variational mesh clustering has been proposed by Cohen-Steiner et al. [CSAD04] as an extension of Lloyd's algorithm [Llo82], sometimes referred to as *k-means algorithm* (see

Section 2.2.1), for planar polygonal mesh approximation. Variational Clustering (VC) is *de facto* the common and most often employed mesh clustering approach. It casts the mesh clustering problem into a variational partitioning problem where the mesh's faces are clustered such that the total approximation error is minimized.

The Algorithm.

Suppose that an input polygonal mesh M with m faces F_j is provided. Clustering M into k non-overlapping clusters C_i means that each cluster C_i consists of a union of n_i mesh faces F_j^i.

In general, each cluster C_i can be summarily represented by a *cluster proxy*, or as originally proposed by a *shape proxy*, according to the Definition 2.1.

Definition 2.1. *A cluster (shape)* ***proxy*** *P_i is a local best representative of the cluster C_i w.r.t. a given energy functional.*

As an example, in [CSAD04] for planar mesh approximation the shape proxy $P_i = (\overline{\mathbf{C_i}}, \overline{\mathbf{N_i}})$ is used, where $\overline{\mathbf{C_i}}$ is the cluster centroid and $\overline{\mathbf{N_i}}$ is the normalized average over all face normals of the cluster C_i. In this case the shape proxy is a local representative of the cluster C_i which best approximates a given cluster geometry. Accordingly, the proxy set $P = \{P_i\}$ approximates the whole mesh geometry.

Now, suppose that a proxy-based energy functional $E(P)$ is provided accroding to the Definition 2.2.

Definition 2.2. *A proxy-based*[1] *energy functional $E(P)$ is an energy functional defined as*

$$E(P) = \sum_{i=0}^{k-1} E(C_i, P_i) = \sum_{i=0}^{k-1} \sum_{F_j \in C_i} E(F_j, P_i). \tag{2.1}$$

where $E(C_i, P_i)$ is the error of the cluster C_i for the corresponding proxy P_i. $E(F_j, P_i)$ is the error given by assigning the face F_j to the cluster C_i with proxy P_i.

For a given energy functional $E(P)$ and a given number of clusters k the algorithm seeks to find a clustering of the input mesh into k clusters C_i and associated set $\{P_i\}_{i\in\{0,\dots,k-1\}}$ of proxies that minimizes the total energy $E(P)$. The Variational clustering performs according to the Algorithm 2.1.

Algorithm 2.1. *(The Variational Clustering Algorithm)*

```
1 Initialization step.
2 Repeat until convergence {
3   Seeding step.
4   Partitioning step.
5   Fitting step.
6 }
```

[1]Note that, in the case of a a proxy-based energy functional we always assume that the proxies are explicitly computed before any energy computation.

In Algorithm 2.1 each step performs as follows:

Initialization step: Here the starting seeds with associated starting proxies are identified. Usually a random initialization is used to choose the starting seeds. For defining the starting proxies the seed's local information is used, e.g. for planar fitting [CSAD04] the normal of the seed face is used.

Seeding step: For each cluster C_i and its associated proxy P_i, identify a face $F_j \in C_i$ with the smallest energy $E(F_j, P_i)$ as a new cluster seed. This is achieved by visiting all faces in the cluster. Sometimes, as a second requirement, the seeds must be as close as possible to the cluster center.

Partitioning step: For each seed face F_i^{seed} insert its three adjacent faces F_j into a global priority queue (PQ) with a priority equal to their respective energy $E(F_j, P_i)$. Thus, a face F_j with m edges can appear up to m times in the PQ with different cluster labeling and priority. After this, an energy-minimizing cluster growing Algorithm 2.2 is used to cluster the mesh elements into k clusters. Here each face is assigned to the best fitting proxy. As a result, a new mesh clustering is obtained.

Fitting step: After the partitioning step a new set of proxies $\{P_i^{new}\}$ is computed from the obtained clustering. This is then used for new seeding and partitioning steps.

Algorithm 2.2. *(Energy-minimizing cluster growing)*

```
while the $PQ$ is not empty {
  pop a face $F_j$ with the smallest cost $E(F_j, P_i)$
  if $F_j$ not assigned to any cluster {
    assign $F_j$ to cluster $C_i$
    push unlabeled incident faces of $F_j$ (up to two) into the $PQ$ with label $i$
  }
}
```

For a user specified number of clusters k, the Variational clustering algorithm provides a k-partitioning of the input mesh with k non-overlapping and connected clusters. An approximative exemplification of the steps of this clustering algorithm is presented in Figure 5.2 on page 94.

In [CSAD04] several enhancements were proposed to improve the algorithm:

1. **Incremental seed insertion and deletion:** The user has not only the possibility to define the desired number of clusters but also to interactively insert or delete a proxy. The insertion is done by finding a cluster with highest energy and within it picking a face with worst distortion as a new seed. This adds a new cluster in the most needed part of the object. The deletion is done by computing for each pair (or random pairs) of clusters the merging energy. The pair with the smallest energy is then replaced with a single one, i.e. the two clusters are merged, and as a result a cluster is deleted.

2. **Cluster teleportation:** In many situation the algorithm may find itself stuck in a local minimum, because the clusters cannot move easily to different parts of the surface to find better positions. To improve this situation, a cluster deletion followed by a cluster insertion is applied at regular intervals. The idea is to teleport a cluster to the most needed mesh location. In practice a heuristic is used to test if the energy added by the deletion is smaller than half of the energy of the worst cluster; if it is not, no teleportation is applied.

3. **Progressive initialization:** The idea is to add one region at a time, perform partitioning and then choose the face with maximum energy as a new seed. This approach works well for non-smooth objects but fails to provide good initialization if a lot of noise is present.

Application areas.

The Variational clustering is used in a wide range of applications and is in fact "a first choice" for performing many mesh clustering tasks. Depending on a specific problem the algorithm (Algorithm 2.1) may undergo some changes, or specific constraints may be imposed on the clustering process to support different clustering needs. Here we present some of these applications with the aim of spotting major technical aspects as well as advantages or disadvantages of the algorithm for further analysis.

- As originally proposed in [CSAD04], the algorithm was employed for planar approximation of the mesh for subsequent remeshing. In addition to the enhancements described above such as incremental seed insertion and deletion of clusters or cluster teleportation, the authors also suggested that, in some situations, it is valuable to have an interactive tool that allows the user to artificially scale up or down the area of specific regions. In this case, the algorithm weights different regions differently, thus preserving different surface details better.

- In [WK05] the approach was extended to support other proxy shapes such as spheres, cylinders and rolling-balls blend patches. The motivation of this choice is the fact that most CAD objects consist of such patches. Thus, compared to the planar approximation, a more compact approximation with fewer number of proxies can be achieved.

  Compared to the standard algorithm, in this approach the major changes appear in the fitting phase. There is no longer only a single proxy to be computed in the fitting step, but four of them, i.e. best fitting plane, sphere, cylinder and blend patch. In this case, only one out of four, the one which gives minimal fitting energy $E(C_i, P_i)$, must be chosen as shape proxy and assigned to a given cluster.

  However, due to complexity and thus slower proxy fitting, a *progressive clustering* was proposed. In the first iterations only planes are used for fitting until the change in the energy is very small. Then sphere and cylinders are also allowed to be fitted. Finally the rolling-ball blend patches are permitted until the final stable clustering is reached.

- In [SS05] the approach was applied to fit ellipsoidal regions for compact geometric representation. The approach allows surface as well as volume oriented fitting. The algorithm obeys the same strategy as in [CSAD04] and terminates when the desired number of iterations has been performed.

- In [JKS05] the algorithm was adapted for identifying quasi-developable surfaces. The authors proposed an automatic procedure to increase or decrease the number of clusters based on a fitting error. In this case, the clusters are grown until a bounding fitting error E_{max} is achieved, using standard Variational clustering. After this, new clusters are added for "large" holes, or clusters are deleted if a developability condition allows merging of two clusters.

- In [YLW06] the method was extended to extract not only planes or special types of quadrics (spheres and circular cylinders), but also general quadric surfaces. Using a progressive initialization [CSAD04] (start with $k = 1$ and then progressively add a cluster) the algorithm terminates when a pre-specified energy threshold is met and the iterations have converged. Checking for redundant proxies was also employed by considering merging each pair of adjacent clusters. Furthermore, a method for boundary smoothing was presented and used to improve the final result.

- In [WZS*06] the method is employed to approximate a triangle mesh by a bounding set of spheres which have a minimal summed volume outside the object. A mesh is discretized into inner points by voxelizing the object and surface points. The clustering is applied to both types of points. The region teleportation is triggered when insufficient error improvements occur.

- In [OS08], in the context of image-based surface compression, the approach was employed to partition the mesh into a set of elevation maps.

- The approach can be used to construct an approximative Centroidal Voronoi Diagram on polygonal meshes. However, as we will point out in Section 3.1, the Valette approach [VC04] is more efficient for this purpose.

Discussions.

The Variational method does provide an optimal clustering, in the sense that at least a local minimum is reached, but for an a-priori user-specified number of clusters k. In general, the determination of an appropriate number of clusters involves manual user intervention.

Additionally, the choice of initial seeds, i.e. starting positions and starting representatives, affects the convergence and the final result. If a random initialization[2] is used then two consecutive executions of the algorithm produce, in general, different results, for an

[2]A usual reasoning behind this choice is that it is fast and it works "well" in practice providing good results.

example see Section 3.1.4. This produces suboptimal clustering results and might hinder fixation of the number of clusters or other clustering parameters.

Other initialization heuristics can also be applied such as the multiple-run approach where multiple random initializations are applied and the solution with the smallest error is reported – a computationally very expensive approach. The farthest point initialization [KJKZ94] can also be used to choose the starting seeds. However, for many applications this uniform sampling does not pay off, because it does not support the underlying energy functional.

Proposed heuristics in [CSAD04] such as incremental insertion and deletion of regions and region teleportation proved to be efficient in many situations. However, it is still an open question when and where to apply them, and the operations are computationally expensive.

Thus, having an appropriate initialization technique for this class of iterative approaches is still a major problem that needs to be addressed.

Last but not least note the sequential nature of this clustering approach. In the partitioning step a PQ is used to identify the smallest cost element, and for cluster growing only one face can be assigned to the best fitting proxy, see Algorithm 2.2. This means that the algorithm can not be efficiently implemented on parallel hardware, e.g. Graphics Processing Units (GPUs).

2.1.3 Hierarchical Mesh Clustering

Vertex hierarchies, which are common in mesh simplification algorithms where edge contraction operations are applied [XV96] [Hop97] [LE97] [Gar99], can be considered an early application of this type of clustering. In [GWH01] the hierarchical face clustering is proposed for planar approximation of polygonal meshes.

The Algorithm.

A hierarchical clustering always produces a binary tree. The algorithm works as described below; see [Sha08].

Algorithm 2.3. *(The Hierarchical Clustering Algorithm)*

```
Initialize a priority queue $PQ$ of pairs $(u, v)$
Until $PQ$ is empty {
  Get next pair $(u, v)$ from $PQ$
  if $(u, v)$ can be merged {
    Merge $(u, v)$ into $w$
    Insert all valid pairs of $w$ to $PQ$
  }
}
```

If a vertex hierarchy is built then the original mesh vertices are the nodes of the hierarchy. Each mesh edge provides a pair of vertices (V_i, V_j) that can be clustered together. According to Algorithm 2.3, all these pairs, the number of which equals exactly the number of edges in the mesh, are inserted into a Priority Queue (PQ) with a problem-specific defined priority[3]. In each step two vertices are contracted together and a new node is created in the tree. For this newly created node, new vertex pairs are created for all adjacent vertices and inserted in the PQ, see [Hop97].

In [GWH01] face hierarchies are proposed, i.e Hierarchical Face Clustering (HFC). Here, to follow the analogy of vertex hierarchies, the algorithm starts by creating a Dual Graph (DG) of the mesh. Each mesh face is assigned to a node in the DG and a dual edge (DE) between two nodes is created if the corresponding mesh faces are adjacent. At the beginning each mesh face F_j is assigned to a specific cluster C_i. In the Priority Queue all created DEs are sorted according to their contraction cost (merging energy). At each step, a DE with highest priority is popped from the PQ and collapsed. Collapsing a DE means merging two clusters into one representative cluster, for an example see Figure 5.4 on page 98. After collapse, the PQ is updated for all DEs incident to the newly created node. For more details see [GWH01]. As a result one obtains a hierarchy (a binary tree) of face clusters.

Application areas.

Hierarchical mesh clustering is of major importance and is employed in applications which seek to identify a multiresolution partitioning of the original mesh for efficient spatial queries or mesh representation/approximation:

- As already pointed out, vertex hierarchies are used for view-dependent reconstruction of surface approximation [XV96] [Hop97] [LE97] [Gar99].
- In [GWH01], the HFC is proposed for planar approximation of polygonal meshes. Some potential application areas of their approach where also suggested, such as distance and intersection queries, collision detection, surface simplification, or multiresolution radiosity.
- In [SSGH01], the HFC is employed to partition the mesh into charts that have disk-like topology. In post-processing the chart boundaries are straightened by computing a shortest path over mesh edges.
- In [She01], the HFC is applied for CAD model simplification for remeshing purposes. Different geometric and topological costs are considered when merging the clusters.
- In [GG04], the hierarchical clustering is used to segment the mesh points or an arbitrary point set into a set of components which possess one or more slippable motions, i.e. rigid motions that slide the transformed version against stationary one without forming any gaps.

[3]The priority is usually modeled in the energy functional.

- In [AFS06], Attene et al. employed the HFC technique for fitting a family of primitives such as planes, spheres and cylinders, which can be used for reverse engineering or mesh denoising and fairing. In that case the cost of collapsing a DE is the minimum of the approximation errors computed against all primitives.

Discussions.

The main advantage of the hierarchical approach is that it does not require any additional parameters or any intervention from the user. This is in contrast to Variational clustering, which requires the initial number and positions of the starting seeds. The hierarchical clustering starts with each element, i.e. vertex or face, as a separate cluster and then decreases the number of clusters in each step applying the merging operation.

Despite the simplicity and wide range of applications, hierarchical clustering is yet a *greedy* approach, i.e. an assigned element can no further be reassigned to other clusters although that may result in a more appropriate configuration. This drawback limits the applicability of hierarchical clustering in many situations, e.g. generating Centroidal Voronoi Diagram (CVD) for mesh coarsening. Applying a greedy hierarchical approach on base of a CVD does not yield a valid solution, i.e. the result is not a CVD.

2.2 Data Clustering

Clustering, as a process of partitioning data elements with similar properties, is an essential task in many application areas [XW08]: computer science (web mining, information retrieval, image segmentation), engineering (machine learning, pattern recognition), medicine (genetics, pathology), economics (marketing, customer and stock analysis) and many other fields.

Correspondingly, a lot of work has been done in these fields. This concerns not only on the data types, which could be discrete or continuous. It also concerns the diversity of clustering problems, which are usually separated into hard (where a data point belongs to one and only one cluster) and soft (fuzzy) clustering (where a data point might belong to two or more clusters with some probabilities). Different clustering algorithms such as hierarchical, partitional, neural network-based or kernel-based clustering, and others, can be applied. Not to mention the diversity of proximity (similarity or dissimilarity) measures that are defined between data elements or between clusters. For a more detailed overview on this topic we refer the reader to [XW08] and [GMW07], which are the most up-to-date and comprehensive references for data clustering aspects.

For the rest of this work we assume that a data set $D = \{\mathbf{Q}_1, \mathbf{Q}_2, ..., \mathbf{Q}_m\}$ is given with m points $\mathbf{Q}_j = \{q_{j1}, q_{j2}, ..., q_{jd}\} \in \mathbb{R}^d$, where d is the dimensionality of data set.

Our goal is to cluster this data set according to some predefined clustering criteria. Despite the diversity of clustering algorithms, k-means approach and hierarchical methods are most often employed. In Section 2.2.1 we describe the k-means approach in more details. In Section 2.2.2 hierarchical (agglomerative and divisive) methods are described.

In all cases we try to identify the advantages and the drawbacks of these approaches.

2.2.1 K-Means Clustering

The *k-means algorithm* [For65], [Mac67], sometimes referred as *Lloyd's algorithm* [Llo82], is one of the simplest and most often employed clustering algorithm.

The algorithm.

For a user specified number of clusters k, the k-means approach seeks an optimal partitioning of the data elements. The main steps of k-means are performed according to the Algorithm 2.4.

Algorithm 2.4. *(The k-means Algorithm)*

```
1 Select $k$ points as initial cluster centroids
2 Repeat until convergence {
3   Assign each element to the nearest cluster
4   Recalculate the centroid of each cluster
5 }
```

The k-means algorithm usually starts with k randomly peeked points and assign them as starting cluster's centroid $\overline{\mathbf{C}_i}$. After that all data points are assigned to the nearest cluster C_l, i.e.

$$\mathbf{Q}_j \in C_l, \text{ if } \|\mathbf{Q}_j - \overline{\mathbf{C}_l}\| < \|\mathbf{Q}_j - \overline{\mathbf{C}_i}\|$$

for $j = 1, ..., m;\ i \neq l$ and $i = 1, ..., k$.

This is followed by recomputation of the centroid for each cluster. The cluster's centroid is computed as $\overline{\mathbf{C}_i} = \sum_{Q_j \in C_i} \mathbf{Q}_j / n_i$, where n_i is the total number of points in cluster C_i. These new centers are used to perform a new partitioning afterwards.

The two steps: partitioning and centroid update in the Algorithm 2.4 are performed until convergence, i.e. the newly computed centroids are the same or the points are assigned to the same clusters, or until a user specified maximum number of iterations.

It can be proven that such an algorithm aims at minimizing the energy functional

$$E = \sum_{i=0}^{k-1} \sum_{Q_j \in C_i} \|\mathbf{Q}_j - \overline{\mathbf{C}_i}\|^2.$$

which is the within-cluster variance, i.e. the squared distance between cluster's centroid and its assigned data points. For such a functional the algorithm always converges, because each step reduces the energy E.

Note that, when assigning the data elements according to the nearest-neighbor rule the input space is divided into Voronoi regions. Thus, there is a strong link between the Centroidal Voronoi Diagram and the k-means clustering [DFG99]. Both minimize the within-cluster variance and the cluster's centroid is exactly the centroid of its associated Voronoi region.

Discussions.

The k-means algorithm works very well in practice. However, although it converges, there is no guarantee that the global minimum will be reached. Due to random initialization, the algorithm usually converges only to a local minimum. Thus the result is strongly dependent on the initialization.

The solution might be to improve the initialization. The multiple random run approach or the farthest point initialization [KJKZ94] can be considered to reduce the effect of initialization. However, these heuristics are computationally expensive and there is no guarantee that the final solution will be better. Thus, in many situation different alternatives to k-means are considered that find better clusterings, see [HE02] for an overview and comparison.

Another problem of the k-means algorithm is that the user needs to define the number of clusters k present in the data set. Like for cluster initialization, there are no efficient and universal methods to define k.

In image segmentation a behavior known as *elbowing effect* is usually used to decide on the number of clusters. As long as the data is not naturally clustered there is only a small increase in the energy E; further decrease in k will then effect much larger increase in E, see for an example [DGJW06].

A good solution for this problem is the X-means approach of Pelleg and Moore [PM00]. They extended the k-means algorithm to efficiently estimate the number of clusters. The algorithm uses the Bayesian Information Criterion (BIC) to decide if a cluster split into two subclusters models a real structure or not. For a given range $[k_{min}, k_{max}]$ of clusters the steps of X-means approach are summarized in Algorithm 2.5.

Algorithm 2.5. *(The X-means Algorithm)*

```
Identify $k_{min}$ starting seeds
Apply k−means.
Repeat until $k < k_{max}$ or no split applied {
  Virtually subdivide each cluster into 2 subclusters
  Use the $BIC$ score to decide if the splits must be applied or not.
}
```

The algorithm starts with a configuration which is obtained using the standard k-means algorithm, see Algorithm 2.4. In each following step each cluster is subdivided into two subclusters using the standard k-means algorithm, i.e. 2-means. At this point a model selection test is performed on all pairs of subclusters, using the BIC score. Depending on the result of the test, a split of a cluster into two subclusters might be accepted or not. If splitting is not allowed, the cluster is left intact. The process is repeated until none of the splits improve the BIC score, i.e. none of the splits are allowed.

For a d dimensional data set D with m number of data elements and a family of different clustering solutions M_k for different k number of clusters, the Bayesian Information

Criterion (BIC) $BIC(M_k)$ or Schwarz criterion is computed as [Sch78], [PM00]:

$$BIC(M_k) = \hat{l_k}(D) - \frac{p_k}{2} log\ m.$$

where $\hat{l_k}(D)$ is the log-likelihood of data D according to the M_k solution, which describes how well a given model fits the data; $p_k = k + dk$ is the number of parameters in M_k. Finally a clustering solution with maximum value of BIC is selected. In general, the BIC imposes a tradeoff between model quality (first term) and model complexity (second term), thus seeking accurate and simple models. For more details on the computation of the BIC see [PM00].

2.2.2 Bottom-up vs. Top-down Hierarchical Clustering

In contrast to meshes where usually only agglomerative (bottom-up) hierarchical methods are used, in data clustering divisive (top-down) hierarchical methods are also used. For some problems divisive approaches appear to lead to better clustering results [SKK00], [SB04].

The algorithms.

The main steps of the agglomerative hierarchical method are summarized in Algorithm 2.6.

Algorithm 2.6. *(The Agglomerative Hierarchical Algorithm)*

```
1 Start with $m$ points as initial $k$ clusters
2 Repeat until $k == 1$ {
3   Search 2 clusters with minimal distance $D(C_i, C_j)$
4   Merge $C_i$ and $C_j$ into a new cluster $C_{ij}$
5 }
```

The agglomerative hierarchical method starts with m clusters. Each of them contains exactly one data point. In each step, two clusters with the smallest distance between them is found and the clusters are combined into one new cluster. There exists a large number of distance definitions for cluster merging. For an overview we refer the reader to [XW08] and [GMW07].

In contrast to agglomerative techniques the divisive approaches proceed in the opposite way. At the beginning, the entire data set is contained in one cluster. In each subsequent step a cluster is divided into two clusters according to some rule.

As an example, the bisecting k-means [SKK00], [SB04] works this way. The steps of this approach are summarized in Algorithm 2.7.

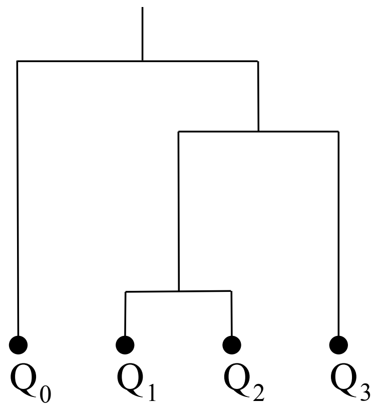


Figure 2.2: A simple example of a possible dendrogram for four points.

Algorithm 2.7. *(The Bisecting k-means Algorithm)*

```
Define the whole data set is a single cluster
Loop until $k$ clusters are found {
  Pick a cluster to split
  Apply bisecting step.
}
```

In general the largest cluster is chosen to be split. In the bisecting step a two clusters k-means, i.e. 2-means, algorithm is applied L times and a split with smallest energy is selected.

The results of hierarchical clustering are usually depicted as a binary tree or dendrogram, which provides very informative description of existing data structures. Figure 2.2 depicts such an example. The height of a dendrogram usually expresses the different distance measures, for more examples see [XW08] and [GMW07].

Discussions.

Note that, the bisecting k-means algorithm is computationally more expensive than the agglomerative hierarchical clustering. Although it is believed to be less affected by the containment problem. However, it must be recognized that the result of both agglomerative or divisive hierarchical methods will suffer from any erroneous decisions made at some step and these will be propagated in the successive steps. This is the major drawback of this kind of approaches.

This can be improved by applying an optimization step after merging or splitting. This is the same idea as the Multilevel clustering approach [CK08], [CK11]. In Chapter 6 we test this idea for data clustering. For divisive hierarchical clustering the X-means algorithm [PM00] can be employed[4] if starting with $k_{min} = 1$.

[4]Note that, in the original paper [PM00] when subdividing a cluster into two subclusters only the

It must be pointed out that the X-means and the divisive approaches use two cluster k-means for the bisecting step. Thus, for different runs, fluctuations might be present in the final result.

Although most clustering algorithms still comply with a sequential order of processing, attempts for parallel data clustering have been made, for an example and an overview see [Ols95]. In the field of image segmentation a parallel region growing paradigm [WLR88] was introduced. The growing is performed by identifying all possible merge partners and merging regions with mutual choices, similar to the parallel Multilevel mesh clustering [CKCL09], [CK11].

2.3 GPU-based Processing

The *Graphics Processing Unit (GPU)* is a highly parallel hardware for accelerating graphics applications. Modern GPUs are capable of rasterizing and processing billions of vertices and fragments per second. Although originally designed to efficiently deal with computer graphics tasks, in the last decade a new trend has emerged to exploit this hardware for general-purpose computing. This is mainly driven both by rapid increase of the GPU computational power and recent improvements in its programmability.

In this section we describe general GPU-related aspects in the context of mesh and data clustering. Section 2.3.1 gives a brief introduction to the modern GPU, together with the processing, computing and programming concepts. Section 2.3.2 describes the previous work of using the GPU for clustering tasks.

2.3.1 GPU Computing

The modern graphics pipeline consists of three main stages: the vertex, geometry and fragment stage. Figure 2.3 depicts these stages and the workflow in the pipeline. With the advent of programmable pipeline, which mostly replaced the fixed-function pipeline, the user can define a *program* specific to each stage.

The three stages operate as follows:

- In the vertex stage a *vertex program* operates on incoming vertices, manipulating them according to the application's objectives. Traditionally this includes vertex coordinate transformation and lighting calculations.
- In the geometry stage assembled primitives can be modified, extended or deleted by a *geometry program*. Here, new extra rendering primitives can be emitted, or the geometry generated procedurally by adding or removing vertices.

points in the original cluster are considered. Thus yielding, as for bisecting k-means, a nested hierarchy. However, in the publicly available implementation of the X-means algorithm [PM] this restriction is relaxed and the points can be assigned also to neighboring clusters.

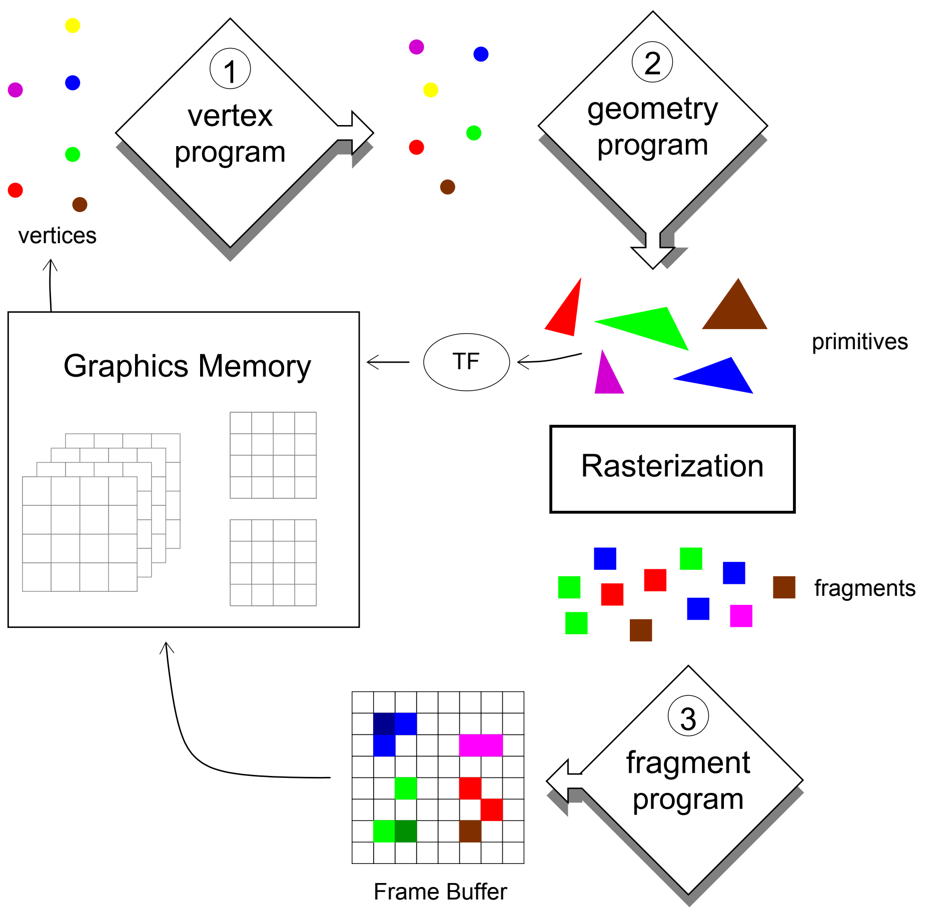


Figure 2.3: The graphics pipeline. TF: transform feedback.

- In the fragment stage a *fragment program* is executed for each rasterized fragment. Here, the color of each output pixel is calculated and written to the frame buffer or the render target, or to multiple render targets[5] (MRT).

Each of these stages is capable of *memory gathering*, i.e. the ability to fetch data from different positions in texture memory. However, only the vertex stage is capable of performing *scattering*, i.e. alter the output position of an element. This can be achieved by drawing the input vertices as points with correspondingly selected output locations. This procedure, however, may lead to memory and rasterization coherence problems, which can ultimately affect the performance [OLG*07]. In contrast, the output address of a fragment is predefined even before the fragment is processed. This limitation needs to be taken into account because it dictates the processing workflow of the fragment stage.

Earlier hardware architectures implemented each of these programmable stages in special dedicated hardware units called *shader units*, each optimized for its task. However, with

[5]The Multiple Render Targets (MRT) mechanism allows to simultaneously write at a given position in all render targets.

the introduction of NVIDIA's GeForce 8 series and ATI Radeon HD 2000 series GPUs the *Unified Shader Architecture* has been proposed. Such hardware is composed of a bank of computing units (shader units), each capable of performing any of the pipeline steps. Thus, the GPU can dynamically schedule the computing units for better load balance, thereby significantly increasing the GPU throughput.

The programmable units of the GPU follow a single program multiple data (SPMD) programing model, i.e. all elements are processed in parallel using the same program. Each element is processed independently from each other and they cannot communicate with each other. Currently, there are two main graphics application programming interfaces (APIs): OpenGL and DirectX. For programing the shader units on the graphics card the so-called *shading languages* emerged: *Cg* (C for Graphics) [MGAK03], *GLSL* (OpenGL Shading Language) [Ros06] and *HLSL* (High Level Shader Language). For our GPU-based implementation we use OpenGL with GLSL.

General-Purpose Computing on the GPU

Utilizing the GPU for non-graphics applications has evolved as a special field of *General-Purpose Computation on the GPU (GPGPU)*[6]. This was mainly driven by a steady increase in the computational power of the GPU compared to the CPU. As a result, many applications or simulations can be speeded up by this highly parallel streaming processor [OLG*07], [OHL*08], [SDK05].

Programing the GPU for general-purpose applications can be achieved in two ways [OHL*08]:

1. Using a graphics API, i.e. shading languages Cg, GLSL or HLSL. Here the program is structured in terms of the general graphics pipeline stages, see Figure 2.3. Usually, the programmer redefines a non-graphics problem using graphics terminology and data structures such as textures, vertices, fragments, buffers, etc. There are many GPGPU techniques which efficiently map complex applications to the GPU. We refer the reader to the state-of-the-art report of Owens et al. [OLG*07] and [OHL*08], where many of these techniques are describe in detail.

2. Using non-graphics interfaces to the hardware. Programming a non-graphics general-purpose application using a graphics API is in many cases cumbersome and the programmable units are only accessible at an intermediate step in the pipeline. For a common programmer a familiar high level language which gives direct access to the GPU programming units is more desirable. NVIDIA's CUDA[7] (Compute Uniform Device Architecture) programing language is a good example in this case. It allows a flexible and better suited environment for general parallel processing. However, this flexibility comes at the cost that the user needs to understand the low-level details of the hardware to achieve good performance. The OpenCL[8] (Open Computing

[6]`http://www.gpgpu.org/`
[7]`http://www.nvidia.com/cuda`
[8]`http://www.khronos.org/opencl`

Language) language, which is an open standard like OpenGL and supports GPUs of multiple vendors, is about to become an alternative to CUDA in the near future.

There are many areas where GPUs have been used for general-purpose computing and this is an ever growing field. Examples include physically-based simulation (game physics, biophysics, climate research), signal and image processing (image segmentation, computer vision, medical imaging), computational finance and many others. We refer the reader to [OLG*07] and [OHL*08] for a more comprehensive overview of these applications. The GPGPU web page[6] also keeps track for most of the ongoing developments in this field. In the next section we review the GPU-based clustering approaches in details.

2.3.2 GPU-based Clustering

One of the most prominent works in GPU-based acceleration of iterative (Lloyd's) clustering was the work of Hall and Hart [HH04]. They proposed a GPU-CPU solution, where the pairwise distance evaluations are done on the GPU and the centroid update on the CPU. For each cluster the model data (centroid) is loaded into shader constants and all pairwise (point-to-cluster) distances are computed. The result is written as a depth value of the fragment and the cluster ID as fragment color value. The depth test ensures that for each point the cluster ID with the smallest distance is kept. After all clusters are processed, the color buffer is read back to the CPU and the cluster's information is updated. Speedup factors of between 1.5 and 3 were reported comparing to the CPU implementation. A similar GPU-CPU solution was proposed also in [TK04].

The work in [CTZ06] proposed more efficient GPU-based solutions to k-means. However, due to overwhelming computation cost, the reduced communication cost of the GPU-based solution did not pay off compared to the GPU-CPU solution.

In [SDT08] another similar solution is proposed. For each cluster an individual distance texture is defined to keep point-to-cluster distances. This, generally, limits the maximum number of clusters that can be handled. Still, for specific configurations speed up factors of 4 to 12 have been reported.

In [ZZ06] a GPU-based acceleration of hierarchical clustering for gene expression profiles was proposed. First, the similarity distance matrix is calculated, i.e. the distances between all pairs of genes. This is achieved, as for k-means applications [HH04] [CTZ06] [SDT08], by rendering a quad that covers the distance matrix texture. For each generated fragment the cluster-to-cluster distance is computed. To identify a minimum distance for two clusters to be collapsed, a reduction operation [KW03] is used. Depending on the input data set, speed up factors of $2-4$ have been reported.

In the recent past, CUDA implemented k-means [FRCC08] [HtLlDt*09] [ZG09] and hierarchical clustering [SDTW09] methods have been proposed. For specific problems, speedup factors of 13 for k-means and $30-65$ for hierarchical clustering have been reported.

However, as we will point out in Section 6.2, the standard k-means approach (Section 2.2.1), although parallelizable, is a brute-force algorithm. All the GPU-based clustering approaches considered so far, including the hierarchical approach, simply try to map these brute-force algorithms on the GPU. In this case, no attempt is actually done to reformulate the algorithms so that higher parallelism can be achieved.

In contrast, the GPU-based Local Neighbors k-means approach proposed by Chiosa and Kolb [CK11], which reformulated the k-means algorithm, achieves better parallelism by taking into account the spatial coherence present during optimization as during cluster merging. Thus this algorithm can deliver better performance on the GPU, see Section 6.2.

Although the solution proposed in [HH04] for Variational clustering has been shown to work for polygonal meshes, no mesh connectivity information is actually used in the clustering process. The mesh is viewed as a triangle soup and clustering is applied on element basis. This, however, can lead to unsatisfactory clustering results, as we show in Figure 5.1 on page 92.

Two major problems can be identified that impede the use of the GPU for mesh clustering, which explains why there is so little work done in this field. First, the lack of parallel algorithms – most of existing mesh clustering algorithms are sequential. Second, the lack of a Half Edge data structure [Män88] on the GPU – to our knowledge there is currently no such data structure on the GPU.

The only existing GPU-based mesh clustering framework which employs the mesh connectivity in the clustering process is presented by Chiosa and Kolb in [CKCL09], [CK11]. They addressed iterative as well as hierarchical (multilevel) mesh clustering on the GPU. Using a special mesh connectivity encoding they have shown that both approaches can be efficiently implemented on the GPU. This is described in detail in Chapter 5.

Chapter 3

Energy Minimization by Local Optimization

As described in Section 2.1.2, the Variational clustering (VC) [CSAD04] is the *de facto* standard approach for performing any mesh clustering tasks. Similar to the k-means algorithm it is attractive due to its elegance and ease of implementation. However, its efficiency for large models is questionable: the two-phases approach, i.e. partitioning and proxy fitting, together with a global priority queue that stores the distortion error does not appear to be ideal. This supposition is more accentuated by the approach of Valette et al. [VC04], which has shown to accomplish the same task, i.e. build a Centroidal Voronoi Diagram (CVD) on a polygonal mesh, but at lower computational effort and with a more simplified data structure.

The aim of this chapter is twofold:

1. We describe[1] a new clustering paradigm, namely the *Energy Minimization by Local Optimization (EMLO)*, which compared to the standard VC performs differently and has desirable proprieties. In Sections 3.1 - 3.2 we describe this clustering algorithm in the context of CVD-based mesh coarsening, as originally proposed in [VC04], [VKC05], [VCP08]. In these sections we only partially touch on the different problems associated with the (uniform and adaptive) CVD-based coarsening of surface polygonal meshes. Different improvements (uniform seed generation, cluster connectivity check, multiplicatively weighted CVD) are also proposed in Section 3.1.5 and Sections 3.2.2 - 3.2.3, respectively. We generalize the Valette approach in Section 3.3 and propose some application areas[2] in Section 3.3.2.

2. Although the Valette approach performs mesh clustering differently compared to the Variational clustering it belongs to the same class of iterative methods. Thus, it suffers from the same inherent problems: choosing the starting number of clusters and seeds' positions. In this chapter we want to show by example this class of associated

[1]Parts of this chapter were published in [CK06], [CK08].
[2]The spherical approximation energy functional in Section 3.3.2 is a completely new material.

problems. The reader can then more comprehensively understand the underlying problems of this class of algorithms. In most cases we give solutions or make constructive analyses of possible solutions, which are then the roots for most of the ideas and algorithms proposed in the rest of this work.

3.1 CVD-based Mesh Coarsening

Mesh coarsening, i.e. the reduction of the mesh complexity in terms of the number of vertices or faces, is still a very exciting field of research. This is mainly due to a growing technological advance in the acquisition and generation systems, which provide meshes of high complexity. For a wide range of applications these meshes are too complex to be used directly. Thus coarsening, i.e simplifying, the mesh can reduce memory requirements, speed up transmission, accelerate different computations such as finite element analysis or collision detection, or be crucial for real-time rendering [HG97], [Coh99], [LRC*02].

Although, there are plenty of techniques to perform mesh coarsening (we review them shortly in Section 3.1.1), the CVD-based approaches proved to be the best regarding the quality of the resulting triangulation. This is due to the intrinsic compactness of the obtained CVD tessellation [DFG99][OBS92]. Thus, CVD provides an optimal strategy for resampling [AdVDI03], [AdVDI05] and coarsening [VC04], [VKC05], [VCP08], [CK06].

A Centroidal Voronoi Diagram is a special form of the Voronoi diagram (VD) where the Voronoi sites (seeds) are also the mass centroid of the associated Voronoi regions, see Section 3.1.2 for more details. The duality between VD and the Delaunay triangulation (DT) in $\mathbb{R}^2$ is well know, see [DBvKOS00], [OBS92]. A DT is obtained by inserting for each Voronoi vertex a triangle where the triangle vertices are the Voronoi seeds. This property is used for triangulation in a CVD-based setting.

For this kind of approach two phases can be identified:

1. Constructing a CVD on the surface of a polygonal input mesh M, see Figure 3.1 (c) on page 33.
2. Triangulating the obtained Voronoi diagram, see Figure 3.1 (d) on page 33. The final coarse mesh (triangulation) is created by inserting one vertex per cluster and connecting them according to the cluster's adjacency.

3.1.1 Overview of Mesh Coarsening Approaches

Simplifying a mesh and at the same time maintaining its original fidelity is certainly challenging. For a boarder overview on the topic see [HG97], [Coh99] and [LRC*02].

As described in [LRC*02], two types of constrains can be identified for mesh coarsening:

- *Fidelity-based*: the mesh is simplified until the difference between simplified and the original mesh is above a user-provided simplification error ϵ.

- *Budget-based*: for a user-specified maximum number of triangles find a simplified mesh with a minimum error ϵ.

To perform the simplification four classes of approaches can be identified:

1. The *refinement methods*, which are coarse-to-fine approaches, start from a base mesh adaptively adding details to the mesh, [EDD*95] or see [HG97] for an overview.

2. The *decimation methods*, which are fine-to-coarse approaches, start from the original mesh by removing the mesh elements: vertices, edges, or faces. Different local simplification operators apply:

   - *Edge collapses* or *vertex-pair collapse* [Hop96], [GH97].
   - *Face collapse* [Ham94], [GHJ*97].
   - *Vertex decimation* [SZL92], [Kle98].
   - *Cell collapse* [RB93].

3. The so-called *remeshing techniques* can also be identified [LSS*98], [KVLS99], [AMD02], [AdVDI03]. Explicit parametric remeshing approaches use global [GGH02] or local [SG03] parameterizations, whereas implicit or volumetric remeshing approaches construct an intermediate volume model [KJ01], [NT03].

4. The *energy minimization methods*, are a new class of algorithms for mesh coarsening. Good examples in this case are the work of Cohen-Steiner et al. [CSAD04] or Valette et al. [VC04], [VKC05], [VCP08], and Chiosa and Kolb [CK06], [CK08]. These approaches simplify the mesh indirectly. First, a clustering according to a given criteria is performed. This is followed, in the first case, by an approximation of the obtained planar clusters by a polygonal mesh, whereas in the second case a Delaunay triangulation is built from the obtained Centroidal Voronoi Diagrams.

At this point it is also important to note a relevant difference between the three above described mesh reduction approaches and the last class of techniques. The former three are specifically designed *only* for mesh reduction whereas the later is more generic and can cope with more tasks, as we show in this and the next chapters.

3.1.2 CVD on a Polygonal Mesh

A *Centroidal Voronoi Diagram (CVD)* is a special form of a *Voronoi Diagram (VD)*. For a more comprehensive discussion on theoretical aspects and practical applicability of the CVD see [DFG99].

Continuous case:

Given a set of k different points $\{\mathbf{z}_i\}_{i=0}^{k-1}$ in the domain Ω, a Voronoi region D_i corresponding to the point $\mathbf{z}_i$ is defined as:

$$D_i = \{\mathbf{x} \in \Omega |\ d(\mathbf{x}, \mathbf{z}_i) < d(\mathbf{x}, \mathbf{z}_j)\ \forall\ j \neq i\} \tag{3.1}$$

where d is the distance measure.

The points $\{\mathbf{z}_i\}_{i=0}^{k-1}$ are called *seeds* or *generators* and the set $\{D_i\}_{i=0}^{k-1}$ is the *Voronoi diagram* of Ω, see for more details [OBS92], [DBvKOS00].

In the general case, the distance d is defined by the metric in a given space. Thus, anisotropic distance (directional distance) can also be considered, as proposed in [LS03], [DW05]. However, for the rest of this work we assume that the distance d is the standard Euclidean distance.

A Centroidal Voronoi Diagram or a Centroidal Voronoi Tessellation [DFG99] is a Voronoi diagram where each Voronoi seed $\mathbf{z}_i$ is also the *mass centroid* $\mathbf{z}_i^*$ of its Voronoi region D_i defined as:

$$\mathbf{z}_i^* = \frac{\int_{D_i} \mathbf{x}\rho(\mathbf{x})d\mathbf{x}}{\int_{D_i} \rho(\mathbf{x})d\mathbf{x}} \tag{3.2}$$

where $\rho(\mathbf{x})$ is a density function.

One of the most important properties of a CVD is that it minimizes the following energy functional:

$$E = \sum_{i=0}^{n-1} \int_{D_i} \rho(\mathbf{x}) \|\mathbf{x} - \mathbf{z}_i\|^2 d\mathbf{x}. \tag{3.3}$$

In other words this energy functional is minimized when D_i's are the Voronoi regions of corresponding seeds $\mathbf{z}_i$ and, at the same time, the $\mathbf{z}_i$'s are the mass centroids of the associated regions D_i, i.e a given tessellation is a CVD.

Common algorithms for constructing a CVD are the Lloyd's method [Llo82], which alter the Voronoi diagram construction and recomputation of the centroids of the associated regions, and the k-means clustering described in Section 2.2.1.

CVD on a 3D surface mesh:

To construct a CVD on a polygonal surface mesh several approaches can be considered, see also [MS09]:

- **Geodesic Distance based Method:** A strict description of the CVD for 3D surface meshes requires the computation of the geodesic instead of the Euclidean distances, as done in [PC04], [PC06] and [SSG03]. However, the geodesic distance computation is very expensive.

- **Euclidean Distance based Method:** In [DGJ03] a *Constrained Centroidal Voronoi Tessellation* is proposed. In this case the Euclidean distance is used and the cluster centroids are constrained, i.e. *(constrained mass centroid)*, to belong to a given surface. However, identifying the constrained centroids is not a trivial task and requires an additional computational effort.

- **Parametrization based Method:** The input mesh is first parametrized to a plane [AdVDI03], [AdVDI05]. In this case the CVD can be performed in the parametric space and then mapped back to the mesh. However, as pointed out in [VCP08], the distortions introduced by the parameterization methods have at least the same shortcomings with regard to the geometric accuracy as using the Euclidean distance methods.

- **Approximation Method:** The work of Valette et al. [VC04], [VKC05], [VCP08] proposed a more efficient method to construct a CVD on a polygonal mesh. The method is stated as a face clustering problem, where the energy functional is minimized according to local queries on the boundaries between clusters. Each CVD region is assumed to consist of a union of several mesh faces, thus it only approximates a Voronoi region.

Constructing a CVD on a surface polygonal mesh is not a trivial task. However, in the context of a mesh coarsening, the approximation method, although it only approximates a CVD, proved to be the best compromise between CVD building efficiency and the resulting triangulation quality [VC04], [VKC05], [VCP08]. Additionally, the approach is very robust allowing a simple integration of different checks and constraints. In the next section we describe this approach in more details.

3.1.3 The Valette Approach

According to [VC04], given a surface polygonal mesh M with faces F_j, we assume that the boundaries of a cluster C_i are a subset of the edges of M. Thus, the C_i consists of a union of several mesh faces F_j. Note that, due to this assumption in the general case the clusters C_i are not Voronoi regions in the strict sense as defined in Eq. (3.1).

The energy functional equivalent to Eq. (3.3) can be defined as:

$$E = \sum_{i=0}^{k-1} \Big(\sum_{F_j \in C_i} \int_{F_j} \rho(\mathbf{x}) \|\mathbf{x} - \mathbf{z}_i\|^2 d\mathbf{x} \Big). \tag{3.4}$$

It can be shown that if approximating each face F_j with constant face density ρ_j by a single point, i.e. by its centroid

$$\boldsymbol{\gamma}_j = \frac{\int_{F_j} \rho(\mathbf{x}) \mathbf{x} d\mathbf{x}}{\int_{F_j} \rho(\mathbf{x}) d\mathbf{x}}.$$

the energy functional defined by Eq. (3.4) can be simplified to

$$E_{CVD} = \sum_{i=0}^{k-1} (\sum_{F_j \in C_i} \rho_j A_j \| \boldsymbol{\gamma}_j - \overline{\boldsymbol{\gamma}}_i \|^2). \tag{3.5}$$

where A_j is the face area and

$$\overline{\boldsymbol{\gamma}}_i = \frac{\sum_{F_j \in C_i} \rho_j A_j \boldsymbol{\gamma}_j}{\sum_{F_j \in C_i} \rho_j A_j}. \tag{3.6}$$

the centroid of the region C_i.

It can be easily proven that substituting the Eq. (3.6) into the Eq. (3.5) yields:

$$E_{CVD} = \sum_{i=0}^{k-1} E_{CVD}(C_i) = \sum_{i=0}^{k-1} (\sum_{F_j \in C_i} m_j \| \boldsymbol{\gamma}_j \|^2 - \frac{\| \sum_{F_j \in C_i} m_j \boldsymbol{\gamma}_j \|^2}{\sum_{F_j \in C_i} m_j}). \tag{3.7}$$

where $\boldsymbol{\gamma}_j$ and $m_j = \rho_j A_j$ is the centroid and the weighted area of the face F_j, respectively.

The equation (3.7) describes the energy of a discrete CVD for a polygonal mesh. However, due to our assumption that Voronoi regions C_i consist of a union of several mesh faces F_j, this is still an energy of an approximated CVD.

Now, constructing an approximated CVD on a polygonal mesh M can be seen as a clustering problem, in which the original mesh faces F_j are assigned to different clusters C_i, i.e. to approximated Voronoi regions, in such a way that the energy E_{CVD}, Eq. (3.7), is minimized.

For a given number of clusters k, Valette et al. [VC04] proposed a very efficient iterative algorithm to perform the minimization.

At the beginning each cluster is assigned to a randomly picked face. Looping over the boundary edge set of each cluster the free faces are set to belong to respective clusters if they are not yet assigned to any other clusters. This way an initial configuration is obtained.

The minimization process works as follows: For each boundary edge e between two clusters C_q and C_p a local test is performed and the following energies for three cases are computed, see Figure 3.15 on page 54:

1. E^0_{CVD} energy: F_m still belongs to C_q and F_n still belongs to C_p.
2. E^1_{CVD} energy: C_q grows and C_p shrinks, i.e. F_m and F_n belong to C_q.
3. E^2_{CVD} energy: C_q shrinks and C_p grows, i.e. F_m and F_n belong to C_p.

The case with the smallest energy is chosen and the cluster configuration is updated accordingly. Thus, the energy functional is iteratively decreased and the final clustering is obtained when no further energy reduction is achieved.

Using the Eq. (3.7) the energies E^0_{CVD}, E^1_{CVD} and E^2_{CVD} are computed as follows:

$$E^0_{CVD} = \sum_{i'} E_{CVD}(C_{i'}) +^* E^0_{CVD} \tag{3.8}$$

$$E^1_{CVD} = \sum_{i'} E_{CVD}(C_{i'}) +^* E^1_{CVD} \tag{3.9}$$

$$E^2_{CVD} = \sum_{i'} E_{CVD}(C_{i'}) +^* E^2_{CVD} \tag{3.10}$$

with $i^{'} \in \{0, \ldots, k-1\} \setminus \{q, p\}$ and

$$\begin{aligned} {}^*E^0_{CVD} = &\sum_{F_j \in C_q \setminus \{F_m\}} m_j \|\boldsymbol{\gamma}_j\|^2 + m_m \|\boldsymbol{\gamma}_m\|^2 - \frac{\| \sum_{F_j \in C_q} m_j \boldsymbol{\gamma}_j \|^2}{\sum_{F_j \in C_q} m_j} + \\ &\sum_{F_j \in C_p \setminus \{F_n\}} m_j \|\boldsymbol{\gamma}_j\|^2 + m_n \|\boldsymbol{\gamma}_n\|^2 - \frac{\| \sum_{F_j \in C_p} m_j \boldsymbol{\gamma}_j \|^2}{\sum_{F_j \in C_p} m_j}. \end{aligned} \tag{3.11}$$

$$\begin{aligned} {}^*E^1_{CVD} = &\sum_{F_j \in C_q \setminus \{F_m\}} m_j \|\boldsymbol{\gamma}_j\|^2 + m_m \|\boldsymbol{\gamma}_m\|^2 + m_n \|\boldsymbol{\gamma}_n\|^2 - \frac{\| \sum_{F_j \in C_q \cup \{F_n\}} m_j \boldsymbol{\gamma}_j \|^2}{\sum_{F_j \in C_q \cup \{F_n\}} m_j} + \\ &\sum_{F_j \in C_p \setminus \{F_n\}} m_j \|\boldsymbol{\gamma}_j\|^2 - \frac{\| \sum_{F_j \in C_p \setminus \{F_n\}} m_j \boldsymbol{\gamma}_j \|^2}{\sum_{F_j \in C_p \setminus \{F_n\}} m_j}. \end{aligned} \tag{3.12}$$

$$\begin{aligned} {}^*E^2_{CVD} = &\sum_{F_j \in C_q \setminus \{F_m\}} m_j \|\boldsymbol{\gamma}_j\|^2 - \frac{\| \sum_{F_j \in C_q \setminus \{F_m\}} m_j \boldsymbol{\gamma}_j \|^2}{\sum_{F_j \in C_q \setminus \{F_m\}} m_j} + \\ &\sum_{F_j \in C_p \setminus \{F_n\}} m_j \|\boldsymbol{\gamma}_j\|^2 + m_n \|\boldsymbol{\gamma}_n\|^2 + m_m \|\boldsymbol{\gamma}_m\|^2 - \frac{\| \sum_{F_j \in C_p \cup \{F_m\}} m_j \boldsymbol{\gamma}_j \|^2}{\sum_{F_j \in C_p \cup \{F_m\}} m_j}. \end{aligned} \tag{3.13}$$

Observe that, for comparing the global energies E^0_{CVD}, E^1_{CVD} and E^2_{CVD} in Eqs. (3.8) - (3.10) there are terms which are identical, thus they are irrelevant. The energy terms which remain are:

$${}^{**}E^0_{CVD} = -\frac{\| \sum_{F_j \in C_q} m_j \boldsymbol{\gamma}_j \|^2}{\sum_{F_j \in C_q} m_j} - \frac{\| \sum_{F_j \in C_p} m_j \boldsymbol{\gamma}_j \|^2}{\sum_{F_j \in C_p} m_j}. \tag{3.14}$$

$${}^{**}E^1_{CVD} = -\frac{\| \sum_{F_j \in C_q \cup \{F_n\}} m_j \boldsymbol{\gamma}_j \|^2}{\sum_{F_j \in C_q \cup \{F_n\}} m_j} - \frac{\| \sum_{F_j \in C_p \setminus \{F_n\}} m_j \boldsymbol{\gamma}_j \|^2}{\sum_{F_j \in C_p \setminus \{F_n\}} m_j}. \tag{3.15}$$

$${}^{**}E^2_{CVD} = -\frac{\| \sum_{F_j \in C_q \setminus \{F_m\}} m_j \boldsymbol{\gamma}_j \|^2}{\sum_{F_j \in C_q \setminus \{F_m\}} m_j} - \frac{\| \sum_{F_j \in C_p \cup \{F_m\}} m_j \boldsymbol{\gamma}_j \|^2}{\sum_{F_j \in C_p \cup \{F_m\}} m_j}. \tag{3.16}$$

It is important to recognize that comparing the reduced energies $^{**}E^0_{CVD}$, $^{**}E^1_{CVD}$ and $^{**}E^2_{CVD}$ is equivalent to comparing the total energies E^0_{CVD}, E^1_{CVD} and E^2_{CVD} in Eqs. (3.8)-(3.10), but at lower memory and computational cost. Indeed, for each cluster *only* the values $\sum m_j \boldsymbol{\gamma}_j$ and $\sum m_j$ need to be stored and a fast cluster update is possible in this case. Additionally, for each face F_j the values of $m_j \boldsymbol{\gamma}_j$ and m_j can be computed only once at the beginning, thus making the overall computational cost very low.

Note that, the reduced energy form in the Eqs. (3.14)-(3.16) was only possible due to a special CVD energy function formulation in Eq. (3.7). We call such an energy form an ***incremental energy formulation***.

Note that, the Valette approach does not require any global priority queue and there is no explicit proxy fitting or seed identification step in the algorithm. These are the major elements of this clustering approach compared to the classical Variational clustering method that allowed us to solve the same problem, i.e. constructing an approximated CVD, but in a more efficient manner.

3.1.4 Uniform CVD-based Mesh Coarsening

There are many applications where high quality triangulations are required, e.g. for finite elements analysis. To speed up the simulations it is desirable to make the mesh as coarse as possible. At the same time to have a stable simulation a mesh with well-shaped triangles (triangles that are as close as possible to equilateral) is required.

Given an input surface polygonal mesh M with m faces F_j, we want to coarsen the mesh M to a k output number of vertices. This means building a CVD clustering with k clusters. We use the Valette algorithm from Section 3.1.3, followed by a triangulation.

A uniform density function $\rho(\mathbf{x})$ needs to be assumed to yield a uniformly coarsened mesh, i.e. a triangulation with well-shaped triangles. In this case we can safely remove any face weight ρ_j from our computations, i.e. we set $\rho_j = 1$ The energy $^{**}E^0_{CVD}$ in Eq. (3.14) becomes:

$$^{**}E^0_{CVD} = -\frac{\| \sum_{F_j \in C_q} A_j \boldsymbol{\gamma}_j \|^2}{\sum_{F_j \in C_q} A_j} - \frac{\| \sum_{F_j \in C_p} A_j \boldsymbol{\gamma}_j \|^2}{\sum_{F_j \in C_p} A_j}.$$

The same applies to $^{**}E^1_{CVD}$ and $^{**}E^2_{CVD}$ in Eqs. (3.15)- (3.16), where A_j is the area of the face F_j.

Thus for each face F_j we compute and save only the values A_j and $A_j \boldsymbol{\gamma}_j$, which are computed only once before the algorithm starts. Correspondingly, for each cluster C_i two values $\sum A_j$ and $\sum A_j \boldsymbol{\gamma}_j$ are saved and updated.

Figure 3.1 shows the result of a uniform mesh coarsening applied to the sphere model. As expected, the algorithm yields a uniform (well-shaped triangles) output triangulation. The final triangulation is created by inserting one vertex per cluster at the position equal to the cluster centroid $\overline{\boldsymbol{\gamma}}_i = \sum A_j \boldsymbol{\gamma}_j / \sum A_j$, which is easily computed because the values $\sum A_j$ and $\sum A_j \boldsymbol{\gamma}_j$ are already available. The vertices are connected, as described in [VC04], according to the cluster's adjacency, where a triangle is created for each vertex where three Voronoi regions meet.

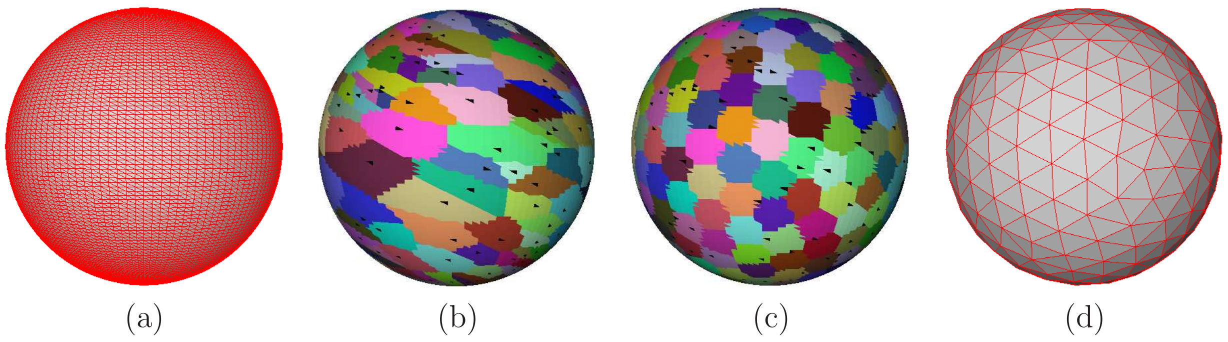


Figure 3.1: (a) Sphere model with 28.5k triangles. Uniform coarsening results for $k = 300$ clusters which results in a mesh with 300 vertices: (b) Initial cluster growing. (c) Uniform CVD clustering. (d) Triangulation of (c).

Although, the result is visually pleasing there are many issues related to the CVD-based mesh coarsening. A discussion of these issues follows.

Initialization problem:

The problem of a "good" initialization, i.e. the starting positioning of seeds, has a long history as the problem of identifying the "true" number of clusters. Here one may ask if or why is the initialization so important, because the optimization step in any case provides a "good" result regardless of the initial configuration, as presented in Figure 3.1. There are two important aspects which are strictly related to the initialization:

1. **Faster convergence.** If the initialization is very close to the final result, then the algorithm converges very quickly thus becoming very efficient.
2. **More qualitative results.** A better initialization always leads to a better clustering result, i.e lower clustering energy. It is well known that during optimization the clusters can be trapped in some region and cannot "jump" easily to other high energy regions to lower the total energy.

A "good" initialization is one that is very close or mimics the final solution of a given problem. It can be observed in the Figure 3.1(b) that a random sampling followed by an initial growing is far away from the final CVD result.

A random sampling is the simplest and most often employed strategy. Sometimes a multiple run random initialization approach (computationally very expensive) is also employed, there the solution with smallest energy is chosen. However, a random initialization leads to a random results, as presented in Figure 3.2. Although the number of clusters is fixed the resulting triangulations differ and some feature points are lost.

The *Farthest Point* initialization [KJKZ94] is more appropriate in this context. It starts with some random seed and then iteratively adds a seed with maximum distance to the already existing seeds. Although this is a good strategy, still a high computational complexity $O(km)$ (k number of clusters and m total number of elements) makes it less attractive.

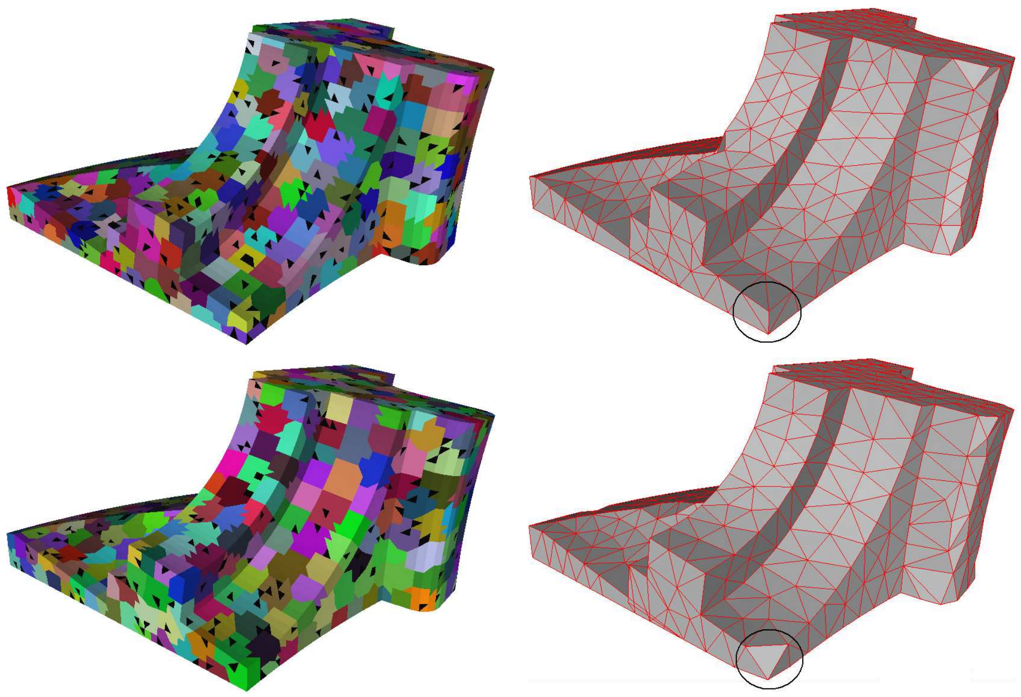


Figure 3.2: Results for two different random initializations with $k = 527$ clusters. (left) Uniform clustering. (right) Corresponding triangulation.

In Section 3.1.5 we propose a new initialization approach, which proves to be a good solution for a uniform CVD-based clustering.

Vertex placement:

In [VC04] the final triangulation is enhanced using as a vertex in the triangulation an original mesh vertex which is the closest to the cluster centroid $\overline{\gamma}_i$ (Eq. (3.6)). Although, this works well for planar or highly tessellated meshes, Figure 3.3(d) depicts an example where this fails to work properly. Note that, the result presented in Figure 3.3(d) is approximately identical to that in Figure 3.3(c) where only the cluster centroid $\overline{\gamma}_i$ is used. In Figure 3.3(f) we exemplify this situation showing that the closest vertex is a suboptimal choice, not to say that identifying these vertices requires an additional computational effort.

In [VKC05] and [VCP08] the *Quadric Error Metrics (QEM)* [GH97] was employed to relocate the cluster centroid position. As proposed in [Lin00], for each face F_j an associated 4×4 quadric matrix Q_j is computed, which provides a very convenient representation for evaluating the squared distance between any point and the plane containing F_j. Then, for each cluster C_i a sum of all QEM of the cluster's faces can be used to compute the "optimal" cluster's centroid position. In this case the centroid position has the smallest deviation from all planes containing F_j in a cluster.

In [VKC05] the QEM is used in the final post-processing to enhance the quality of the approximating mesh. Figure 3.3(e) shows the result of using the QEM for final vertex

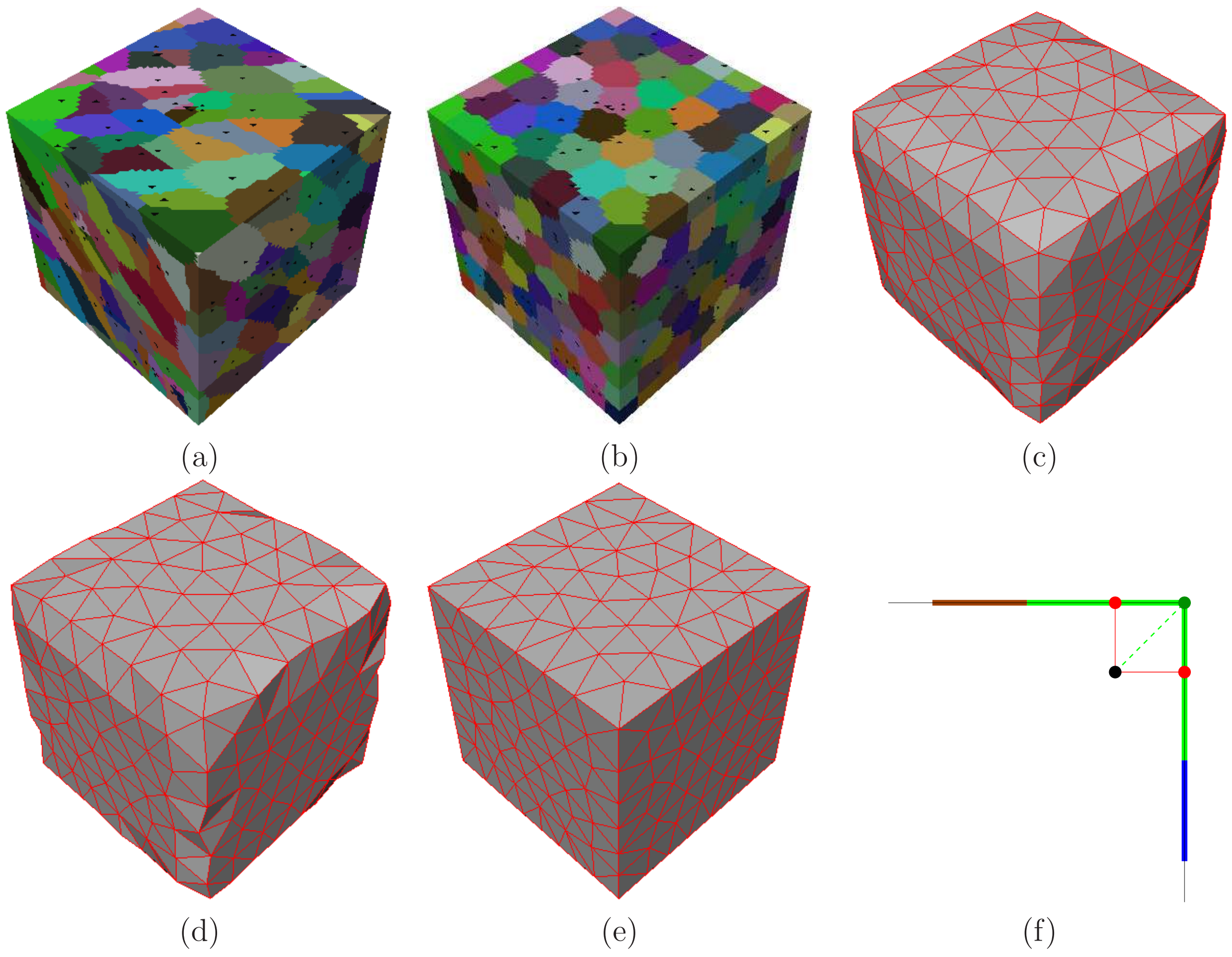


Figure 3.3: Cube model with 30k triangles. Uniform coarsening results for $k = 274$: (a) Initial cluster growing. (b) Uniform CVD clustering. (c) Triangulation of (b) with cluster centroid as vertex. (d) Triangulation of (b) with closest original mesh vertex to the cluster centroid as vertex. (e) Triangulation of (b) with the QEM approach. (f) Illustration of an inadequate mesh vertex selection according to the smallest distance. The red vertices are the closest and not the green one.

placement, which indeed outperforms the results presented in Figure 3.3(c)-(d). In [VCP08] this post-processing is embedded in the clustering process itself. At each iteration a better cluster centroid is computed and injected in the energy computation. This is in the same spirit as done for constrained CVD [DGJ03].

However, additionally to an increased computational complexity the QEM approach fails sometimes to provide reliable results. Figure 3.4 provides such an example. In a given situation a cluster (composed from three intersecting planes) contains two feature lines which are almost parallel, thus the optimal vertex position is far away from the original mesh vertices. To fix this problem additional checks are required to detect and project the vertex on the original mesh.

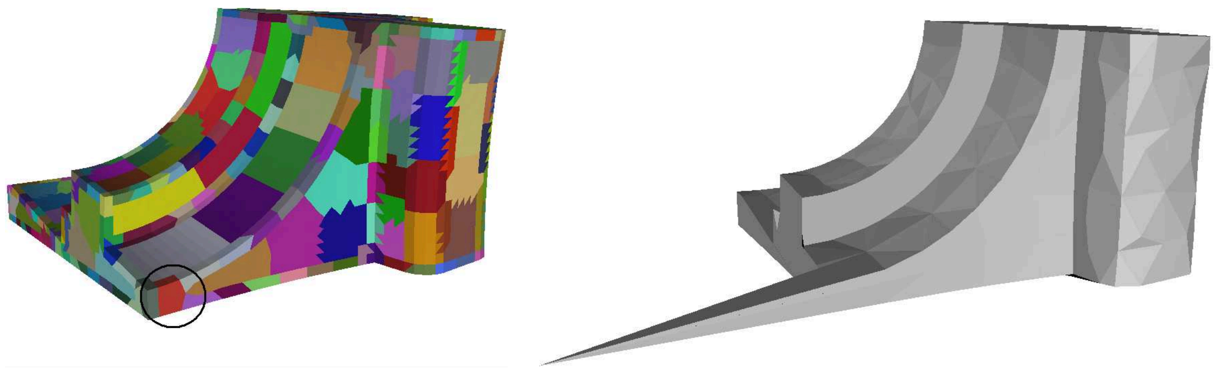


Figure 3.4: Triangulating with QEM post-processing. (left) Clustering result, number of clusters $k = 527$. (right) Triangulation of (left) with QEM positioning.

It is clear that the above described possibilities lack efficiency in many situations. A possible solution in this case is to use an adaptive CVD [VKC05], [CK06] where each triangle gets a specific density related to the surface property. Thus if the higher curvature regions have a larger weight than lower curvature regions the cluster centroid $\overline{\gamma}_i$ (Eq. (3.6)) will be "attracted" to the mesh surface, which in turn gives a lower approximation error. Section 3.2 shows how this approach works.

Connectivity check for valid triangulation:

To obtain a valid triangulation in the final clustering each cluster has to be a 1-connected set of faces. To ensure this constraint in [VKC05] a three-step approach is proposed:

1. Perform the clustering without any checks.
2. Clean any disconnected cluster, by removing the smallest components and leaving the largest component only.
3. Perform the clustering with an additional embedded check, which does not allow the clusters to get disconnected.

As pointed out in [CK06] the algorithm proposed in [VKC05] for the third step does not guaranty a genus zero clusters, which is also a prerequisite for a valid triangulation. In Section 3.1.5 we propose a new *Vertex Boundary-edge Count* approach, which is able to handle such configurations.

3.1.5 Improvements

In this section we describe different improvements that relate to CVD-based mesh coarsening [CK06]. First, we present an algorithm for generating an initial set of seeds that is approximately uniformly distributed over the input mesh. Secondly, we describe an algorithm that prevents the clusters from getting disconnected during the clustering process.

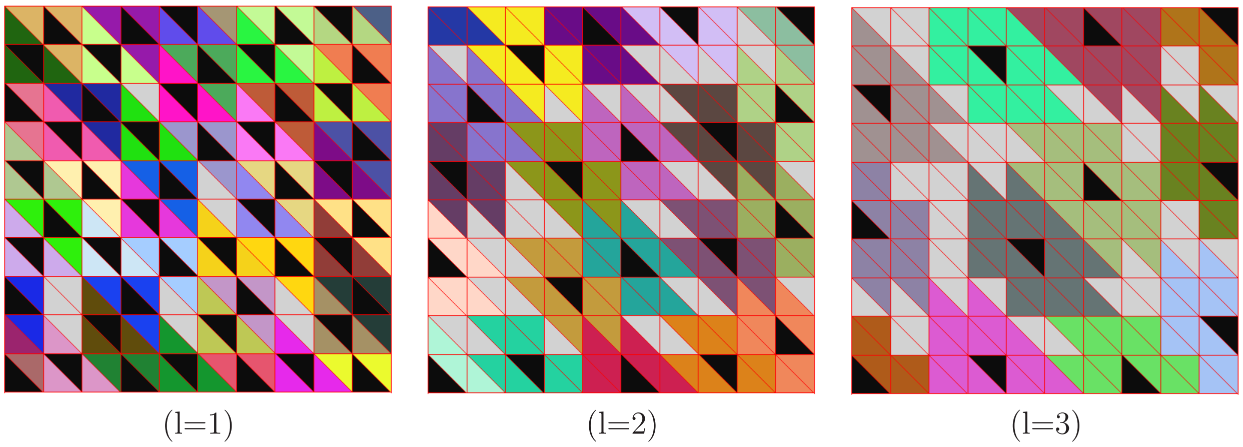


Figure 3.5: Result of the l-neighborhood initialization for different levels l.

l-Neighborhood Initialization:

Given a polygonal mesh M with faces F_j. Assume that all valid faces are contained in an array (F_j^{valid}). Holes are considered to be invalid. We also assume that each face has references to its neighbors, which is a standard data element in mesh data structures such as the half-edge data structure [Män88].

The basic idea is to assume that each face is a potential starting seed. Then for a user specified level l a check is done to see if the lth-neighborhood of a given seed contains only `valid` faces. If that is the case, the face is reported as a seed and all faces in the l-neighborhood are set as `invalid`, thus no other seed can contain them in its l-neighborhood. Interestingly, the algorithm does not use any distance computation, it merely uses the face adjacencies across edges (*face-neighborhood*). The Algorithm 3.1 describes the steps of the l-neighborhood[3] initialization.

Algorithm 3.1. *(l-neighborhood Initialization)*

```
∀ face in $(F_j^{valid})$: setValid(face)
∀ face ∈ $(F_j^{valid})$ do:
    if( isValid(face) && isValid($l$−neighborhood of face) )
        setInvalid($l$−neighborhood of face)
        addToSeedSet(face)
```

For example, for level $l = 2$ a given face is reported as a seed if its neighboring faces are `valid` and, at the same time, each neighboring face also has its neighboring faces `valid`, as presented in Figure 3.5.

The algorithm selects a set of seed faces, whose l-neighborhoods do not overlap. In a brute-force implementation, for a triangulated mesh this algorithm has a runtime com-

[3]In [CK06] the original name of the algorithm was k-neighborhood initialization. However, throughout this work k is related to the number of clusters. To avoid the confusion between k number of levels, the name of the algorithm is changed to l-neighborhood, where l stands for levels.

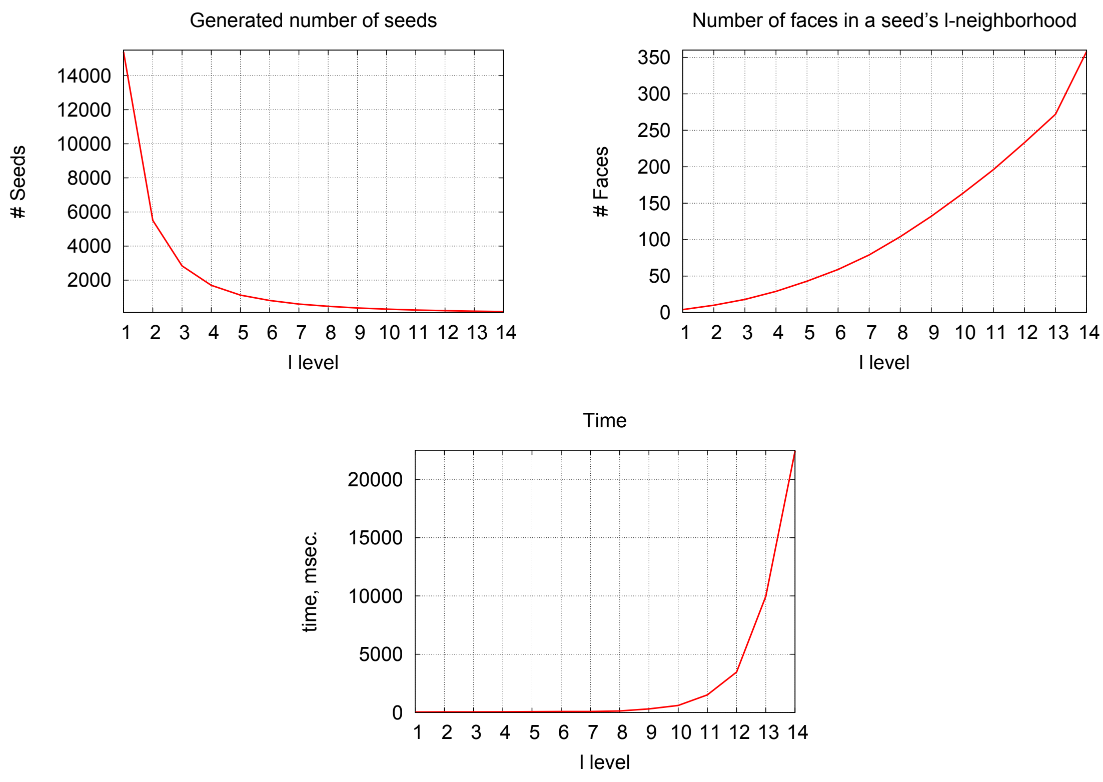


Figure 3.6: Result of l-neighborhood initialization for the Bunny model.

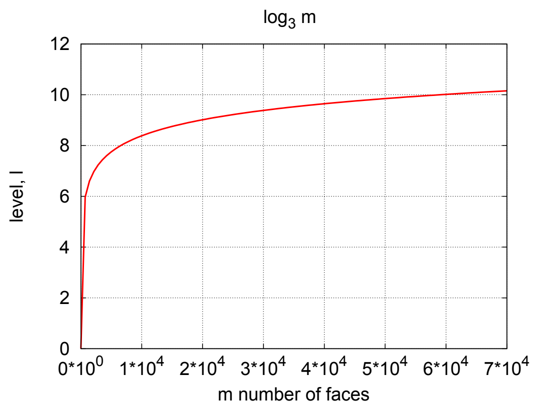


Figure 3.7: Plot of $log_3 m$ for Bunny model consisting of $7 \cdot 10^4$ faces.

plexity of 3^l to identify a single seed or a l-neighborhood, see Figure 3.6. The number of seeds is of the order of $m/(2.5l^2)$, where m is the total number of faces, yielding an overall complexity of $(m \cdot 3^l)/(2.5l^2)$.

If comparing l-neighborhood initialization with Farthest Point initialization [KJKZ94], which has $O(km)$ complexity, then the l-neighborhood approach is better suited for values of $l \leq log_3 m$. Figure 3.7 depicts an example for Bunny model ($7 \cdot 10^4$ faces), which indicates

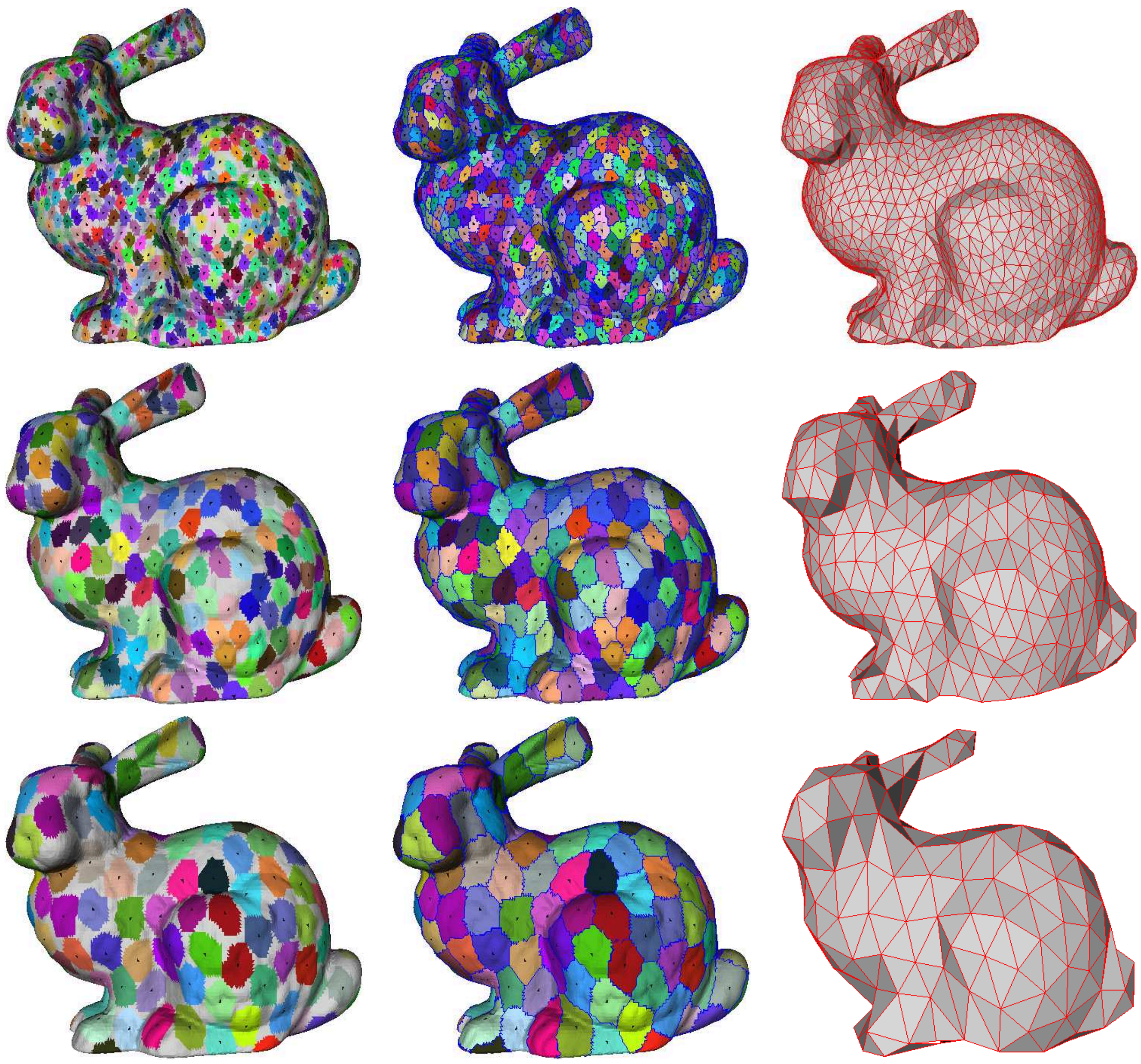


Figure 3.8: Result of the l-neighborhood initialization for the Bunny model with level $l = 4$, $l = 8$ and $l = 12$, respectively. (a) Obtained seeds and their l-neighborhood; (b) Result of the initial cluster growing; (c) Triangulation of (b) using the cluster centroid as vertex.

that for $l \leq 10$ we have a more efficient initialization compared to the Farthest Point initialization.

Observe in Figure 3.5 that there are some faces that are not assigned to any given seed's l-neighborhood. This is due to the fact that the algorithm simply works on the array of faces trying to identify the next face with `valid`, i.e. not yet covered, l-neighborhood.

The algorithm possess a number of important features:

- For a mesh with uniformly sized triangles the resulting l neighborhoods are nearly disk-shaped. This yields very good starting seed sets for algorithms that try to construct equally sized and compact clusters, such as the uniform mesh coarsening in Section 3.1.4.
- The algorithm is in the same spirit as farthest point initialization [KJKZ94], although no distance computations are involved.
- In some situations the result of the l-neighborhood initialization is already good enough for a final mesh coarsening. This is especially important if the user wants to evaluate the final mesh resolution interactively, before deciding on the level l or on the number of clusters k to be used in the overall optimization. In Figure 3.8 we give such an example.

It must be recognized that the l-neighborhood algorithm gives an arbitrary number of clusters k, which is related to the level l. However, some applications may require a fixed number of seeds. In this case a modified version of the algorithm can be used to reduce the number of obtained seeds. First, an initial growing based on the given l-neighborhoods is performed. After this, the area of the resulting clusters is checked and the number of seeds is reduced by merging the three neighboring clusters having a common vertex with the smallest summed area. The seed's new position is assigned to the common vertex of the three original clusters. Note that, the same idea can be used to avoid inefficient initializations for large l, thus starting with lower l and then adding new seeds at the common vertex of three neighboring clusters.

Connectivity Check:

In the following, consider a boundary edge c between two adjacent clusters, see Figure 3.9(a). We check whether the red cluster can grow in the direction of the edge c, i.e. whether the triangle $\Delta(\mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3)$ may switch to the red cluster, without yielding a disconnected cluster. Let n and p denote the next and previous edges w.r.t. c for the considered triangle, respectively.

Our algorithm is related, although developed independently, to the work of Valette et al. [VKC05]. They relate their approach solely on the check of the edges n and p and on vertex $\mathbf{V}_2$. In their implementation growing is allowed if: n or p are exclusively boundary edges. Otherwise, if n and p are not boundary edges, all triangles in the 0-ring of the vertex $\mathbf{V}_2$ should be of the same cluster. Their approach has the disadvantage, that the check fails to handle the disconnectivity as presented in Figure 3.9 (c).

Our algorithm is based on an additional counter for each vertex storing the number of adjacent cluster boundary edges, referred to as *vertex boundary-edge count*. After the initial growing, the counter is set to the number of cluster boundary edges. Thus, for example, in

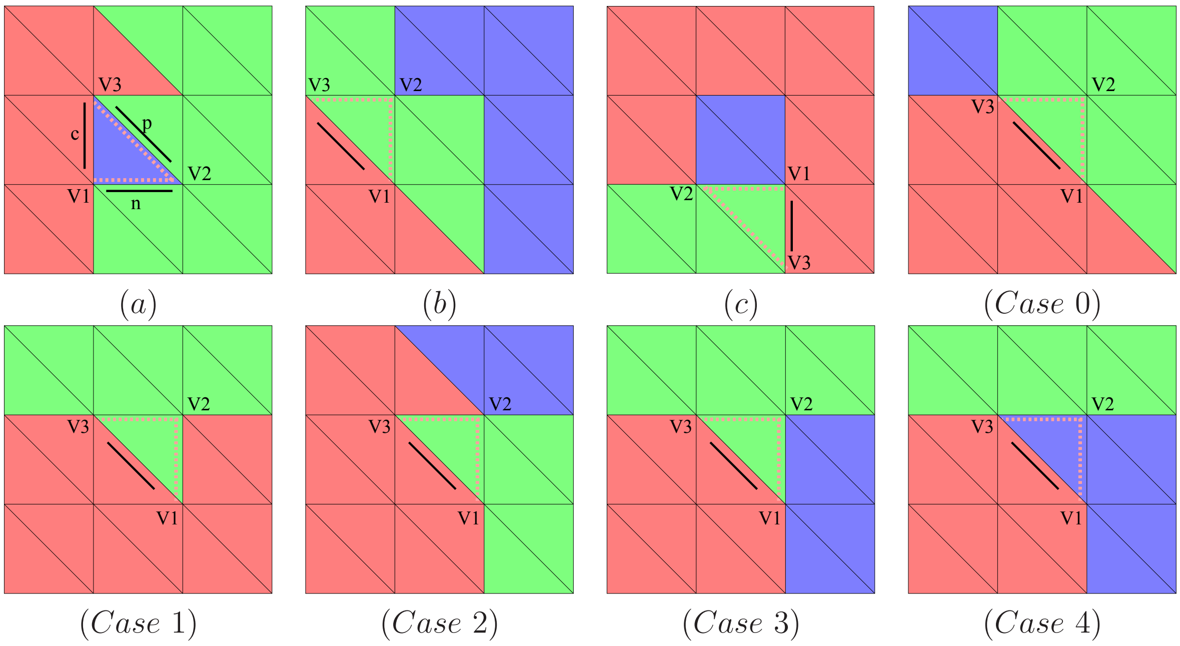


Figure 3.9: Connectivity check configurations. (a) Boundary edge c is one which points from vertex V_1 to V_3; n and p denote the next and previous edges w.r.t c. (b) - (c) Growing leads to disconnected and non-zero genus clusters, respectively. (case0) - (case4) Allowed growing configurations.

Figure 3.9 (case 0) vertex $\mathbf{V}_1$ will have count 2, $\mathbf{V}_3$ will have count 3, whereas $\mathbf{V}_2$ still has a count equal to zero.

Clearly, growing does not lead to a disconnected cluster, if the boundary-edge count for $\mathbf{V}_2$ is zero. Also, if only one of the edges n or p is a boundary of the *current cluster*, w.r.t. edge c, growing does not lead to an invalid situation. Otherwise, if n XOR p is a boundary of the *different cluster*, we check if vertex $\mathbf{V}_2$ has no adjacent boundary of the current cluster, thus avoiding the configuration in Figure 3.9 (c), i.e. a cluster with genus > 0.

Putting this together, we have the following connectivity check as presented in Algorithm 3.2.

Algorithm 3.2. *(Vertex Boundary-Edge Count Algorithm)*

```
if( count(V2) == 0 ) OR // Case 0
( isBoundaryOfCurr(n) XOR isBoundaryOfCurr(p) ) OR // Case 1,2
( (isBoundary(n) XOR isBoundary(p))
     AND !V2.hasBoundaryEdge(curr) ) // Case 3,4
Then allow growing and update vertex counter
```

Where *curr* denotes the current cluster and *V2.hasBoundaryEdge(curr)* checks for boundary edges for the current cluster at $\mathbf{V}_2$.

Using this check the following situations are avoided:

- The adjacent cell represent a single cluster. Here, a growing would lead to an implicit deletion of one cluster, Figure 3.9 (a).
- Splitting the cluster into two parts. This is the only situation that can possibly cause a splitting of the green cluster, Figure 3.9 (b).
- Growing may result in nonzero genus clusters, Figure 3.9 (c).

The update of the boundary-edge counter for vertices after the growing is performed is quite simple. With respect to the cases noted in the above algorithm, we have:

`Case 0:` Update: `count(v2) = 2`.

`Case 1:` n is a boundary of the current cluster.

Update: `count(V1) = 0`.

`Case 2:` p is a boundary of the current cluster.

Update: `count(V3) = 0`.

`Case 3:` n is a boundary of a different cluster.

Update: `count(V1) --; count(V2) ++`.

`Case 4:` p is a boundary of a different cluster.

Update: `count(V3) --; count(V2) ++`.

Note, that (case 0) is the most encountered situation during cluster optimization, which leads to a very efficient check in this situation. Cases 3 and 4 are the most expensive checks, but they do not occur very frequently.

3.2 Adaptive CVD-based Mesh Coarsening

Adaptivity is a key for a better approximation of high curvature regions of the underlying surface mesh. Figure 3.10 provides such an example, where it is easily recognized that case ($\gamma = 4$) better approximates the surface than case ($\gamma = 0$) does.

As pointed out in [DW05], the aspect ratio and orientation of the triangles in the coarse mesh must depend on the underlying problem. Thus, different metrics apply: isotropic [SAG03], [AdVDI03], [AdVDI05], [VKC05] or anisotropic [LS03], [DW05], [VCP08]. The anisotropic case is of interest in boundary viscous or fluid flow simulations, where the solutions often have an anisotropic behavior, and thus anisotropic meshes are desirable. In the following we only consider the isotropic case and show how using the weighted CVD yields an adaptive coarsening. In Sections 3.2.2 - 3.2.3, in the context of feature-preserving coarsening, we will show that an anisotropic behavior is still obtainable using a multiplicative weighted CVD and a special cluster's weight definition, yielding feature elongated clusters.

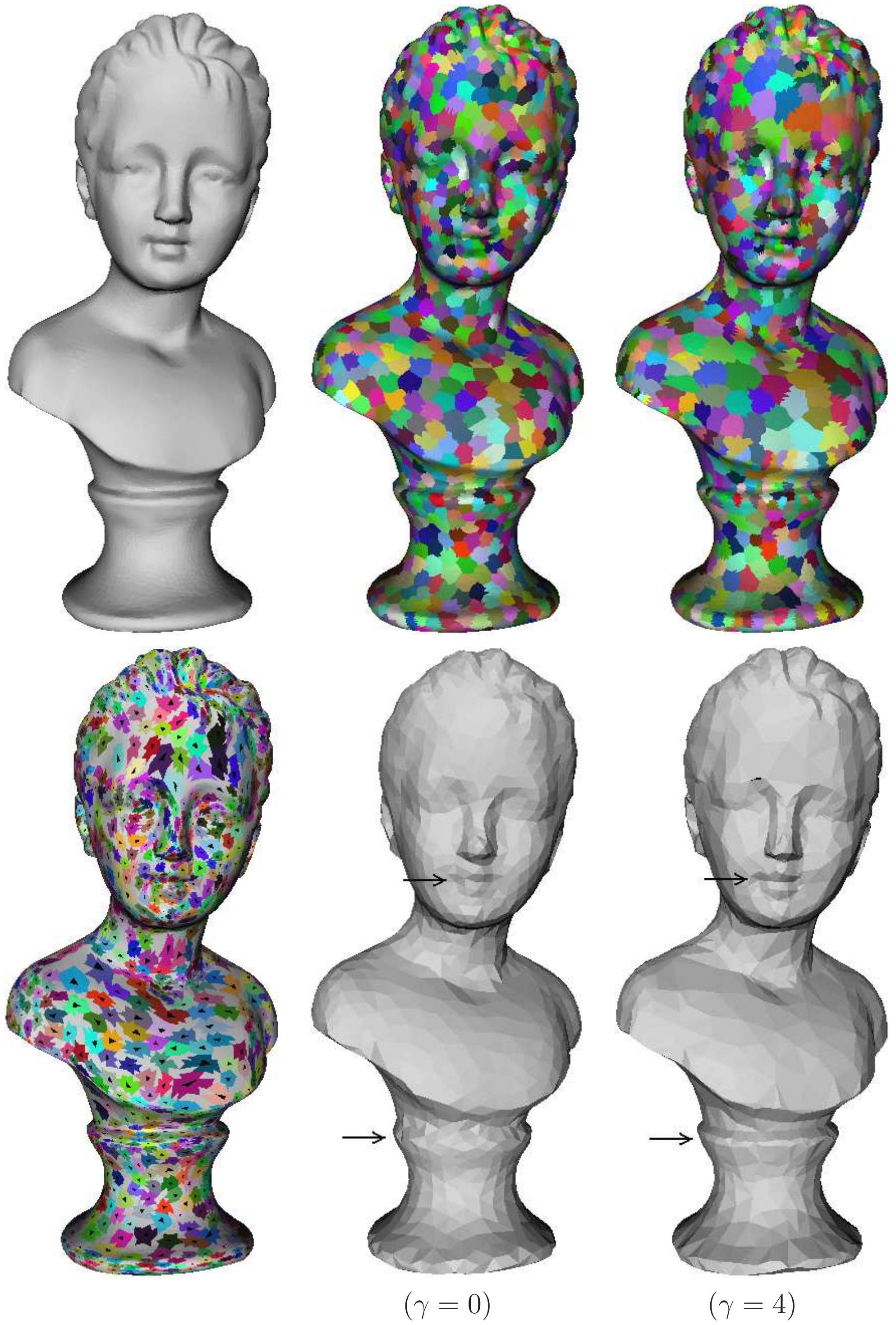


Figure 3.10: A comparison between uniform and adaptive coarsening. Top left: The original Bust model consists of 30.6K vertices. Bottom left: Initialization with l-neighborhood for level $l = 3$ resulting in $k = 2272$ clusters. Top: clustering result and bottom: corresponding triangulation, for different values of γ, respectively.

3.2.1 Nonuniform CVD-based Mesh Coarsening

For a better approximation it is generally expected to have smaller sized clusters (Voronoi regions) in higher curvature areas and larger sized ones in lower curvature areas, resulting in more or less vertices in these regions, respectively.

To achieve such a behavior a nonuniform density function $\rho(\mathbf{x})$ must be used. This is in contrast to uniform coarsening where uniform density is assumed, see Section 3.1.4. Under the assumptions made in Section 3.1.3 each face has its own weight ρ_j. Thus, ρ_j needs to be related to the surface properties, e.g. curvature.

In [VKC05] the density ρ_j was related to the local principal curvatures $k_{j,1}$ and $k_{j,2}$ as $\rho_j = \left(\sqrt{k_{j,1}^2 + k_{j,2}^2}\right)^\gamma$. Here γ is the gradation parameter which controls how different regions are weighted. Note, that for $\gamma = 0$ a uniform coarsening is obtained. This density-curvature relation is in the same spirit as proposed in [AMD02], [AdVDI03], [AdVDI05].

Different curvature estimation techniques, e.g. [CSM03] [Rus04] [CP05], can be employed to deal efficiently with noise or bad sampling conditions present in the input mesh, for a more comprehensive overview of existing techniques see [GG06] and [BPK*08]. However, due to a relatively high time complexity, see for example [VKC05] where the curvature computation takes longer than the actual clustering process, we propose to use a slightly simpler and thus computationally more efficient approach to define ρ_j. Based on the observation that generally the normal field discontinuities directly indicate the mesh features, e.g. a face which belongs to a feature will have large deviation of its normal w.r.t. its neighbors, we set ρ'_j equal to the mean of the normal difference of the faces in the 1-ring of face F_j. For the face F_j with vertices $\mathbf{V}_{i_1}, \ldots, \mathbf{V}_{i_m}$we have:

$$\rho'_j = \frac{1}{n} \sum_{i=1}^{m} \sum_{F_k \text{ adj. to } V_i} \|\mathbf{n}_j - \mathbf{n}_k\| \tag{3.17}$$

where $\mathbf{n}_k$ is the normal of face F_k and $n = \sum_{i=1}^{m} \sum_{F_k \text{ adj. to } V_i} 1$.

Note that, the ρ'_j is zero for planar regions. Thus, a mapping is required to prevent division by zero in Eq. (3.6). One could set the density as $\rho_j = (\rho'_j + \Delta)^\gamma$ (we use $\Delta = 10^{-2}$). Or a linear mapping from $[\rho'_{min}; \rho'_{max}]$ to $[1; \rho^{new}]$ as:

$$\rho_j = \frac{\rho'_{max} - \rho'_j}{\rho'_{max} - \rho'_{min}} + \rho^{new} \frac{\rho'_j - \rho'_{min}}{\rho'_{max} - \rho'_{min}}$$

Where γ or ρ^{new} are user defined parameters. For large values of these parameters the cluster centroids move towards cells with higher density, resulting in more vertices in these areas, see Figure 3.10.

The results presented in the Figure 3.10 were obtained using the l-neighborhood algorithm for initialization. To ensure a valid triangulation the vertex boundary-edge count approach (Section 3.1.5) is directly embedded in the optimization process, thus preventing the clusters from getting disconnected. The vertices of the final coarsened mesh are chosen to be equal to the centroid of the cluster, which are already computed during the clustering process. This makes the final triangulation a simple and fast operation.

Although, the results are visually pleasing there are two very important aspects which relate to the adaptive coarsening:

Initialization Problem:

The use of the l-neighborhood initialization for an adaptive clustering is mainly due to our observation that a compact initialization is less prone to provide disconnected clusters than a non compact one, e.g. random initializations always provide elongated clusters, see Figure 3.1 (b) on page 33. However, for an adaptive clustering such an initialization is suboptimal because it provides, in general, a uniform sampling.

The solution proposed in [VKC05] can be used to alleviate this problem. A global average cluster density $AD_{C_i} = \frac{1}{k}\sum_{j\in C_i}\rho_j$ is computed first. Then a random face is picked to start growing a cluster C_i until its cumulated density reaches AD_{C_i}. Although, it proved to work well in practice, this technique has some drawbacks, namely:

- *Random behavior*: each starting face is picked randomly to grow a cluster with average cluster density AD_{C_i}. This results in a random behavior of the clustering result with the same problems as described in Section 3.1.4 (Figure 3.2 on page 34).
- *Discretization problems*: because we operate on a polygonal mesh, as pointed out in [VKC05], it often happens that some clusters remain to be initialized and there are no free faces available to start with. The authors proposed, in this case, to pick at random a face and assign it to a given cluster. This may lead to more disconnected clusters, because the picked face most probably will be inside of some cluster.
- *No energy functional support*: Although this technique works very well in practice, one must recognize that it is only a good initialization heuristic. It does not provide an initialization which supports a given energy functional. As an example, for Multiplicatively Weighted CVDs, which we will propose in Section 3.2.2, it will not provide an appropriate initialization that can take into account the cluster's weight.

In this respect one may be tempted to reformulate the l-neighborhood initialization and use the average cluster density AD_{C_i} to stop at a lower level to mimic the same behavior. However, finally we will end up with the same set of problems as mentioned above.

Influence of Face Density and Cluster Weight:

Assume that we are building an adaptive (weighted) CVD on a model as depicted in Figure 3.11. As expected, for large values of γ some clusters get smaller or bigger in high or low curvature areas, respectively. However, as indicated in Figure 3.11 ($\gamma = 2$ to $\gamma = 6$), although larger values of γ are used, some highly weighted clusters on the model features do not change their size or get elongated in the direction of low homogeneously weighted regions. Such a behavior contradicts the general expectation that highly weighted clusters always decrease their size as γ increases. The expected behavior in this case is as presented in Figure 3.11 (w_i^A).

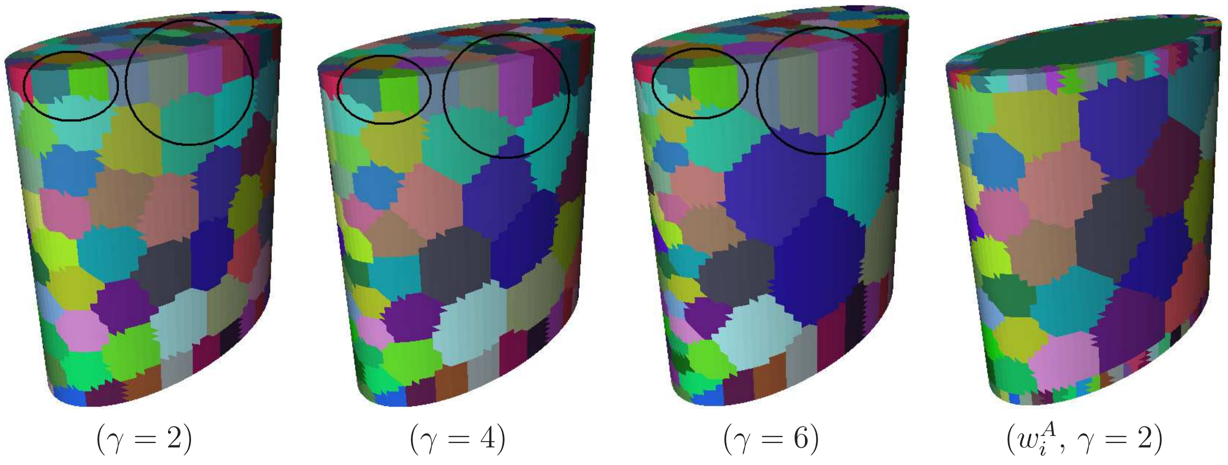


Figure 3.11: Result of an adaptive CVD for different values of γ. (w_i^A) Result obtained with multiplicatively weighted CVD (Section 3.2.3).

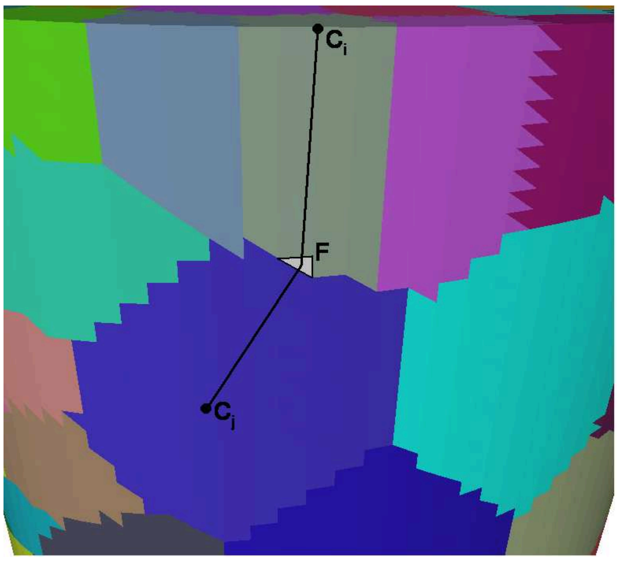


Figure 3.12: Final clustering configuration for two clusters C_i and C_j. The face F is the triangle under consideration.

In general, increasing the values of γ attracts the cluster centroids to higher density regions, as can be seen in Figure 3.11. However, for clusters on the model features, increasing γ has no substantial effect on the clusters centroids because they are already “fixed” at these positions. The only effect for such clusters is that their total weight increases.

As an example, consider the configuration as presented in Figure 3.12. Here the boundary update between cluster C_i (lies on the model feature) and its adjacent one C_j (not on the model feature) is done. For a boundary face F the energy contribution to the total energy is given by $\rho_f\|\boldsymbol{\gamma}_f - \overline{\boldsymbol{\gamma}}_i\|^2$ and $\rho_f\|\boldsymbol{\gamma}_f - \overline{\boldsymbol{\gamma}}_j\|^2$, respectively. Increasing γ moves the centroid $\overline{\boldsymbol{\gamma}}_j$ of C_j in the direction of higher density regions where the centroid $\overline{\boldsymbol{\gamma}}_i$ of C_i remains almost fixed. In this case the total energy contribution of two clusters is only influenced by the distance between face-cluster and the density ρ_f of the face under consideration, but not the cluster weight.

To account for cluster weight it becomes clear that we need methods that can weight the face-cluster distances in different ways. The energy most probably must be given by $w_i \rho_f \|\boldsymbol{\gamma}_f - \overline{\boldsymbol{\gamma}}_i\|^2$ and $w_j \rho_f \|\boldsymbol{\gamma}_f - \overline{\boldsymbol{\gamma}}_j\|^2$, with w as cluster weight. Thus, we introduce the Multiplicatively Weighted CVD in Section 3.2.2, and in Section 3.2.3 appropriate cluster weights are proposed that can take into account the cluster weight.

3.2.2 Discrete Multiplicatively Weighted CVD

The Voronoi Diagram (VD) concept can also be extended to the *weighted* Voronoi diagrams. In this section we introduce the Multiplicatively Weighted CVD (MWCVD) as an extension of the CVD [CK06]. This concept is the basis of our feature preserving mesh coarsening. First, we review the concepts of a Weighted Voronoi Diagram and Multiplicatively Weighted Voronoi Diagram and finally we give a discrete formulation of the MWCVD.

Weighted Voronoi Diagrams:

Weighted Voronoi diagrams are well known in 2D (see for example [OBS92]). For a given set of n different sites $\{\mathbf{z}_i\}_{i=0}^{k-1}$ in the domain Ω, the weighted Voronoi-region is defined as:

$$D_i^w = \{\mathbf{x} \in \Omega |\ d_i(\mathbf{x}, \mathbf{z}_i) < d_j(\mathbf{x}, \mathbf{z}_j)\ \forall\ j \neq i\} \tag{3.18}$$

where d_i is a weighted distance measure for cluster i. For this type of VD the assumption is that the site has a predetermined weight which reflects an application-specific property, e.g. additive, multiplicative or others, see [OBS92] for an overview.

For an ordinary VD one assumes that all clusters have the same weight, i.e. $d_i(\mathbf{x}, \mathbf{z}_i) \equiv d(\mathbf{x}, \mathbf{z}_i)$, where d is the standard Euclidean distance.

Multiplicatively Weighted Voronoi Diagram:

For this type of weighted Voronoi Diagrams (see [AE84], [OBS92]) d_i is given by:

$$d_i(\mathbf{x}, \mathbf{z}_i) = w_i \|\mathbf{x} - \mathbf{z}_i\| \tag{3.19}$$

where $\{w_i\}_{i=0}^{n-1}$ are predetermined positive weights. In this case, we refer to D_i^w as *multiplicatively weighted (MW) Voronoi-region* D_i^{mw} and to the set $\{D_i^{mw}\}_{i=0}^{k-1}$ as *MW Voronoi diagram.*

Here it is important to note that this definition of the MW Voronoi Diagram is a special case of an anisotropic Voronoi Diagram defined in [LS03], see for more discussions [DW05].

Generally, MW-Voronoi regions are not necessarily connected or convex. In some situations, depending on the associated weights, they may also contain holes, i.e. they may have a non-zero genus.

Multiplicatively Weighted CVD (MWCVD):

Given a set of k different seeds $\{\mathbf{z}_i\}_{i=0}^{k-1}$ together with predetermined positive weights $\{w_i\}_{i=0}^{k-1}$ and a positive density function $\rho(\mathbf{x})$ in the 2D domain Ω. A *Multiplicatively Weighted Centroidal Voronoi Diagram* is a MW-Voronoi diagram for which the seeds $\{\mathbf{z}_i\}_{i=0}^{k-1}$ of the regions $\{D_i^{mw}\}_{i=0}^{k-1}$ are their corresponding centroids (Eq. (3.2)).

Similar to the CVD the following property holds:

PROPOSITION 3.1. *Given a set of k different seeds $\{\mathbf{z}_i\}_{i=0}^{k-1}$ with associated positive weights $\{w_i\}_{i=0}^{k-1}$ and a density function $\rho(\mathbf{x})$ in the domain Ω. Let $\{D_i^{mw}\}_{i=0}^{k-1}$ denote any tessellation of Ω into k regions. Define:*

$$E^{mw} = \sum_{i=0}^{k-1} \int_{D_i^{mw}} \rho(\mathbf{x})\ w_i \|\mathbf{x} - \mathbf{z}_i\|^2 d\mathbf{x} \tag{3.20}$$

E^{mw} is minimized if and only if $\{D_i^{mw}\}_{i=0}^{k-1}$ is a MWCVD.

The proof of this Proposition is given in Appendix A.

Observe that in the case of uniform weights $\{w_i\}_{i=0}^{k-1}$ the MWCVD becomes an ordinary CVD, so that the MWCVD is a generalization of the CVD.

Discrete MWCVD:

Because we work on a polygonal mesh the discrete formulation for MWCVD is required. Using the same assumptions as in Section 3.1.3 the MWCVD energy functional (3.20) becomes:

$$E_{MWCVD} = \sum_{i=0}^{k-1} \Big(\sum_{F_j \in C_i} \rho_j A_j w_i \|\boldsymbol{\gamma}_j - \overline{\boldsymbol{\gamma}}_i\|^2 \Big) \tag{3.21}$$

with the seed of each cluster C_i chosen as its centroid

$$\overline{\boldsymbol{\gamma}}_i = \frac{\sum_{F_j \in C_i} \rho_j A_j \boldsymbol{\gamma}_j}{\sum_{F_j \in C_i} \rho_j A_j}. \tag{3.22}$$

Here A_j and ρ_j are the area and the density function of the face F_j, respectively, and $\boldsymbol{\gamma}_j$ is the triangle's center of mass.

Eq. (3.21) together with Eq. (3.22) can be simplified to:

$$E_{MWCVD} = \sum_{i=0}^{k-1} w_i \Big(\sum_{F_j \in C_i} \rho_j A_j \|\boldsymbol{\gamma}_j\|^2 - \frac{\| \sum_{F_j \in C_i} \rho_j A_j \boldsymbol{\gamma}_j \|^2}{\sum_{F_j \in C_i} \rho_j A_j} \Big) \tag{3.23}$$

As in Section 3.1.3, the energy computation is based only on three values for each cluster: $\sum \rho_j A_j \|\boldsymbol{\gamma}_j\|^2$, $\sum \rho_j A_j \boldsymbol{\gamma}_j$ and $\sum \rho_j A_j$. The ${}^*E^0_{MWCVD}$ is given by:

$$
{}^*E^0_{MWCVD} = w_q(\sum_{F_j \in C_q} \rho_j A_j \|\boldsymbol{\gamma}_j\|^2 - \frac{\| \sum_{F_j \in C_q} \rho_j A_j \boldsymbol{\gamma}_j \|^2}{\sum_{F_j \in C_q} \rho_j A_j}) +
$$
$$
w_p(\sum_{F_j \in C_p} \rho_j A_j \|\boldsymbol{\gamma}_j\|^2 - \frac{\| \sum_{F_j \in C_p} \rho_j A_j \boldsymbol{\gamma}_j \|^2}{\sum_{F_j \in C_p} \rho_j A_j}).
$$

The ${}^*E^1_{MWCVD}$ and ${}^*E^2_{MWCVD}$ can be computed in a similar way, refer to Eq. (3.11)-Eq. (3.13) and Figure 3.15 on page 54.

3.2.3 Feature Preserving Mesh Coarsening

The visual quality of a coarsened model is directly related to the preservation of the surface features. We aim at capturing the mesh features as good as possible and at the same time reducing the overall computational effort. We achieve this through direct usage of the cluster centroid in the clustering process as in the triangulation, without using any specially-designed vertex positioning techniques, e.g. QEM or constrained positioning. We show that with appropriate cluster's weights the MWCVD provides a natural way for an efficient feature-preserving coarsening.

We relate the weights w_i to the face's density ρ_j in the cluster, which itself resembles the mesh features. As the cluster configuration changes during different iterations, the weight will be updated accordingly and thus resulting in differently shaped clusters in the final MWCVD.

We propose three different types of cluster's weight:

Weighted Area:

$$
w_i^A = \sum_{F_j \in C_i} \rho_j A_j \tag{3.24}
$$

The cluster area is weighted with the face densities. This means, clusters with higher density get smaller in size compared to clusters with lower density.

Density Variance:

$$
w_i^V = 1 + \sum_{F_j \in C_i} (\rho_j - \overline{\rho}_i)^2 \tag{3.25}
$$

This will lead to clusters with the same density variance distribution. Thus clusters with higher density will tend to contain only faces with higher density and build around those regions and vice versa.

Maximum Density:

$$
w_i^M = max_{F_j \in C_i}\{\rho_j\} \tag{3.26}
$$

Assigning the maximum density to the cluster weight is based on the idea, that cells with low or high density should be contained in clusters with low or high weight, respectively.

Note that, in the case of a uniform density function our MWCVD will result in an ordinary CVD for the maximum density based and density variance cluster's weight. This is not the case for w^A and could be solved by normalizing w^A with the cluster area $\sum_{F_j \in C_i} A_j$. This, however, yields unsatisfactory clustering results.

A proper reformulation of the cluster weights (Eqs. (3.24)-(3.26)) results in local incremental computation rules which can be efficiently updated and evaluated during clustering process.

For the weighted-area based approach w_i^A (Eq. (3.24)) the value $\sum_{C_j \in V_i} \rho_j A_j$ is already computed for the energy functional Eq. (3.23). Therefore, this cluster weight is obtained with no additional computational cost.

The density-variance approach w_i^V (Eq. (3.25)) can be simplified to

$$w_i^V = 1 + \sum_{F_j \in C_i} \rho_j^2 - (\sum_{F_j \in C_i} \rho_j)^2 / n_i \tag{3.27}$$

where n_i is the number of faces in a cluster. This allows the computation of w_i^V using only local computations based on the additional quantities $\sum_{F_j \in C_i} \rho_j^2$ and $\sum_{F_j \in C_i} \rho_j$ per cluster.

For the maximum-density approach w_i^M (Eq. (3.26)), the current maximum needs to be identified by checking all faces in a cluster. This means keeping explicitly the cluster faces, a requirement that must be avoided for efficiency reasons. To overcome this problem we propose to use the following approximation:

$$max_{F_j \in C_i}\{\rho_j\} = \lim_{p \to \infty} (\sum_{F_j \in C_i} \rho_j^p)^{1/p} \tag{3.28}$$

This approximate value of the maximum ρ in a cluster can be computed using the additional value $\sum_{F_j \in C_i} \rho_j^p$ which again requires only local updates. To get a good approximation of the maximum ρ in a given cluster, the value of p must be large enough. In our implementation we use a value of 10, since larger values may lead to numerical problems.

Figure 3.13 (top) shows different clustering results obtained for the Fandisk model using the proposed cluster's weights. Note that, in all three cases the obtained clustering provides smaller-sized clusters in higher curvature regions where larger-sized ones are covering lower curvature areas. Using the maximum-density or density-variance approach, i.e. w^M or w^V respectively, results in clusters that are elongated along the feature lines, indicating an anisotropic behavior in this case. For the weighted-area based, i.e. w^A, the clusters are more compact and also containing a significant number of cells with low curvature. From this perspective, the clustering results obtained using the former two cluster's weight, i.e. w^M or w^V, are more suitable for our final triangulation, because the cluster centroid is closer to the original surface.

Figure 3.13 (bottom) shows the corresponding triangulations. Although we use only the cluster centroid as a vertex for the triangulation, the final coarse mesh still preserves the main features of the original mesh. Small defects which can be seen in the obtained triangulation are due to the fact that some clusters contain more than one feature point,

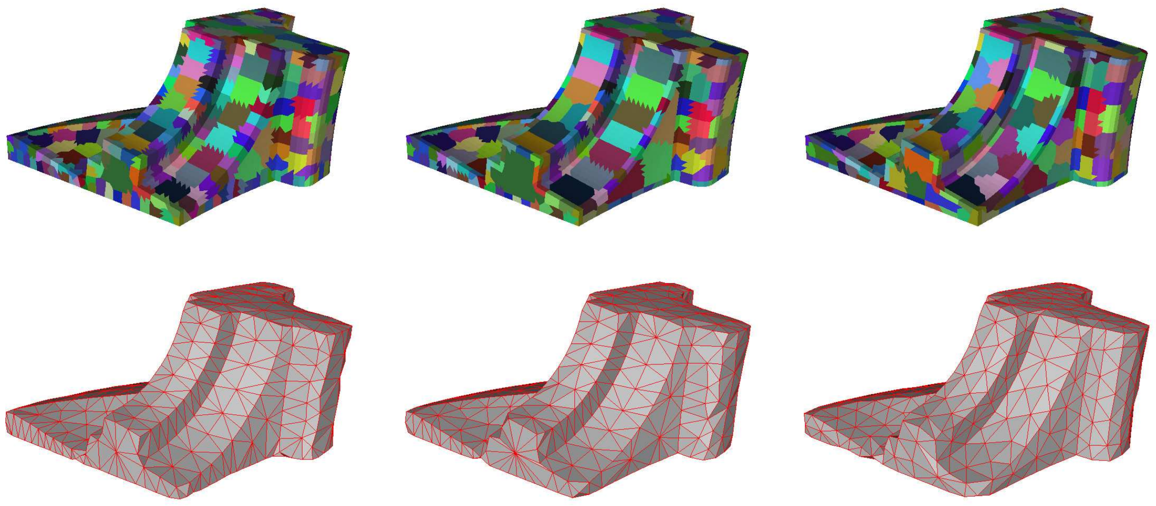


Figure 3.13: (Top) Clustering results and (bottom) corresponding triangulation for the Fandisk model. Left: result using w^A; Center: result using w^M; Right: result using w^V.

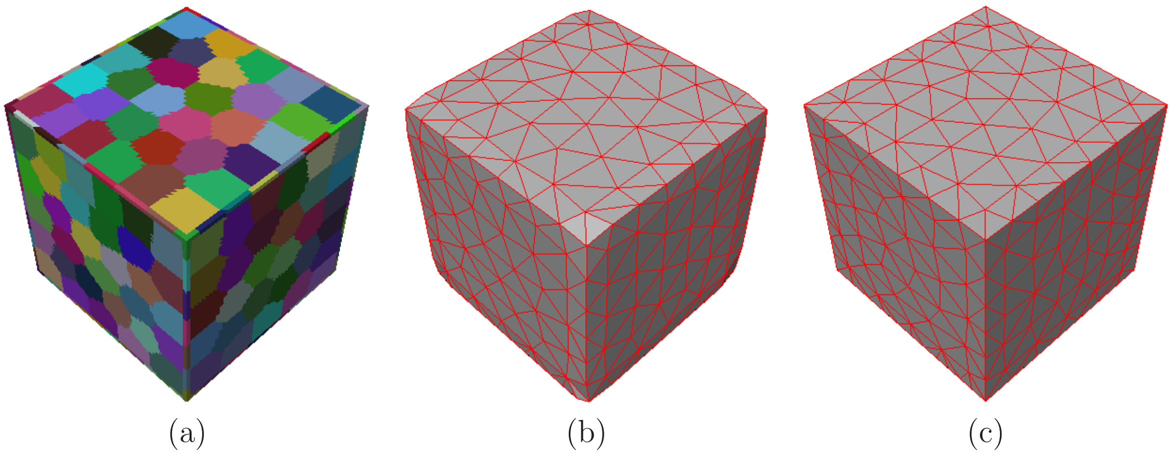


Figure 3.14: (a) Clustering result with w^M, for $l = 7$ resulting in 274 clusters. (b) Triangulation of (a) using the cluster centroid as vertex. (c) Triangulation of (a) using the QEM for vertex placement.

i.e. the intersection of several feature lines. In the final triangulation these feature points are reduced to a single vertex.

Figure 3.14 (a) shows the result of MWCVD utilizing the maximum density w^M cluster weight. The result exhibits the desired result in sense of the feature elongated clusters. Figure 3.14 (b) - (c) provides a comparison of the corresponding triangulations using direct or QEM vertex placement. Although, in the first case only the cluster centroid is used for triangulation, the feature lines are well preserved.

Table 3.1 provides the timing results for different meshes. The first row gives the number

Model		Fandisk	Cube	Bunny	Horse	Armadillo
# V(input mesh)		6475	15002	34834	48485	172974
l (level)		3	7	7	5	5
# seeds (obtained)		527	274	594	1579	4594
Init (msec.)		31	110	281	312	1047
Clustering, w^A	ρ^{new}	50	50	50	100	100
	(msec.)	688	1078	3438	11375	17641
Clustering, w^M	ρ^{new}	50	50	50	100	100
	(msec.)	671	1062	4890	6937	20000
Clustering, w^V	ρ^{new}	3	3	3	5	5
	(msec.)	468	860	3234	5235	18750

Table 3.1: Timing for different input meshes. The results are generated using a 3GHz Intel Core(TM)2 Duo CPU PC.

of vertices of the input mesh. The second one shows a chosen value for level l. For a given value of l, the l-neighborhood initialization gives the total number of seeds k, i.e. the number of vertices in the final coarse mesh. The fourth row shows the time needed to perform l-neighborhood initialization. For different cluster's weights, the scaling parameter ρ^{new} and the time required to obtain the final clustering are presented.

In the case of the density-variance approach the value for the scaling parameter ρ^{new} is usually chosen to be small compared to the value used for the weighted-area and maximum-density approach, see Table 3.1. This is due to the quadratic influence of the density function ρ on the cluster weight w^V, see Eq. 3.25.

3.3 Energy Minimization by Local Optimization

Chiosa and Kolb [CK08] have pointed out that the Valette algorithm, Section 3.1.3, can be employed to perform other tasks, i.e. used with other energy functionals, and not only for building a CVD as originally proposed in [VC04]. As an example we have shown that the $L^{2,1}$ norm [CSAD04] can be represented in an incremental form, which allows an efficient energy computation, we describe this in Section 3.3.2. Thus, in the next section we present a generalization of the Valette algorithm, i.e. Energy Minimization by Local Optimization (EMLO) algorithm, and in Section 3.3.2 we show different application areas of the approach.

3.3.1 The EMLO Algorithm

To facilitate the description of the algorithm, it is helpful to introduce the notion of a Boundary Loop [CK08]:

Definition 3.1. *A* Boundary Loop (BL) *is a closed sequence of all* boundary *half edges of a 1-connected set of faces.*

In Figure 3.15 the BL is represented by a dashed line. If a cluster C_i has more than one component, then each component will have its own boundary loop. Any change in the cluster configuration changes the BLs of the affected clusters, see Figure 3.15. This notion is very important in this as in the context of defining and identifying an Optimal Dual Edge, see Section 4.2.2.

Now, suppose that an *energy functional* E is provided:

$$E = \sum_{i \in \{0,\ldots,k-1\}} E_i = \sum_{i \in \{0,\ldots,k-1\}} \sum_{F_j \in C_i} E^i_j. \tag{3.29}$$

E^i_j is the positive semidefinite[4] cost of assigning the face F_j to the cluster C_i. Note, that in general $E^l_j \neq E^m_j$ if $l \neq m$, e.g. see the CVD energy functional Eq. (3.5). The value of E in Eq. (3.29) depends *only* on a given clustering of the mesh M into k clusters C_i, where each cluster C_i contains a set of faces $\{F^i_j\}$.

The basic idea of the algorithm is that the total energy E can be minimized by simply reassigning the cluster's faces to other clusters in such a way that the energy E decreases, i.e. **energy minimization by reclustering**.

A simple example of this process for two adjacent clusters C_q and C_p and two neighboring faces F_m and F_n for a boundary edge e is presented in Figure 3.15. The energies of three configurations are computed:

- For the initial configuration with corresponding energy E^0 the face F_m belongs to C_q and F_n belongs to C_p.
- For the case when cluster C_q grows and C_p shrinks, thus both F_m and F_n belong to C_q.
- For the case when cluster C_p grows and C_q shrinks, thus both F_m and F_n belong to C_p.

For a given boundary edge e the smallest energy E is chosen and a corresponding configuration is updated, i.e. a cluster grows or shrinks or no configuration change is performed [VC04], [CK08].

Given an initial (starting) configuration the algorithm iteratively reduces the energy E of a given configuration as described in the Algorithm 3.3.

Algorithm 3.3. *(Energy Minimization by Local Optimization (EMLO))*

```
Loop until no configuration changes {
 Forall Clusters $C_i$
   Forall Boundary Loops $b$ of $C_i$
     Loop over boundary edges $e \in b$ {
       Compute energies $E^0$, $E^1$, $E^2$
       Choose smallest energy and update cluster configuration
     }
}
```

[4]A function f is positive semidefinite, if $f(x) \geq 0$ for every $\mathbf{x} \in \mathbf{D}$.

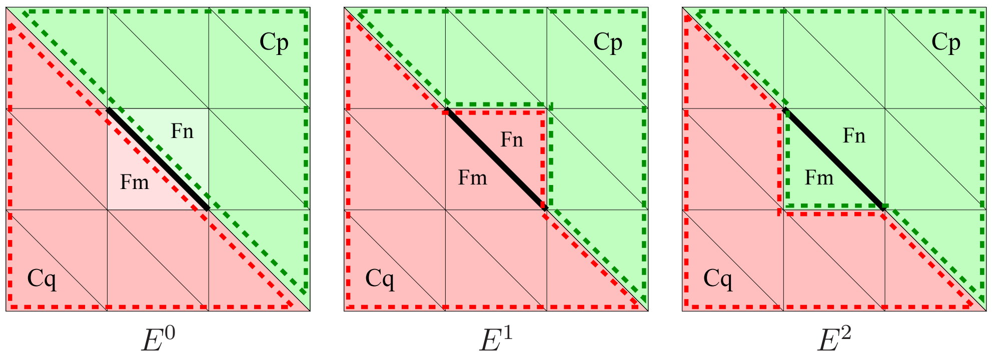


Figure 3.15: Local tests performed for a given boundary edge e (solid line) of two adjacent clusters C_q and C_p where corresponding faces F_m and F_n are checked. The **Boundary Loop** of the cluster is represented by a dashed line. E^0 configuration energy: F_m still belongs to C_q and F_n still belongs to C_p. E^1 configuration energy: C_q grows and C_p shrinks, i.e. F_m and F_n belong to C_q. E^2 configuration energy: C_q shrinks and C_p grows, i.e. F_m and F_n belong to C_p.

In each iteration, the algorithm loops over the BLs of each cluster. For a given boundary edge, energies E^0, E^1, E^2 are computed, refer to Figure 3.15. The cluster configuration is changed and updated only if E^1 or E^2 is smaller than E^0 otherwise the configuration is left intact.

The energies E^0, E^1 and E^2 are defined as follows, refer to Figure 3.15:

$$E^0 = \sum_{i'} E_{i'} + \sum_{F_j \in C_q} E_j^q + \sum_{F_j \in C_p} E_j^p. \tag{3.30}$$

$$E^1 = \sum_{i'} E_{i'} + \sum_{F_j \in C_q \cup \{F_n\}} E_j^q + \sum_{F_j \in C_p \setminus \{F_n\}} E_j^p. \tag{3.31}$$

$$E^2 = \sum_{i'} E_{i'} + \sum_{F_j \in C_q \setminus \{F_m\}} E_j^q + \sum_{F_j \in C_p \cup \{F_m\}} E_j^p. \tag{3.32}$$

with $i^{'} \in \{0, \ldots, k-1\} \setminus \{q, p\}$

Because the energy functional E is supposed to be positive semi-definite and any modification in the cluster's configuration reduces E, the algorithm converges, i.e. there are no cluster configurations for which E^1 or E^2 is smaller than E^0. The result is an optimized clustering for which the functional E is minimal. However, as for all algorithms from this class of iterative approaches, there is no guarantee that the global minimum will be reached.

It is important to observe that for comparing the energies E^0, E^1 and E^2 in the above

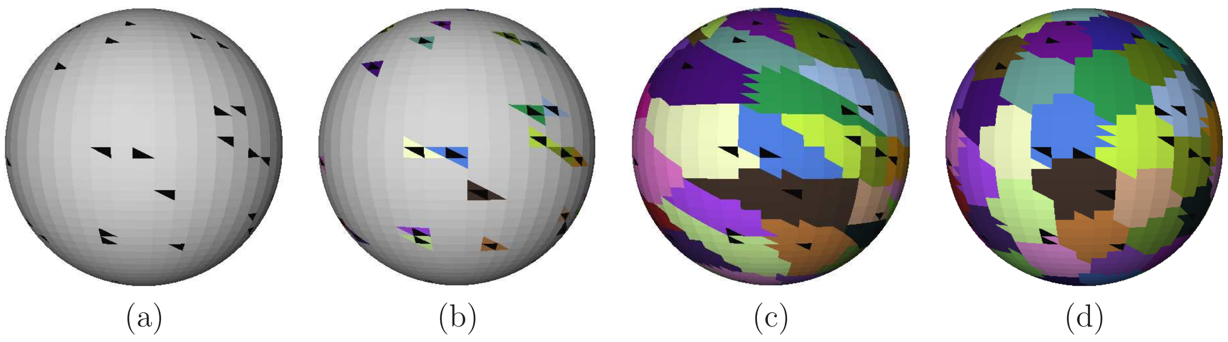


Figure 3.16: EMLO algorithm stages: (a) Initialization, black triangles represent the starting seeds F_j^{start}. (b) Result after one loop of the initial cluster growing. (c) Final result of the initial cluster growing. (d) Final clustering obtained after the energy minimization.

formulas, the first sums are irrelevant, thus Eqs. (3.30)-(3.32) can be reduced to:

$$^*E^0 = \sum_{F_j \in C_q} E_j^q + \sum_{F_j \in C_p} E_j^p. \tag{3.33}$$

$$^*E^1 = \sum_{F_j \in C_q \cup \{F_n\}} E_j^q + \sum_{F_j \in C_p \setminus \{F_n\}} E_j^p. \tag{3.34}$$

$$^*E^2 = \sum_{F_j \in C_q \setminus \{F_m\}} E_j^q + \sum_{F_j \in C_p \cup \{F_m\}} E_j^p. \tag{3.35}$$

Note that, comparing $^*E$ in Eqs. (3.33)-(3.35) is equivalent to comparing the energies E in Eqs. (3.30)-(3.32), but at lower computational cost because the irrelevant summations are dropped.

As a result, comparing the total energy for different configurations requires *only* the data from two adjacent clusters which border a given boundary edge. Thus the energy minimization is done by applying only *local* (thus the name Energy Minimization by Local Optimization) queries on the BL. Based *only* on this information the cluster configuration is optimized, i.e. functional E is minimized.

In general, any energy functional can be used with the EMLO algorithm. However, as ca be seen from Eqs. (3.33)-(3.35), to compute efficiently the total energy for different configurations a direct cluster's data update is required. This can be efficiently achieved only if the energy functionals are in an ***incremental energy formulation***. As an example, the CVD energy functional used in Sections 3.1 - 3.2 has such a formulation. For other energy functionals, such as the ones used in [AFS06], the requirement is store the cluster's faces explicitly and solve an eigensystem to compute the energies for each step. Using these energy functionals is, in general, possible but very inefficient.

Finally the steps of the generalized Valette approach can be summarized as follows:

Initialization: At the beginning each face F_j of a mesh is set to be free F_j^{free}, i.e. not assigned to any cluster. Now, given an initial number of clusters k, identify k starting seeds (faces) F_j^{start}. Assign each F_j^{start} to an individual cluster C_i. Usually a random initialization is used in this step, see Figure 3.16 (a) for an example.

Initial Cluster Grow: Grow each cluster C_i one at a time, by looping over the BLs of the cluster. For each boundary edge e of the cluster C_i check if it has a free face F_j^{free} and assign it to the cluster C_i, i.e. $F_j^{free} \Rightarrow F_j^i$, for an example see Figure 3.16 (b). This process is repeated until the entire model is covered (Figure 3.16 (c)). In this step no energy computation is in general required, thus a very fast growing can be achieved.

Energy Minimization: For this step the Energy Minimization by Local Optimization is used as described in the Algorithm 3.3. The result of the energy minimization is a configuration for which no further decrease in the total energy E is possible, see Figure 3.16 (d).

Taking this algorithm by itself, it requires the initial k number and the positions of the initial seeds to be specified, just as Variational Clustering (Section 2.1.2). Thus, the convergence and the final result is also highly dependent on the starting configuration.

However, compared to the standard Variational Clustering this approach, see Algorithm 3.3, reveals some very important aspects:

1. There is no Priority Queue used in the clustering process.

2. There is no initial seed initialization in each iteration. Remember that for this an explicit check of the faces for each cluster is done to identify the one with smallest energy as starting seed, see Section 2.1.2.

These are the major reasons that make this algorithm so attractive.

3.3.2 Different Application Areas

As pointed out in the last section the EMLO algorithm requires the energy functionals to be in an *incremental formulation*. The advantages of an incremental energy formulation can be easily recognized:

- Efficient cluster's data update, yielding an efficient energy computation.

- Only local queries are required to compute the energy.

- This kind of representation allows an energy simplification as obtained in Eq. (3.14)-(3.15), thus reducing the overall memory requirement and allowing an efficient energy minimization.

However, such a formulation is very hard to find, especially for already existing energy functionals. In this section we try to extend the class of energy functionals which can be used efficiently with the EMLO algorithm, thus showing the generic nature of the approach.

The energy functional proposed in [CSAD04] for planar mesh approximation can be simplified to such a form, as we show in the next subsection.

Thus it is also important to extend the EMLO algorithm to fit, at least, other shapes such as: spheres, cylinders or cones. In this section we give a new incremental energy formulation for sphere fitting. Defining an incremental energy formulation for approximating the mesh with cylinders or cones is still a challenging task. Though, in Chapter 5 we introduce the Boundary-based clustering algorithm, which can be seen as a counterpart of the EMLO algorithm. In that context the requirement on the energy formulation is more relaxed and can accept existing energy formulation.

Planar Approximation:

In [CSAD04] a new shape metric $L^{2,1}$ was introduced, which in the discrete case can be written as:

$$L^{2,1}(C_i) = \sum_{F_j \in C_i} \rho_j A_j \|\mathbf{n}_j - \overline{\mathbf{N}_i}\|^2. \tag{3.36}$$

where $\mathbf{n}_j$ is the normal of the faces F_j and $\overline{\mathbf{N}_i}$ is the representative cluster normal computed as:

$$\overline{\mathbf{N}_i} = \frac{\sum_{F_j \in C_i} \rho_j A_j \mathbf{n}_j}{\| \sum_{F_j \in C_i} \rho_j A_j \mathbf{n}_j \|}. \tag{3.37}$$

In the following, we derive a very efficient formulation from combining Eq. (3.36) and Eq. (3.37). This can be used not only with the EMLO algorithm but also with the classical VC approach. One can show that:

$$E_{Planar} = \sum_{i=0}^{k-1} L^{2,1}(C_i) = \sum_{i=0}^{k-1} 2(\sum_{F_j \in C_i} \rho_j A_j - \| \sum_{F_j \in C_i} \rho_j A_j \mathbf{n}_j \|). \tag{3.38}$$

Applying similar arguments as for Eq. (3.14) in Section 3.1.3, the energy of the initial configuration for planar mesh approximation is:

$$^{**}E^0_{Planar} = -\| \sum_{F_j \in C_q} \rho_j A_j \mathbf{n}_j \| - \| \sum_{F_j \in C_p} \rho_j A_j \mathbf{n}_j \|. \tag{3.39}$$

$^{**}E^1_{Planar}$ and $^{**}E^2_{Planar}$ are computed in a similar way, see Eq. (3.15), (3.16).

The functional E^{Planar} with reduced energies $^{**}E^0_{Planar}$, $^{**}E^1_{Planar}$ and $^{**}E^2_{Planar}$ provides the same advantages as the CVD energy functional. For each cluster only the value $\sum \rho_j A_j \mathbf{n}_j$ needs to be stored. Again, these values can be updated easily and thus a fast energy computation is possible.

Figure 3.17 (a) - (b) shows a clustering result using the $^{**}E_{Planar}$ energy functional. It can be seen that the algorithm performs well. Although, two problems can be observed:

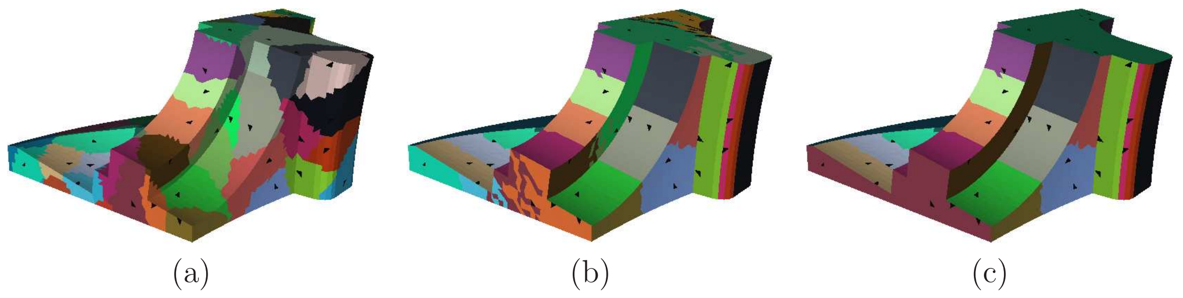


Figure 3.17: Results for planar clustering of the Fandisk model. (a) Random initialization with initial grow for $k = 70$ clusters. (b) Clustering result. (c) Result after merging with $\Delta_{merging} = 1°$, resulting in 44 clusters.

1. The clusters are not compact in strictly planar regions.
2. Some planar patches contain more than one cluster.

The first problem is due to the energy functional, which simply identifies regions with the same representative normal $\overline{\mathbf{N}_i}$. Thus for strict planar regions the decision on whether a face belongs to one or another cluster is generally undefined (zero energy regions). This effect can be even more pronounced if a slight amount of noise is present in the model, thus elongated or branched clusters might appear, see Figure 3.17 (b).

The second problem is due to poor initialization, where for one planar patch more starting seeds are generated.

A possible solution in this case, as proposed in [CSAD04], is to use a sort of a farthest-point initialization (a cluster is inserted one at a time at the face with maximum energy) or the cluster teleportation technique. However this will not pay-off on noisy or smooth models.

Another simple solution could be to use a user-specified parameter $\Delta_{merging}$. If the angle between representative normals $\overline{\mathbf{N}_i}$ of two neighboring clusters is smaller than $\Delta_{merging}$, then the clusters are merged into one cluster. The resulting clustering after applying this step can be seen in Figure 3.17 (c). However, this results in less clusters as originally intended.

Spherical Approximation:

In this section we present a new energy functional, which can faithfully approximate a given surface's geometry with spherical segments.

We assume that for each face F_j the normal $\mathbf{n}_j$ and its centroid $\mathbf{p}_j$ are provided. A sphere can be represented by the sphere center $\overline{\mathbf{C}_i}$ and its radius R_i. Then the sphere-approximating energy functional can be written (see Figure 3.18) as:

$$E_{sphere} = \sum_{i=0}^{k-1} E_i^{sphere} = \sum_{i=0}^{k-1} \sum_{F_j \in C_i} \|\mathbf{c}_j - \overline{\mathbf{C}_i}\|^2. \tag{3.40}$$

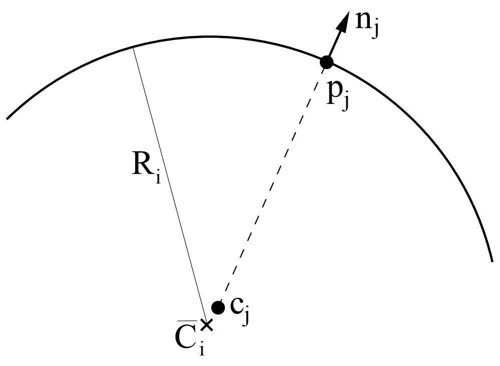


Figure 3.18: Sphere approximation.

where

$$\mathbf{c}_j = \mathbf{p}_j - R_i \mathbf{n}_j. \tag{3.41}$$

is the translated position $\mathbf{p}_j$ along the normal $\mathbf{n}_j$ with the amount R_i, and

$$\overline{\mathbf{C}_i} = \frac{\sum_{F_j \in C_i} \mathbf{c}_j}{k_i}. \tag{3.42}$$

is the cluster centroid using translated position $\mathbf{c}_j$, where k_i is the size of the cluster C_i.
It can be easily shown that Eq. (3.40) can be simplified to:

$$E_i^{sphere} = \sum_{F_j \in C_i} \|\mathbf{c}_j\|^2 - \frac{\| \sum_{F_j \in C_i} \mathbf{c}_j \|^2}{k_i}. \tag{3.43}$$

Using the Eq. (3.41) in Eq. (3.43) one obtains:

$$\begin{aligned} E_i^{sphere} = & \sum_{F_j \in C_i} \|\mathbf{p}_j\|^2 - 2R_i \sum_{F_j \in C_i} (\mathbf{p}_j \cdot \mathbf{n}_j) + k_i R_i^2 - \frac{1}{k_i} \| \sum_{F_j \in C_i} \mathbf{p}_j \|^2 + \\ & \frac{2R_i}{k_i} \left(\sum_{F_j \in C_i} \mathbf{p}_j \cdot \sum_{F_j \in C_i} \mathbf{n}_j \right) - \frac{R_i^2}{k_i} \| \sum_{F_j \in C_i} \mathbf{n}_j \|^2. \end{aligned} \tag{3.44}$$

The value of R_i for which the E_i^{sphere} is minimal can be obtained by taking $\frac{\partial E_i^{sphere}}{\partial R_i} = 0$. Thus R_i is computed as:

$$R_i = \frac{k_i \sum_{F_j \in C_i} (\mathbf{p}_j \cdot \mathbf{n}_j) - \left(\sum_{F_j \in C_i} \mathbf{p}_j \cdot \sum_{F_j \in C_i} \mathbf{n}_j \right)}{k_i^2 - \| \sum_{F_j \in C_i} \mathbf{n}_j \|^2}. \tag{3.45}$$

Using the Eq. (3.45) in Eq. (3.44) one obtains:

$$\begin{aligned}E_i^{sphere} &= \sum_{F_j \in C_i} \|\mathbf{p}_j\|^2 - \frac{1}{k_i}\|\sum_{F_j \in C_i} \mathbf{p}_j\|^2 - \\ &\frac{1}{k_i}\frac{\left[k_i \sum_{F_j\in C_i}(\mathbf{p}_j \cdot \mathbf{n}_j) - \left(\sum_{F_j\in C_i}\mathbf{p}_j \cdot \sum_{F_j\in C_i}\mathbf{n}_j\right)\right]^2}{k_i^2 - \|\sum_{F_j\in C_i}\mathbf{n}_j\|^2}.\end{aligned} \tag{3.46}$$

Applying similar arguments as for Eq. (3.14) in Section 3.1.3, the energy of the initial configuration for spherical cluster approximation is:

$$\begin{aligned}{}^{**}E^0_{sphere} &= -\frac{1}{k_i}\|\sum_{F_j \in C_i} \mathbf{p}_j\|^2 - \\ &\frac{1}{k_i}\frac{\left[k_i \sum_{F_j\in C_i}(\mathbf{p}_j \cdot \mathbf{n}_j) - \left(\sum_{F_j\in C_i}\mathbf{p}_j \cdot \sum_{F_j\in C_i}\mathbf{n}_j\right)\right]^2}{k_i^2 - \|\sum_{F_j\in C_i}\mathbf{n}_j\|^2}.\end{aligned} \tag{3.47}$$

${}^{**}E^1_{sphere}$ and ${}^{**}E^2_{sphere}$ are computed in a similar way, see Eq. (3.15), (3.16).

The functional E_{sphere} with reduced energies ${}^{**}E^0_{sphere}$, ${}^{**}E^1_{sphere}$ and ${}^{**}E^2_{sphere}$ provides the same advantages as the CVD energy functional. For each cluster C_i only the value $\sum \mathbf{p}_j$, $\sum \mathbf{n}_j$ and $\sum(\mathbf{p}_j \cdot \mathbf{n}_j)$ needs to be stored. Again, these values can be updated easily and thus a fast energy computation is possible.

Note in Eq. (3.40) that for planar clusters $R_i = \infty$. This is exactly the case when $k_i^2 - \|\sum_{F_j\in C_i}\mathbf{n}_j\|^2 \approx 0$. Thus, during the clustering process we first check if $|k_i^2 - \|\sum \mathbf{n}_j\|^2| < \Delta_{zero}$ (we use $\Delta_{zero} = 5\times 10^{-5}$). If this is the case we then simply set $E_i^{sphere} = 0$ allowing a planar fitting, otherwise Eq. (3.46) is employed. Or, if the reduced energy from Eq. (3.47) is used, set ${}^{**}E^0_{sphere} = -\infty$ to get planar regions fit first.

Figure 3.19 (a) - (b) shows the result of a sphere approximation for a model which can be ideally approximated by 11 spherical clusters. Figure 3.20 (a) shows the result for a model which consists of a sphere and a cube, which can be ideally approximated by 7 clusters. Note that the newly proposed energy functional performs very well.

However, in contrast to the expected result, as presented in the Figure 3.19 (c) and Figure 3.19 (b) (produced with the ML algorithm; see Chapter4), two problems can be identified:

1. Some spherical patches contain more than one cluster.
2. One cluster contains more than one spherical patch.

Both cases are *only* due to non-optimal starting seeds positioning, i.e. random initialization. As in the case of a planar approximation (see previous section) the teleportation mechanism [CSAD04] can be employed to improve the final clustering. However, the decision on whether two regions needs to be merged is a heuristic one, which simply checks if

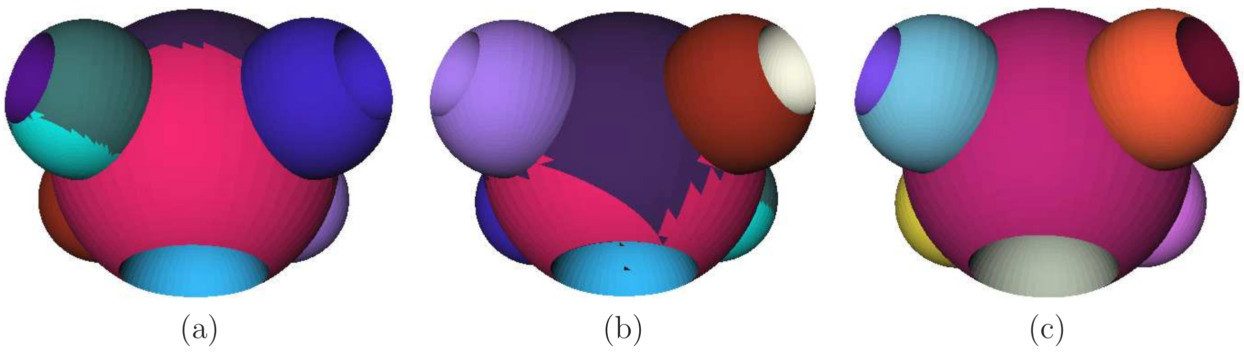


Figure 3.19: Sphere approximation clustering results for 11 clusters. (a)-(b) Result of the EMLO algorithm with random starting seeds. (c) Result of the ML algorithm (see next chapter) for the same number of clusters.

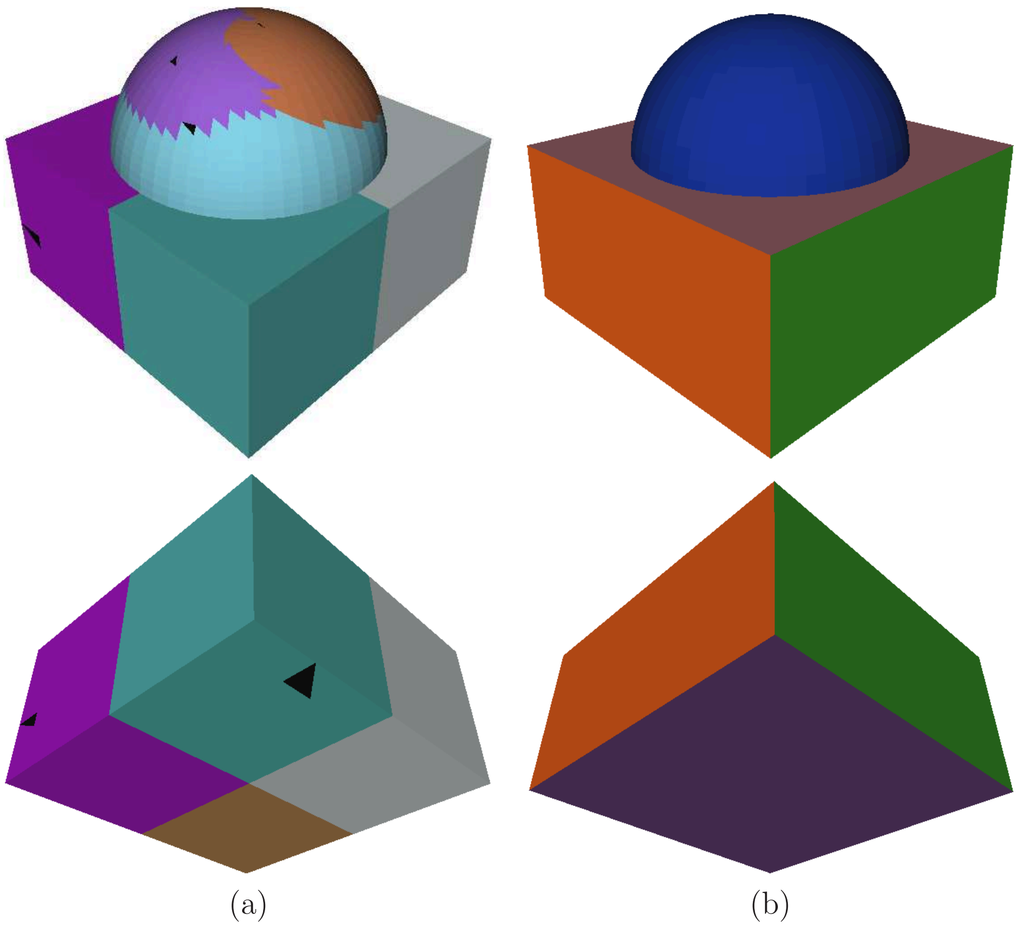


Figure 3.20: Sphere approximation clustering results for 7 clusters. (a) Result of the EMLO algorithm with random starting seeds. (b) Result of the ML algorithm (see next chapter) for the same number of clusters.

the resulting merging error is smaller than half of the error of the worst cluster and this may not pay off if models are noisy. The insertion is also not a cheap operation as the face with worst distortion needs to be identified. Not to say that the user interaction is required if the final number of clusters is not know in advance.

3.4 Conclusions

In this chapter we developed and exploited a new clustering paradigm, namely the EMLO algorithm. The algorithm is proposed as a generalization of the Valette approach [VC04], which was originally proposed only in the context of building a Centroidal Voronoi Diagram on polygonal meshes.

In the first two parts of the chapter the EMLO algorithm was employed for CVD-based mesh coarsening. It was shown that the Multiplicatively Weighted CVD with appropriate cluster weights and direct vertex placement is the best tradeoff between approximation quality and the efficiency of the coarsening process. In the last part the planar and spherical approximation were proposed and applied.

From the obtained results three major problems can be identified:

1. Defining the "true" number of clusters. For the EMLO algorithm, as for all algorithms in this class of iterative techniques, the user needs to define the number of clusters k.

2. The problem of a "good" initialization that can take into account the underlying energy functional. Proposed energy functionals perform very well with the EMLO algorithm, however they are not always able to provide expected results mostly due to the initialization problems. Thus this is still a major issue that needs to be addressed. In the next chapter we try to tackle it together with the first problem.

3. The requirement on the energy functional to be in an incremental energy formulation severely limits the applicability of the EMLO algorithm. However, as already pointed out this can be alleviated using the Boundary-based clustering algorithm, which we propose in Chapter 5.

Chapter 4

Multilevel (ML) Mesh Clustering

In Chapter 2 we described state-of-the-art approaches for mesh clustering. In the last chapter, we have listed and exemplified most of the existing clustering problems regarding the iterative clustering. We also gave different solutions which proved to be efficient for individual problems.

However, it becomes indisputably clear in this context that existing algorithms do not provide all necessary means for controlling or performing the clustering process. Each approach is partially defective and in some cases requires an input from the user or yields suboptimal results.

In this chapter we address these inherent problems of the variational and hierarchical mesh clustering. We describe[1] a novel *Multilevel (ML)* mesh clustering[2] approach [CK08]. The proposed algorithm incorporates the advantages of both methods and consequently overcomes the drawbacks of both. The result is a Multilevel construction, which allows a fast and complete mesh analysis.

Since a brute-force combination of variational and hierarchical methods would be rather expensive regarding the computational effort and storage requirements, an efficient implementation and an optimized discrete data structure is proposed in Section 4.2 and Section 4.3, respectively.

We also show the generic nature of this algorithm in Section 4.4 by applying it to different tasks: Multilevel Centroidal Voronoi Diagram construction, planar and spherical mesh approximation. In this context, it will be recognized that the ML construction can be speeded up without any loss of quality and different variants of the ML clustering are possible in this case, these are discussed in Section 4.5.

4.1 The Multilevel Clustering Algorithm

Let us first review the advantages and the drawbacks of the most frequently employed mesh clustering algorithms: hierarchical and variational.

[1]Parts of this chapter were published in [CK08].

[2]The original name of the algorithm [CK08] was Variational Multilevel Clustering (VMLC).

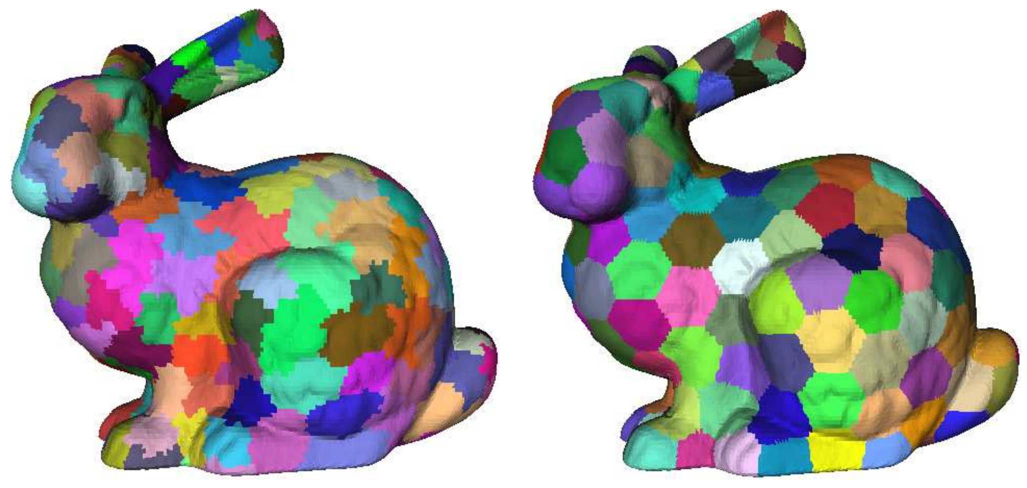


Figure 4.1: CVD result for Bunny model for 200 clusters. (a) Hierarchical clustering. (b) Variational clustering.

Hierarchical method: It produces a hierarchy, i.e. a binary tree, of clusters, see Section 2.1.3. The main advantage of such an approach is that it does not require any additional parameters or intervention from the user.

Despite the simplicity and wide range of applications, hierarchical clustering is yet a greedy (non-optimal) approach. Due to a strict containment property in the cluster hierarchy, it yields worse quality result compared to the variational approaches. If a face is assigned to some cluster during merging it can no longer be reassigned to other clusters although that may result in a more appropriate configuration. This drawback limits the applicability of hierarchical clustering in many situations. E.g. generating a Centroidal Voronoi Diagram (CVD) for mesh coarsening, as exemplified in Figure 4.1. Applying a greedy hierarchical approach on the base of a CVD does not yield a valid solution, i.e. the result is not a CVD.

Variational method: In contrast to the hierarchical approach, this method does provide an optimal clustering[3]. However, an a-priori specified number and position of seeds is required as input, see Section 2.1.2.

The determination of an appropriate or "true" number of clusters involves, in general, the intervention of the user. Still, some applications are steered by quality or approximation criteria thus making a predefinition of the number of clusters impractical. Additionally, as exemplified in the last chapter, it is difficult to choose "good" initial seeds, i.e. starting positions, that can take into account the underling energy functional and starting representatives. This also affects the convergence and the final clustering result. In this context, some results obtained in the last chapter, see Fig-

[3]In the sense that at least a local minimum is reached.

Hierarchical Clustering	**Variational Clustering**
Advantages: • does not require the initial number of clusters k • no initial seed positioning • nested hierarchy and provides all levels (solutions)	**Drawbacks:** • requires the number of clusters k • requires and is dependent on the initialization
Drawback: • a greedy (non-optimal) approach	**Advantage:** • provides an optimized clustering

Table 4.1: Hierarchical mesh clustering vs. Variational mesh clustering.

ure 3.2 on page 34 and Figures 3.19 - 3.20 on page 61, clearly depicts the situations where these problems severely affect the final result.

Table 4.1 summarizes all above described aspects for both clustering algorithms. A simple analysis reveals an interesting fact about these standard and most frequently employed clustering approaches. It can be seen that the advantages of one approach are in fact the drawbacks of the other and vice versa.

Thus, if both algorithms can be fused together, such that they complement each other, a new "hierarchical-variational" approach can be obtained, which incorporates *only* the advantages of both hierarchical and variational algorithms. The new algorithm must obey the following desired proprieties:

- No need to define the number of clusters k.
- No initialization dependency.
- Provides all levels (solutions).
- Each clustering configuration is optimized

This way, it must give to the user the possibility to choose any desired "hierarchy" level, and for each one an optimized solution is obtained. This contrasts the greedy result of a "true" hierarchical clustering. A visual exemplification of this idea is provided in Figure 4.2.

In its simplified form this clustering idea can be realized as a hybrid between hierarchical and variational clustering algorithms. It must alternately perform the merging of two adjacent clusters (hierarchical phase) and apply an optimization step (variational phase). Such a construction is called a *Multilevel (ML)* clustering [CK08].

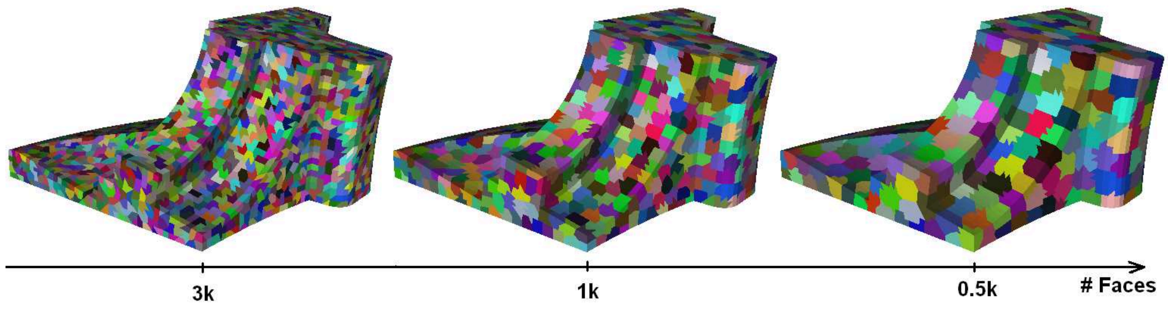


Figure 4.2: Different levels of the ML mesh clustering.

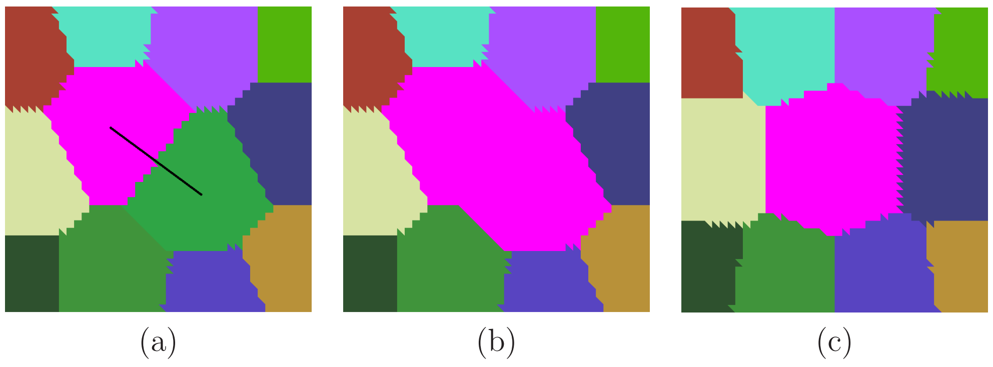


Figure 4.3: An example of a single step of the ML algorithm. (a) Initial configuration, the solid line represents the Dual Edge (DE) to be collapsed. (b) The new configuration after the DE was collapsed. (c) The resulting configuration after the optimization step.

The ML algorithm starts with a configuration where each mesh face F_j is assigned to an individual cluster C_i, i.e the total number of clusters k is equal to the number of faces m in the mesh. In each ML step, two adjacent clusters with smallest merging cost are identified and merged, see Figure 4.3 (a) - (b). Here the total number of clusters is decreased by one. The newly obtained configuration (Figure 4.3 (b)) is then optimized w.r.t. an underlying energy functional, i.e. the total energy is minimized, see Figure 4.3 (c). These general steps of the algorithm are summarized in the Algorithm 4.1.

Algorithm 4.1. *(The Multilevel Clustering Idea)*

```
1 Initialization step.
2 Repeat until $k == 1$ {
3   Merging step.
4   Optimization step.
5 }
```

Thus, after initialization the algorithm alternately executes the merging and the optimization steps until the final number of clusters is equal to one. This way, an optimized multilevel clustering is built.

The cost of merging two clusters C_q and C_p into one representative cluster C_r, i.e. the collapse cost of a Dual Edge which connects C_q and C_p, is defined [CK08] as:

$$Cost_{DE} = E_{C_q \cup C_p} - (E_{C_q} + E_{C_p}). \tag{4.1}$$

where E is the energy of a given cluster C_i.

The cost definition (4.1) is different from the one used for hierarchical clustering in [AFS06]. There, only the cost for the merged clusters, i.e. only the first term in Eq. (4.1), is used. This cost definition, on the contrary, ensures that the collapse operation is done only for clusters which give the smallest increase in the total energy E, i.e. the selected collapse has the least negative impact on the overall energy. Additionally, as we will show in Section 4.4, this cost definition leads to a very compact energy representation and makes the algorithm even more efficient.

The merging of two clusters is always done according to the least merging cost. The following optimization step, see Algorithm 4.1, provides a better initial configuration for the next level. Thus, the initial starting configurations at any level are always the most appropriate for any specific clustering process and obey the underling energy functional. This resolves the inherent problems of the Variational algorithms, for which the result and the convergence is strictly related to the initial selection of seeds.

4.2 Realizing Multilevel Mesh Clustering

An efficient implementation of the Multilevel approach (Algorithm 4.1) is challenging. The direct combination of the hierarchical and variational methods is, in general, difficult, even impossible. There are many reasons for this:

Non-nested hierarchy: The ML mesh clustering algorithm, in contrast to hierarchical face clustering algorithm, does *not*, in general, generate a nested hierarchy. Figure 4.4 provides such an example. Note that the lower level clusters are not completely contained in a single upper level cluster.

The multilevel construction is a nested hierarchy only when no optimization takes place. Thus, navigating between different levels of the multilevel representation is a non trivial task in this case.

Validity and varying number of DEs: The standard hierarchical approach, see Section 2.1.3, starts with the creation of the Dual Graph (DG) of the mesh M. However, after an optimization step some Dual Edges may no longer be valid, see Figure 4.5 (b). This means the DG is no longer valid. These DEs should be removed from their corresponding nodes in the DG. Even worse, new DEs may appear after optimization, see Figure 4.5 (c). Thus, new DEs must be created and added to the corresponding nodes in the DG.

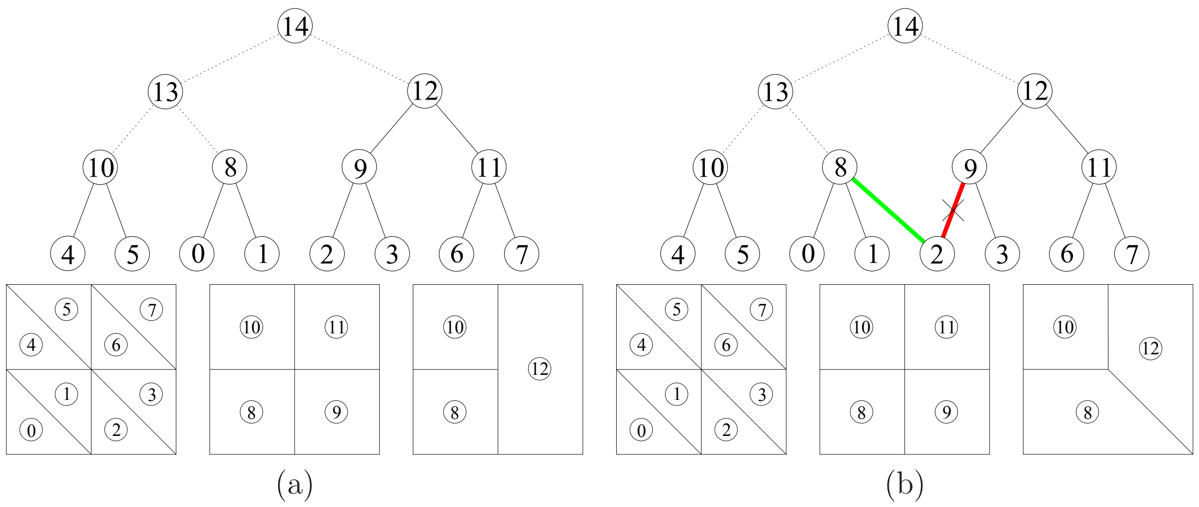


Figure 4.4: Example of a non nested "hierarchy" for ML construction. (a) Hierarchical clustering: hierarchy is nested. (b) ML clustering: merging cluster 9 with cluster 11 and applying optimization destroys the nested hierarchy, i.e. cluster 2 is no longer contained in cluster 9 but belongs to cluster 8.

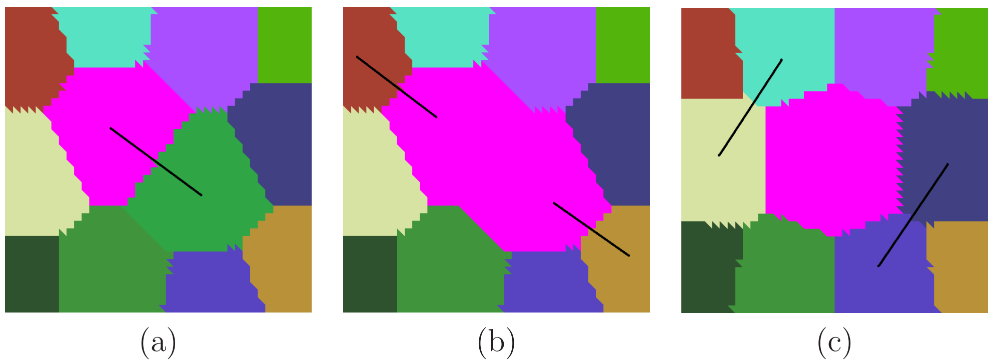


Figure 4.5: Example of an ambiguity in the validity of the DEs. (a) Initial configuration, the solid line represents the Dual Edge (DE) to be collapsed. (b) The new configuration after the DE was collapsed. The solid lines represent DEs which will not be valid after the optimization step. (c) The resulting configuration after the optimization step. The solid lines represent newly created DEs.

This is in contrast to the standard hierarchical algorithm, where only DEs incident to a newly created node need to be updated. All this makes the DG structure rather inefficient and special care must be taken for a correct update.

Computational effort: The variational clustering is an iterative approach where partitioning and fitting steps are alternated until an optimal partitioning according to an energy functional is obtained, see Section 2.1.2. Thus, using variational clustering in conjunction with hierarchical clustering will require, in general, two priority queues: one for all DEs and the second one for the priority of faces in the partitioning step. Hence, using this approach in the optimization step is computationally prohibitive.

Tracking of cluster faces: Applying the optimization step after each dual edge collapse will reassign some of the faces to different clusters, see Figure 4.5 (b) - (c). Thus, all such changes must be tracked and represented in an appropriate data structure for a fast "hierarchy" navigation.

To tackle the first two problems we depart from the standard usage of a Dual Graph and of a Priority Queue to sort the DEs. Instead, in Sections 4.2.1-4.2.2 we propose new and more efficient methods for solving these problems.

The computational effort of the ML approach can be reduced by choosing a more efficient optimization approach compared to the variational clustering. We propose to use the Energy Minimization by Local Optimization approach developed in the last chapter. This technique is more suitable, because it does not require any Priority Queue for performing the optimization.

To address the last problem, in Section 4.3 we propose a new data structure for tracking the cluster faces.

4.2.1 Optimal Dual Edge

A Dual Edge between two clusters C_k and C_l, with respective Boundary Loops BL_k and BL_l, see Definition 3.1, is identified according to the Definition 4.1. Each DE has assigned *cost* according to the Eq. (4.1) and the reference to two merging clusters, see Figure 4.6.

Definition 4.1. *A* dual edge (DE) *between two clusters C_k and C_l can be created if and only if the Boundary Loops BL_k and BL_l share at least one common edge.*

The total number of DEs for a specific cluster equals to the number of different BLs that are adjacent to a given cluster. In general, a cluster must keep track of these DEs and correspondingly update them. However, as already pointed out, during the optimization step the cluster's BL most probably changes, which results in a varying number of DEs, see Figure 4.5. Each of these affected clusters must be checked for possible changes in its DE list. These changes, i.e. deletion or insertion of new DEs, must be propagated and saved in some form, which is rather inefficient.

The major idea here is to store only *one* DE per cluster. We call this DE an *Optimal Dual Edge (ODE)* and define it according to the Definition 4.2, refer to Figure 4.6.

Definition 4.2. *An Optimal Dual Edge (ODE) is a Dual Edge, see Definition 4.1, with the smallest merging cost out of all DEs of a given cluster.*

This notion of an ODE (Definition 4.2) helps us in dealing with the problem of varying number of DEs:

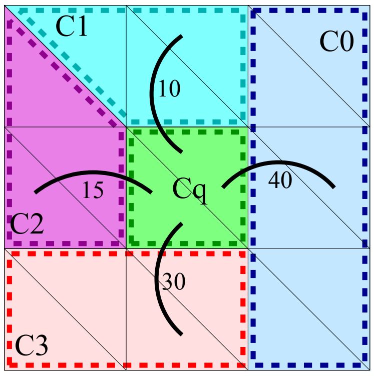


Figure 4.6: Identification of an *Optimal Dual Edge (ODE)* for a cluster C_q. A DE is represented by an arc with corresponding collapse cost. A loop over BL_q is done and all possible DE are checked. The DE with the smallest cost is stored. In this example it is $C_q.ODE.cost = 10$ and $C_q.ODE.opposite = C_1$.

- We no longer track the validity of DEs or the appearance of new DEs during optimization. We rather indicate that there has been a change to the cluster configuration and consequently its ODE is probably no longer valid or the merging cost is not updated, i.e. `C.validODE=false`.

- Only one Dual Edge (in our case it is an optimal DE) needs to be identified for each cluster. Thus, the current number of DEs is exactly equal to the current number of clusters.

The process of identifying an ODE is implemented as the `updateODE()` subroutine. Any cluster which has an invalid ODE, i.e. `C.validODE==false`, requires a new ODE. To identify an ODE for a cluster C_i a loop over the cluster's BL is done. A check of all possible DEs is performed and the *one* with the smallest cost is stored. An example is provided in Figure 4.6.

An ODE, as a DE, keeps the information of two possibly merging clusters. One of them is the current cluster `C` for which the ODE is identified. The second one is some of its adjacent clusters, which is stored in `C.ODE.opposite`. Each ODE has also an associate merging cost stored in `C.ODE.cost`, see Figure 4.6.

4.2.2 Implementing the ML Clustering Process

To keep track of all valid clusters, we propose to use a Cluster Array structure defined according to the Definition 4.3.

Definition 4.3. *A Cluster Array (CA) is an array of a fixed length k that stores the references to all created clusters, see Figure 4.7.*

A Cluster Array, see Figure 4.7, keeps references to all clusters. If a cluster gets invalidated due to some merging operation, then it is simply marked as `invalid`.

C_0	C_1	C_2	C_3	. . .	C_{k-4}	C_{k-3}	C_{k-2}	C_{k-1}

Figure 4.7: The Cluster Array (CA).

Now, each cluster C_i in part has three flags that describe the cluster's state:

- `C.affected` flag is set to `true` if the cluster configuration is affected in some way by a merging or optimization operation.
- `C.needOptimization` flag is set to `true` if the cluster configuration requires optimization due to merging or a change in the cluster configuration.
- `C.validODE` flag is set to `true` if the cluster's ODE is valid and the collapse cost is updated.

Initialization:

Before a ML clustering (Algorithm 4.1) can start it requires an initial starting configuration. In the initialization step each mesh face $F_j \in M$ is assigned to an individual cluster C_i, $i = j$, as shown in Figure 4.8. As a result the total number of clusters is equal to m.

F_0	F_1	F_2	F_3	. . .	F_{k-4}	F_{k-3}	F_{k-2}	F_{k-1}
				$\Downarrow$				
C_0	C_1	C_2	C_3	. . .	C_{k-4}	C_{k-3}	C_{k-2}	C_{k-1}

Figure 4.8: The Cluster Array (CA) after initialization.

At this step no ODEs are identified. We only flag each cluster in the CA as being unchanged `C.affected=false` and `C.needOptimization=false`, i.e. the cluster configuration has not changed and it does not require optimization. However, we set the cluster's ODE as being non-updated `C.validODE=false`, i.e. the ODE of this cluster is no longer determined.

Dual Edge Collapse:

At this step an ODE with minimal cost must be identified in the CA and the collapse operation applied to this ODE.

To efficiently identify a cluster with the smallest cost regarding its ODE out of all ODEs, we apply a *single* step of a *bubble sort* to the CA to move this cluster to the end of the CA, see Figure 4.9. If a cluster has an invalid ODE, i.e. `C.validODE==false`, then an ODE is identified for this cluster using the `updateODE()` subroutine, see Section 4.2.1. The

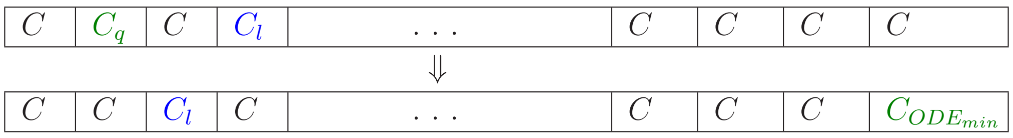

C C_q C C_l . . . C C C C
⇓
C C C_l C . . . C C C $C_{ODE_{min}}$

Figure 4.9: Applying a single step of *bubble sort* to the CA will move the cluster with smallest cost ODE to the end of CA. In this case $C_{ODE_{min}}.ODE.opposite = C_l$.

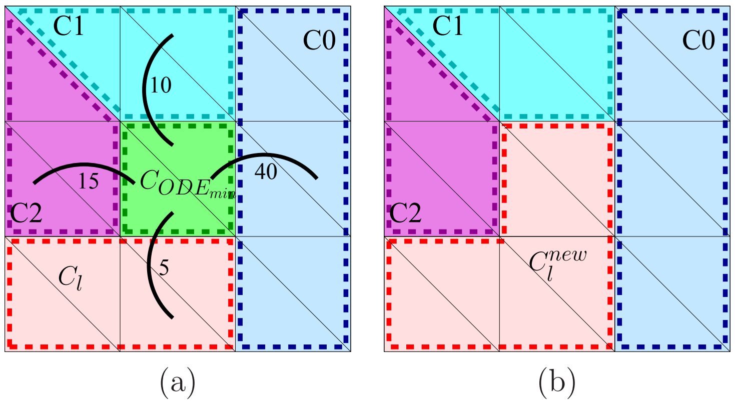


Figure 4.10: Example of a Dual Edge collapse. (a) Initial configuration. The $C_{ODE_{min}}$ has smallest cost ODE. (b) Collapsing the ODE. Reassigns cluster's data and faces to the cluster C^l_{new}.

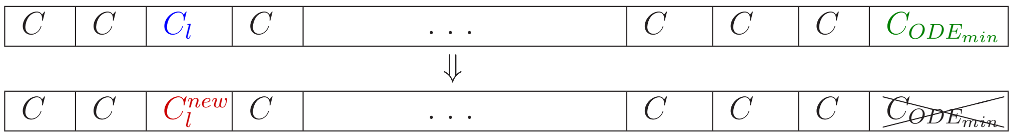

C C C_l C . . . C C C $C_{ODE_{min}}$
⇓
C C C_l^{new} C . . . C C C $\cancel{C_{ODE_{min}}}$

Figure 4.11: Merging cluster $C_{ODE_{min}}$ with C_l will result in a new cluster C_l^{new}. Where the last cluster in the CA automatically becomes invalid by decreasing the number of clusters by one.

merging cost of the ODE is then used to decide if a given cluster has the smallest merging cost[4].

Note that using the CA with all clusters ODEs and a single step of the bubble sort is an efficient solution to identify the minimal cost DE. Using a Priority Queue will result in $O(n \log n)$ complexity, because the PQ needs to be rebuilt after each optimization phase which vary the number of DEs. This is in contrast to hierarchical clustering where the number of PQ elements is fixed.

A Dual Edge *collapse operation* is applied to the ODE of the last valid cluster in the

[4]In the case of two clusters having the same merging cost, the cluster with smallest number of faces must be in general promoted. This reduces the number of operations in the merging step and the information stored for the multilevel representation.

CA, i.e. to the $C_{ODE_{min}}$, see Figure 4.9. Collapsing an ODE means merging two adjacent clusters into one representative cluster. The merging is applied between $C_{ODE_{min}}$ and the opposite cluster that makes up the ODE, e.g. in Figures 4.10-4.11 the cluster C_l is the opposite cluster of ODE_{min}.

Figure 4.10 provides an example of a collapse operation, while Figure 4.11 shows how these changes are propagated to the CA. In the merging operation the cluster's data and faces are reassigned to a new representative cluster C^l_{new}. After this operation the last cluster $C_{ODE_{min}}$ in CA is invalidated, by reducing the number of valid clusters by one, as shown in Figure 4.11. Because the configuration of C^l_{new} cluster has changed we set its identifier C^l_{new}`.affected` to `true` and C^l_{new}`.needOptimization` to `true`, which indicates for the next step that only this cluster requires optimization.

Optimization:

The EMLO algorithm (Algorithm 3.3) is used to perform the optimization. The optimization is executed only for clusters that require optimization, i.e. which are flagged as `C.needOptimization==true`.

After performing the optimization we indicate that a given cluster does not require any further optimization by resetting its flag to `C.needOptimization=false`.

Any change in the cluster configuration during energy minimization flags the opposite cluster as requiring optimization, i.e. `C.needOptimization=true`. Thus, in the next optimization loop these clusters are also handled. This process repeats until no cluster configuration change occurs, i.e. all clusters have `C.needOptimization=false`.

Observe that the energy minimization always starts with the cluster most affected by the collapse operation, i.e. the cluster optimization is always applied to the merged cluster first, followed by an outwards propagation of that change to the neighboring clusters. This will lead to a better convergence behavior of the Multilevel approach.

After the optimization step any cluster that has been affected by the optimization will have a flag `C.affected=true`. Each of these clusters and its neighbors most probably have an invalid ODE, thus we reset the flag for all of them to `C.validODE=false`. This triggers a new identification of ODEs for these clusters and leads to a correct identification of ODEs for neighboring clusters as well.

4.3 Multilevel Data Structure

In this section we describe a very fast and memory efficient differential data structure to store the multilevel representation. It provides an easy way for reconstructing any level and for computing any additionally required cluster information. Although here we describe this structure only in the context of mesh clustering, the same data structure can be used to represent any multilevel construction, e.g. Multilevel data clustering described in Chapter 6.

Suppose that we are given an initial mesh M as presented in Figure 4.12 (step0). Each mesh face $F_j \in M$ has an index `ID=j`. At the beginning each F_j (see Section 4.2.2) is assigned to a corresponding cluster C_i with `ID=i=j`.

After each optimization step (or merging step if the optimization is deactivated) different faces are exchanged between different clusters. For keeping track of these exchanges the basic idea is to store *only* the information corresponding to the faces which have moved from one cluster to another, i.e the difference between two consecutive clusterings.

Thus, for each such face F_j^{moved} with `ID=a` we store a triplet `(a, b, c)`, where:

`a`: Is the face index.

`b`: Is the `ID` of the cluster to which the face has moved.

`c`: Is the `ID` of the cluster from which the face has been removed.

Figure 4.12 shows an example of the differential data storage; here only the final situation after the complete optimization steps is depicted. Note that even though a face can be moved to different clusters during a complete optimization step, we store only the final cluster ID.

Now, to reconstruct level l from level $l-1$ using the stored information at level l and the differential data $\{(a_1^l, b_1^l, c_1^l), \ldots, (a_n^l, b_n^l, c_n^l)\}$, we change all addressed faces with `ID=`a^l to a cluster with `ID=`b^l. Taking the example in Figure 4.12, going from *step*1 to *step*2 will assign the face with `ID=5` to the cluster with `ID=4`. Vice versa, if going from $l+1$ to l level, for all faces with `ID=`a^{l+1} the cluster ID is changed to `ID=`c^{l+1}. Thus going from *step*4 to *step*3 in Figure 4.12 requires to change for face F_1 its cluster's `ID` to the `ID=1`.

Moreover, the core algorithm does not actually store any intermediate information like energy for any level in the multilevel representation. However, for some applications it may be important to have this kind of information to justify or compare individual cluster levels or to compute the cluster energy.

The energy of each level can be computed using the stored triplets `(a, b, c)`. Suppose that each cluster C_i in the initialization step, i.e. the cluster consists of one face only, has some initial values $C_i^0\{\mathbf{m}_i\}$ for different additive measures $\mathbf{m}_i$, see Table 4.2. The measures $\mathbf{m}_i$ may represent any cluster data such as the centroid γ_i for a CVD, the normal $\mathbf{n}_i$ for planar fitting or the area ρ_j, see Section 4.4. In this case, index a refers to the measure $\mathbf{m}_a$ of the cluster C_a^0. Index b indicates that the measure $\mathbf{m}_a$ should be "inserted" into the cluster C_b and at the same time index c indicates that $\mathbf{m}_a$ should be "removed" from the cluster C_c. "Insertion" and "removal" may include various mathematical operations depending on the measure. For example for the cluster area the insertion operation is a simple addition and the removal operation is a subtraction of the terms. A detailed example which refers to Figure 4.12 is presented in Table 4.2.

It should be noted, that based on this principle arbitrary measures can be tracked throughout the multilevel construction.

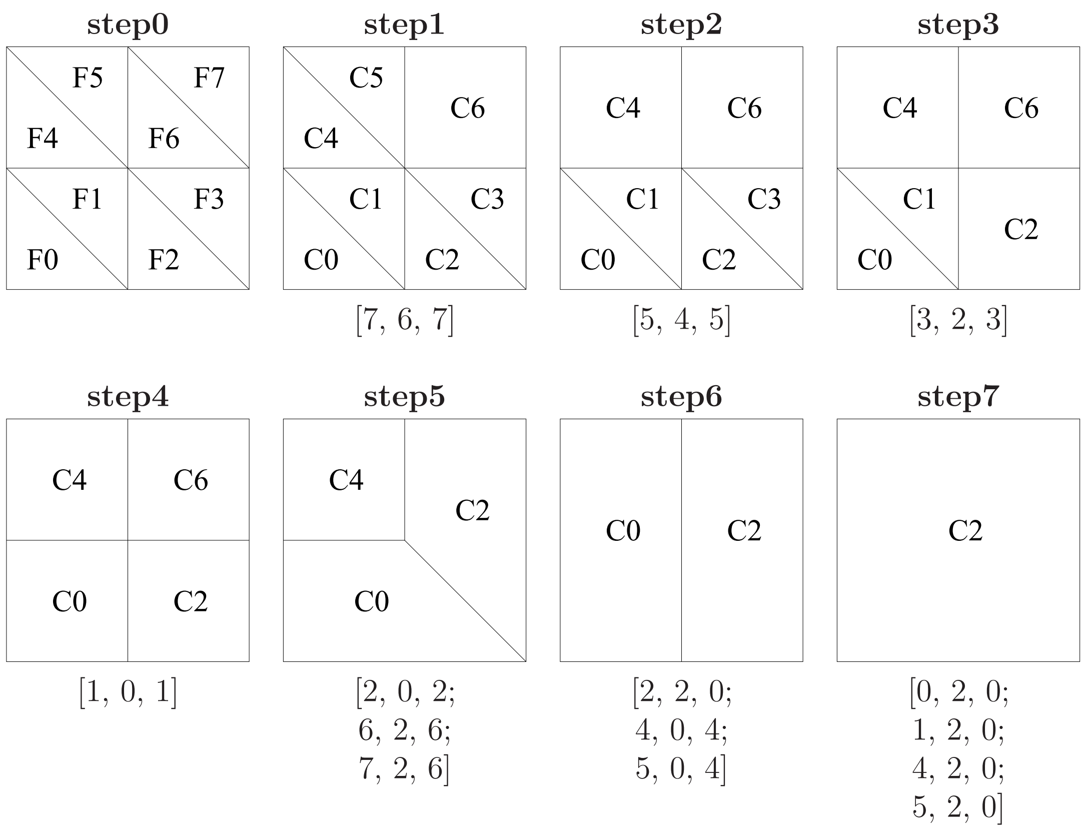


Figure 4.12: An example of stored information for each step of the Multilevel clustering algorithm applied to a simple mesh. *step*0 is the initial configuration. Here the cluster `ID` corresponds to the face `ID`. *step*7 is the final clustering.

*step*0	*step*1	*step*2	*step*3	*step*4	*step*5	*step*6	*step*7
$C_0^0\{m_0\}$	m_0	m_0	m_0	$m_0 \oplus m_1$	$(m_0 \oplus m_1) \oplus m_2$	$(m_0 \oplus m_1 \oplus m_2) \ominus m_2 \oplus m_4 \oplus m_5$	$(m_0 \oplus m_1 \oplus m_4 \oplus m_5) \ominus m_0 \ominus m_1 \ominus m_4 \ominus m_5$
$C_1^0\{m_1\}$	m_1	m_1	m_1	$m_1 \ominus m_1$	$\times$	$\times$	$\times$
$C_2^0\{m_2\}$	m_2	m_2	$m_2 \oplus m_3$	$(m_2 \oplus m_3)$	$(m_2 \oplus m_3) \ominus m_2 \oplus m_6 \oplus m_7$	$(m_3 \oplus m_6 \oplus m_7) \oplus m_2$	$(m_3 \oplus m_6 \oplus m_7 \oplus m_2) \oplus m_0 \oplus m_1 \oplus m_4 \oplus m_5$
$C_3^0\{m_3\}$	m_3	m_3	$m_3 \ominus m_3$	$\times$	$\times$	$\times$	$\times$
$C_4^0\{m_4\}$	m_4	$m_4 \oplus m_5$	$(m_4 \oplus m_5)$	$(m_4 \oplus m_5)$	$(m_4 \oplus m_5)$	$(m_4 \oplus m_5) \ominus m_4 \ominus m_5$	$\times$
$C_5^0\{m_5\}$	m_5	$m_5 \ominus m_5$	$\times$	$\times$	$\times$	$\times$	$\times$
$C_6^0\{m_6\}$	$m_6 \oplus m_7$	$(m_6 \oplus m_7)$	$(m_6 \oplus m_7)$	$(m_6 \oplus m_7)$	$(m_6 \oplus m_7) \ominus m_6 \ominus m_7$	$\times$	$\times$
$C_7^0\{m_7\}$	$m_7 \ominus m_7$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$

Table 4.2: The data refers to the multilevel representation example in Figure 4.12. It shows how cluster data can be obtained for different steps. The operator $\oplus$ indicates an insertion operation and $\ominus$ an removal operation, i.e. a given additive measure $\mathbf{m}$ is inserted or removed from the cluster. The flag $\times$ in the table indicates that the cluster at that step is no longer valid.

4.4 Multilevel Mesh Clustering Results

In this section we evaluate the Multilevel approach. Because the EMLO algorithm (see last chapter) is used in the optimization phase of the Multilevel construction[5], the same energy functionals apply, as presented in Section 3.1.3 and Section 3.3.2.

All the results presented in this chapter are obtained on a 3GHz Intel Core(TM)2 Duo CPU PC.

Building an Approximated Centroidal Voronoi Diagram:

As described in Section 3.1.4 the CVD energy functional can be simplified to [VC04]:

$$E_{CVD} = \sum_{i=0}^{k-1} \Big(\sum_{F_j \in C_i} m_j \|\boldsymbol{\gamma}_j\|^2 - \frac{\| \sum_{F_j \in C_i} m_j \boldsymbol{\gamma}_j \|^2}{\sum_{F_j \in C_i} m_j} \Big).$$

where $\boldsymbol{\gamma}_j$ and $m_j = \rho_j A_j$ is the centroid and the weighted area of the face F_j, respectively.

In case of the EMLO approach the energy of the initial configuration is:

$$^{**}E^0_{CVD} = -\frac{\| \sum_{F_j \in C_q} m_j \boldsymbol{\gamma}_j \|^2}{\sum_{F_j \in C_q} m_j} - \frac{\| \sum_{F_j \in C_p} m_j \boldsymbol{\gamma}_j \|^2}{\sum_{F_j \in C_p} m_j}.$$

$^{**}E^1_{CVD}$ and $^{**}E^2_{CVD}$ are computed in a similar way, refer to Eqs. (3.15)- (3.16).

For the ML approach we need additionally to define the collapse cost of a Dual Edge. According to Eq. (4.1) the CVD merging cost for two merging clusters C_q and C_p is:

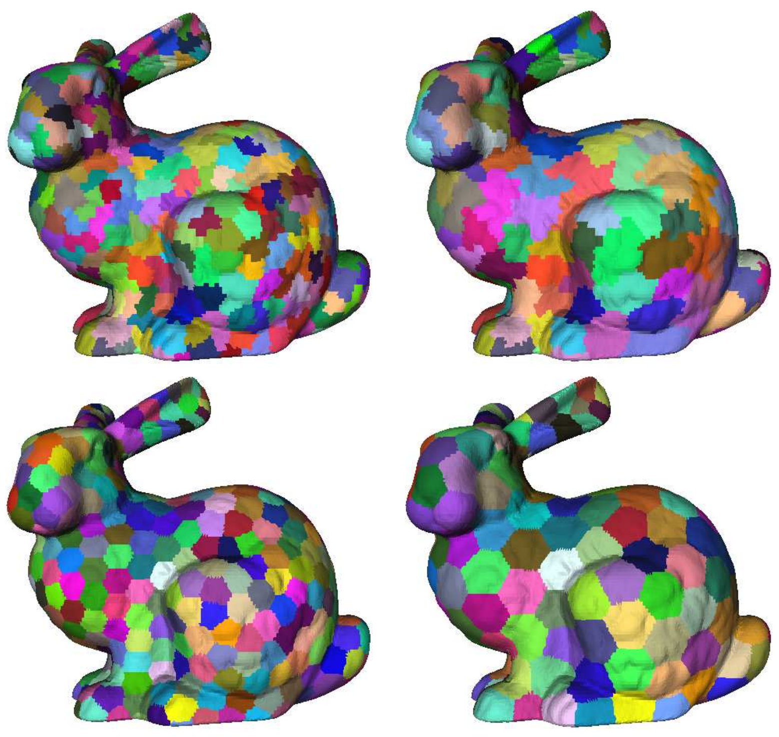
$$\begin{aligned} Cost_{CVD} = & \sum_{F_j \in (C_q \cup C_p)} m_j \|\boldsymbol{\gamma}_j\|^2 - \frac{\| \sum_{F_j \in (C_q \cup C_p)} m_j \boldsymbol{\gamma}_j \|^2}{\sum_{F_j \in (C_q \cup C_p)} m_j} - \\ & \sum_{F_j \in C_q} m_j \|\boldsymbol{\gamma}_j\|^2 + \frac{\| \sum_{F_j \in C_q} m_j \boldsymbol{\gamma}_j \|^2}{\sum_{F_j \in C_q} m_j} - \\ & \sum_{F_j \in C_p} m_j \|\boldsymbol{\gamma}_j\|^2 + \frac{\| \sum_{F_j \in C_p} m_j \boldsymbol{\gamma}_j \|^2}{\sum_{F_j \in C_p} m_j}. \end{aligned} \tag{4.2}$$

Eq. (4.2) simplifies to a very compact form:

$$Cost_{CVD} = -\frac{\| \sum_{F_j \in (C_q \cup C_p)} m_j \boldsymbol{\gamma}_j \|^2}{\sum_{F_j \in (C_q \cup C_p)} m_j} - {}^{**}E^0_{CVD}. \tag{4.3}$$

Observe that this form leads to a very fast and efficient energy/cost computation. For each cluster only the values $\sum m_j \boldsymbol{\gamma}_j$ and $\sum m_j$ need to be stored.

[5]Note, as already discussed, this restriction on the optimization algorithm is only due to high computational effort required by the ML approach.



($level = 500$) ($level = 200$)

Figure 4.13: The CVD clustering results using the HFC algorithm (top) and the ML algorithm (bottom) for the Bunny model for different *levels*, i.e. number of clusters.

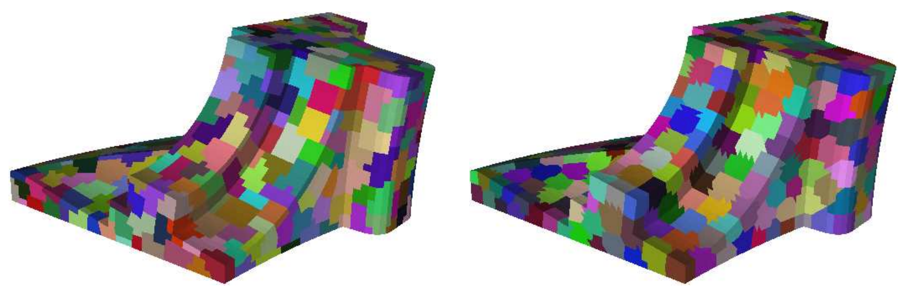


Figure 4.14: The CVD clustering results for the Fandisk model at $level = 450$ using: (left) The HFC algorithm, (right) The ML algorithm.

Figure 4.13 depicts the result of a CVD for the Bunny model using the HFC compared to ML algorithm at the same level. The same comparison is provided in Figure 4.14 for a mechanical Fandisk model. Observe that the ML algorithm provides a valid CVD construction at each level with well shaped clusters. However, applying the HFC algorithm does not yield a valid solution, i.e. the result is not a CVD. Consequently, the energy of the ML is lower than that of the HFC as depicted in Figures 4.15 - 4.16.

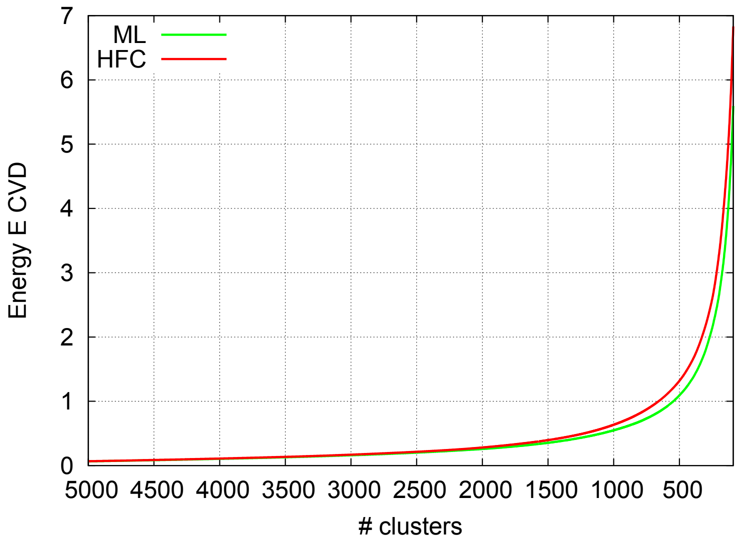


Figure 4.15: CVD Energy versus # of clusters for the Fandisk model. HFC: hierarchical face clustering, ML: multilevel clustering.

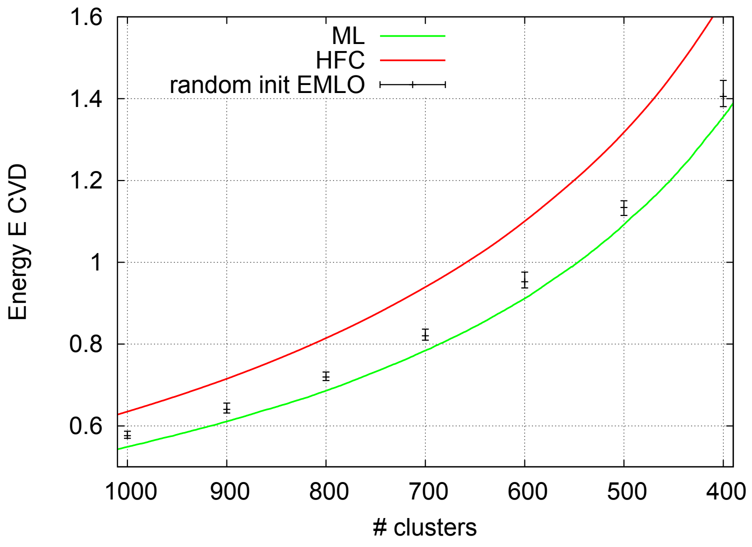


Figure 4.16: CVD Energy versus # of clusters for the Fandisk model. HFC: hierarchical face clustering, ML: multilevel clustering, EMLO: with 50 random initialization for a given number of clusters.

Figure 4.16 additionally depicts the dependency between the total CVD energy (Eq. (3.7)) and the number of clusters for the EMLO algorithm with random initializations. To obtain the energy variation limits, the algorithm was applied 50 times for the same number of clusters with different random seeds. Although the ML uses in its optimization stage the EMLO algorithm the energy obtained with ML is always lower, thus leading to a better quality clustering than that of the EMLO algorithm. This behavior is due to a better initialization configuration provided by the ML algorithm, this in contrast to random initialization of the EMLO algorithm. The same behavior has been observed with similar quality for other models.

It can be seen that the ML algorithm does not require any input from the user. It performs a complete mesh analysis, i.e. a complete set of solutions is obtained, and the user has the possibility to choose any solution. As expected, the results are of higher quality than those obtained with the classical algorithms alone.

To reflect on the question of how many clusters best suit a given CVD clustering, or in other words, what is the best level that needs to be chosen out of all, one could check the energy behavior and look for the so-called *elbow effect* [DGJW06]. However, it can be seen from Figure 4.15 that the energy of a CVD is steadily increasing, and does not indicate clearly the point from which the increase in the energy is much faster. To judge the number of clusters, in this case, one could also check the quality of the resulting triangulation or the approximation error between original mesh and the resulting coarsened mesh, or any other problem specific measures.

Planar Approximation:

As described in Section 3.3.2 the energy of planar mesh approximation for the initial configuration can be written as:

$$^{**}E^0_{Planar} = -\| \sum_{F_j \in C_q} \rho_j A_j \mathbf{n}_j \| - \| \sum_{F_j \in C_p} \rho_j A_j \mathbf{n}_j \|. \tag{4.4}$$

$^{**}E^1_{Planar}$ and $^{**}E^2_{Planar}$ are computed in a similar way, see Eq. (3.15) - (3.16).

The cost of a dual edge is then calculated similar to the Eqs (4.2) - (4.3) as:

$$Cost_{Planar} = -\| \sum_{F_j \in (C_q \cup C_p)} \rho_j A_j \mathbf{n}_j \| - {}^{**}E^0_{Planar}. \tag{4.5}$$

Observe that the functional E^{Planar} provides the same advantages as the CVD energy functional. For each cluster only the value $\sum \rho_j \mathbf{n}_j$ needs to be stored for a fast energy and merging cost computation.

Figure 4.17 shows the results of planar clustering applied to the Fandisk model. Note that for this model the results of both algorithms are similar. This is due to the fact that in regions with zero Gaussian curvature the hierarchical approach adequately merges regions in principal directions of zero curvature.

However, as presented in Figure 4.18 and Figure 4.19, for shapes with non-zero Gaussian curvature the ML algorithm provides a better planar approximation, resulting in a higher overall fitting quality. It is also important to observe in Figure 4.19 the "smoothness" of the cluster boundary. Thus, for shapes with non-zero Gaussian curvature the ML algorithm will, in general, provide better results compared to hierarchical clustering.

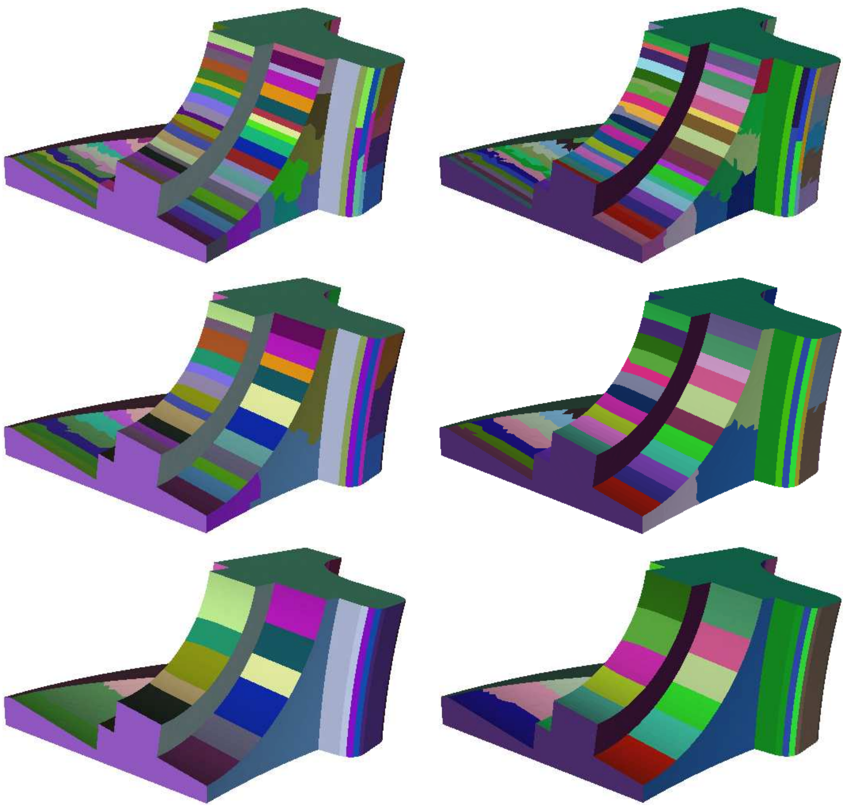


Figure 4.17: Planar clustering results using the HFC algorithm (left) and the ML algorithm (right) for the Fandisk model at the level of 200, 100 and 50 clusters, respectively.

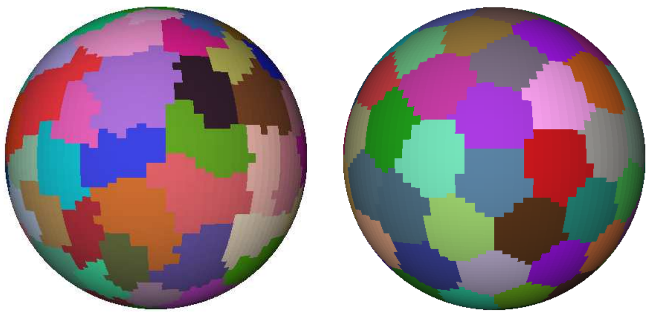


Figure 4.18: Planar clustering results using the HFC algorithm (left) and the ML algorithm (right) for a sphere model at a level of 100 clusters.

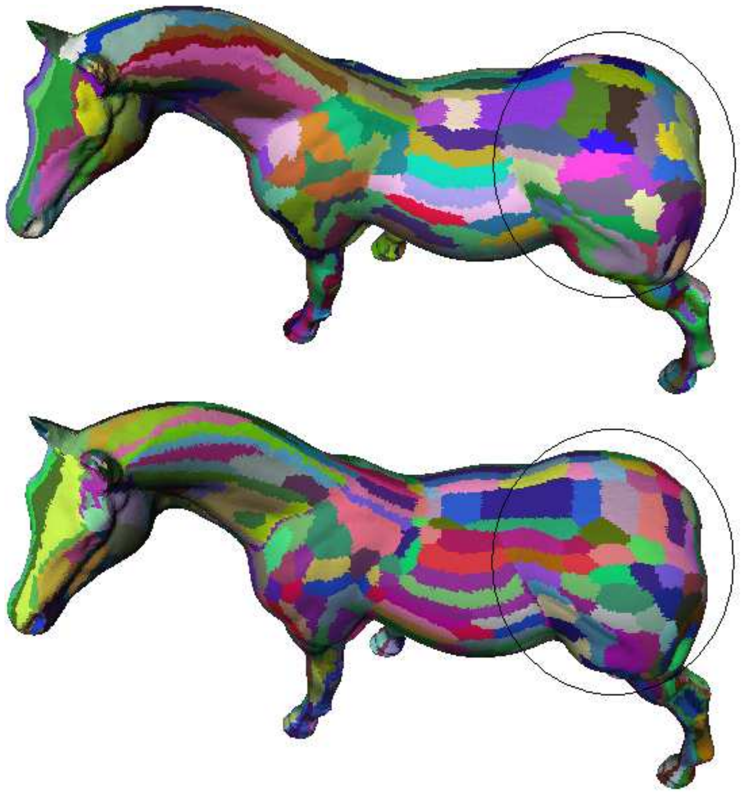


Figure 4.19: Planar clustering results using the HFC algorithm (top) and the ML algorithm (bottom) for the Horse model (top view) at a level of 500 clusters.

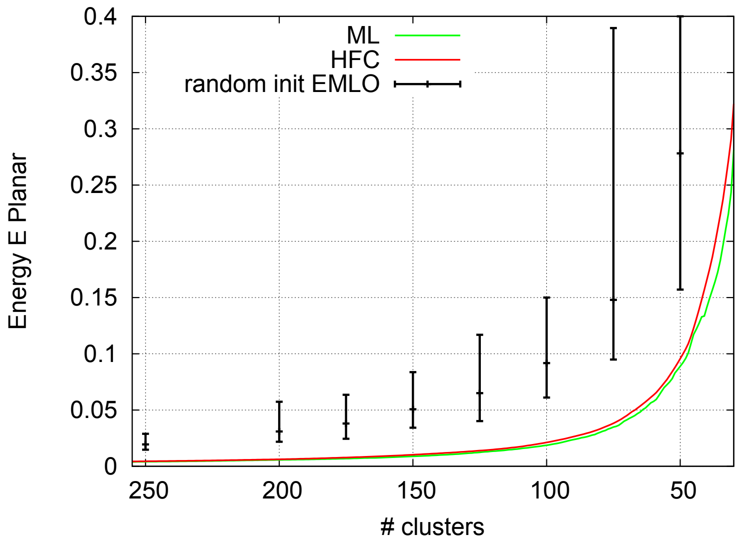


Figure 4.20: Planar energy versus # of clusters for the Fandisk model. HFC: hierarchical face clustering, ML: multilevel clustering, EMLO: with 50 random initialization.

Figure 4.20 shows the energy behavior of the planar clustering for different number of clusters, applied to the Fandisk model. The ML energy is always lower than that of HFC and EMLO approaches, which means a better clustering quality.

However, note that in contrast to the CVD clustering problem (see Figure 4.16) the energy of the EMLO is larger than that of the HFC and respectively of the ML. This behavior is due to random initialization, which is not able to generate more starting seeds in higher curvature regions for a better approximation. In contrast, the HFC is performing first the mergings in zero curvature regions, thus leaving more clusters in higher curvature regions, and this yields a lower energy.

This result is very important because, in general, it is believed that the variational methods always outperform the result of hierarchical clustering. This result indicates the contrary and any clustering problem with similar energy functional is expected to have similar energy behavior.

Sphere Fitting:

The energy of a spherical mesh approximation, see Section 3.3.2, for initial configuration can be written as:

$$^{**}E^0_{sphere} = (E^{reduced}_{sphere})_q + (E^{reduced}_{sphere})_p.$$

$$(E^{reduced}_{sphere})_i = -\frac{1}{k_i}\|\sum_{F_j\in C_i}\mathbf{p}_j\|^2 - \frac{1}{k_i}\frac{\left[k_i\sum_{F_j\in C_i}(\mathbf{p}_j\cdot\mathbf{n}_j) - \left(\sum_{F_j\in C_i}\mathbf{p}_j\cdot\sum_{F_j\in C_i}\mathbf{n}_j\right)\right]^2}{k_i^2 - \|\sum_{F_j\in C_i}\mathbf{n}_j\|^2}.$$

$^{**}E^1_{sphere}$ and $^{**}E^2_{sphere}$ are computed in a similar way, see Eq. (3.15), (3.16).

The cost of a Dual Edge for two clusters C_q and C_p is then calculated similar to the Eqs (4.2) - (4.3) as:

$$Cost_{Sphere} = -^{**}E^0_{Sphere} - \frac{1}{k_{qp}}\|\sum_{F_j\in(C_q\cup C_p)}\mathbf{p}_j\|^2 - \frac{1}{k_{qp}}\frac{\left[k_{qp}\sum_{F_j\in(C_q\cup C_p)}(\mathbf{p}_j\cdot\mathbf{n}_j) - \left(\sum_{F_j\in(C_q\cup C_p)}\mathbf{p}_j\cdot\sum_{F_j\in(C_q\cup C_p)}\mathbf{n}_j\right)\right]^2}{k_{qp}^2 - \|\sum_{F_j\in(C_q\cup C_p)}\mathbf{n}_j\|^2}.$$

where $k_{qp} = k_q + k_p$ the size of both merged clusters.

Note that this representation provides the same advantages as the CVD energy functional. For each cluster C_i only the value $\sum\mathbf{p}_j$, $\sum\mathbf{n}_j$ and $\sum(\mathbf{p}_j\cdot\mathbf{n}_j)$ needs to be stored. Again, these values can be updated easily and thus a fast energy or merging cost computation is possible.

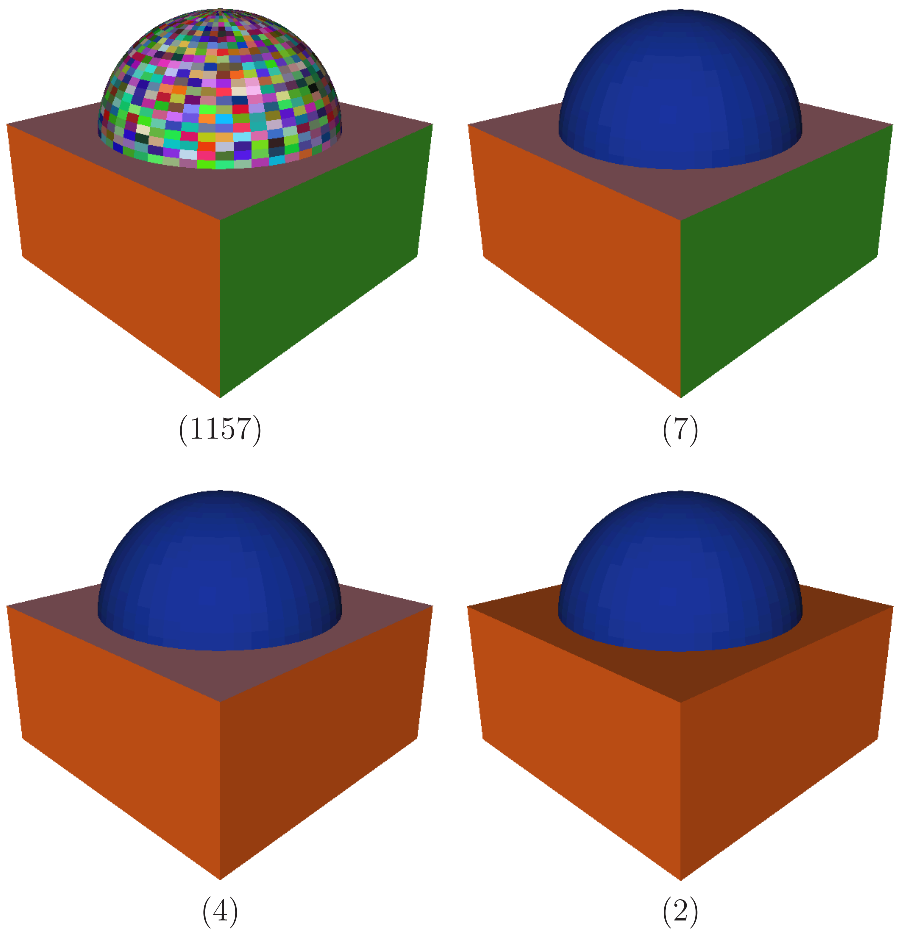


Figure 4.21: The sphere fit clustering results using the ML algorithm at different levels.

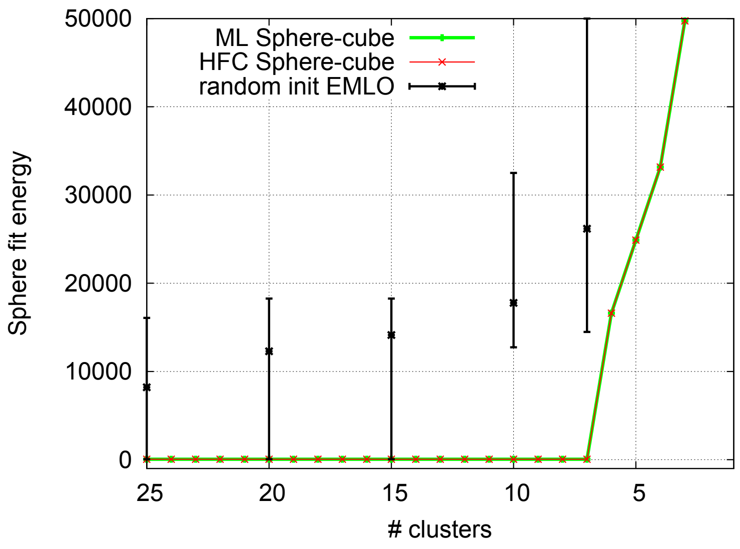


Figure 4.22: Sphere fit energy E_{sphere} behavior for the model presented in Figure 4.21.

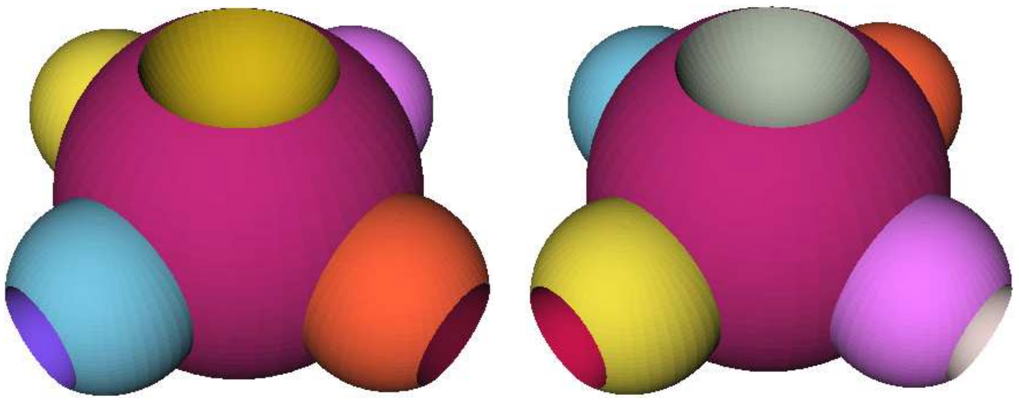


Figure 4.23: The sphere fit clustering results using the ML algorithm at the level of 11 clusters. A given model is composed of 11 spherical patches.

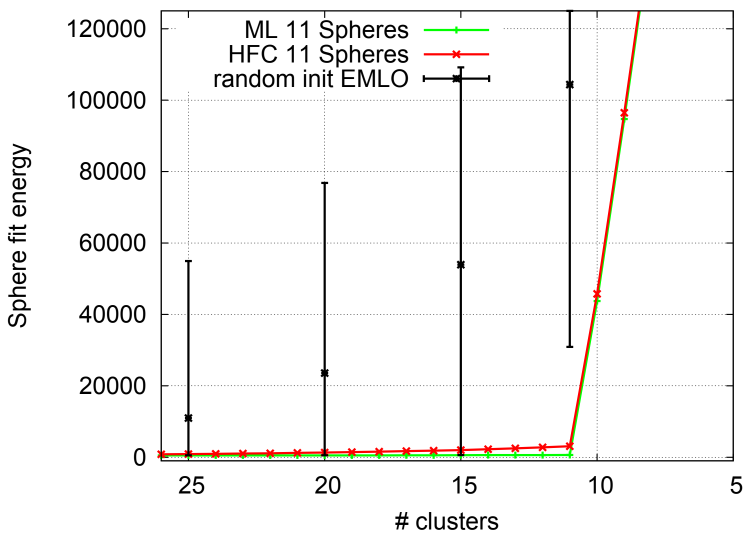


Figure 4.24: Sphere fit energy E_{sphere} behavior for the model presented in Figure 4.23.

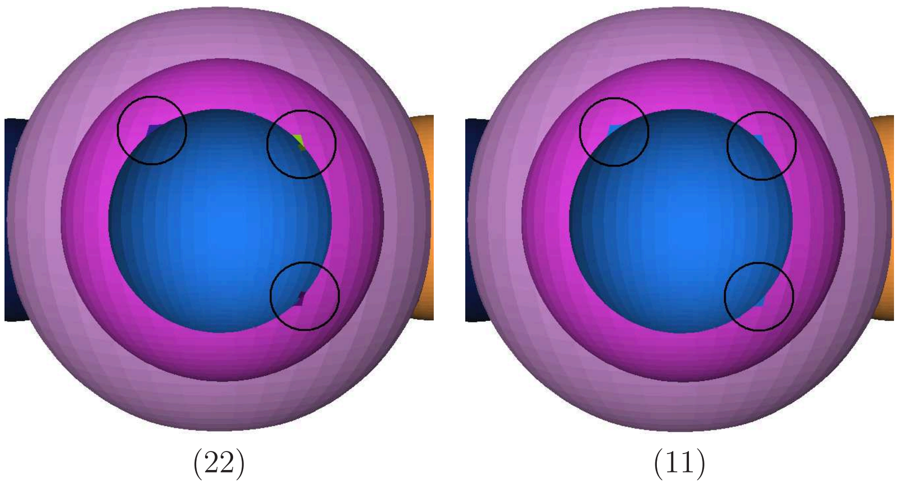


Figure 4.25: The sphere fit clustering results using the HFC algorithm at different levels. The “wrong” mergings applied at the beginning are propagated throughout all levels.

Figure 4.21 depicts the result of the Multilevel clustering algorithm at different levels for a model consisting of a cube and a spherical patch. Figure 4.23 depicts the result of the Multilevel clustering algorithm for a model consisting of 11 spherical patches. As expected, the ML approach is able to identify correctly all spherical patches present in the model. In Figure 4.21 the planar regions are merged first because they have zero energy, see discussions in Section 3.3.2.

Figure 4.22 and Figure 4.24 present the energy behavior for the models presented in Figure 4.21 and Figure 4.23, respectively. The energy of the ML approach remains zero until the “true” number of clusters present in the model is reached, followed by a fast increase as the number of clusters decreases, the so-called elbow effect. This fact can be used to stop the algorithm at an appropriate level and report the number of clusters which best approximate a given model.

Note in Figure 4.22 that the energy of the hierarchical clustering is identical to that of the ML. This indicates that there is no optimization taking place after the merging step. In contrast, for the model presented in Figure 4.23 the energy of the hierarchical clustering is slightly higher than that of the ML clustering, see Figure 4.24. This results from “wrong” mergings which are applied at the beginning of the clustering process. Because at the beginning each cluster consists of a small number of triangles, two clusters can merge even though they correspond to two distinct spheres. By applying optimization, the ML approach can recover itself from these “wrong” mergings at later levels, as the separation between spheres becomes more apparent. However, the HFC has no such possibility and propagates these “wrong” mergings in the upper levels.

An example of this problem for HFC is provided in Figure 4.25. Although, for this model this problem only slightly affects the final result, there can be situations, e.g. for noisy models, where this will accumulate more severely in the case of HFC. Thus, the ML approach is the best remedy in such situations.

Figure 4.22 and Figure 4.24 also show the energy of the EMLO algorithm with random initializations. For large number of clusters the EMLO is able in some situations to reach the energy of the ML and HFC algorithm, although on average the result is worse. However, as the number of clusters approach the "true" number of patches present in the model the result of the EMLO becomes even worse. This behavior, as in the case of the planar approximation, see Figure 4.20 on page 81, is due to bad random initialization.

Thus, the quality of the provided results for the Hierarchical face clustering compared to variational clustering is, in general, problem dependent. In contrast, the Multilevel approach does not suffer from such problems and always provides the best solutions, regardless of the clustering problem.

Timing:

Table 4.3 presents the timing for the Multilevel approach applied to different models. As expected, because the ML clustering provides all solutions, it is time consuming.

Model	# Faces (input mesh)	Energy Functional	Time (sec.)
Fandisk	13k	CVD	2
Fandisk	13k	planar	2
Fandisk	13k	sphere	35
11Spheres	28k	CVD	20
11Spheres	28k	planar	21
11Spheres	28k	sphere	91
Bunny	70k	CVD	281
Bunny	70k	planar	288
Bunny	70k	sphere	518
Horse	97k	CVD	710
Horse	97k	planar	622
Horse	97k	sphere	1153

Table 4.3: Clustering time for ML algorithm.

Nonetheless, due to the usage of the EMLO algorithm in the optimization step and because this optimization is applied only locally, see Section 4.2, the ML algorithm converges quickly. The time required to build a CVD for Bunny model for one level at 594 clusters is approximatively equal to 3 seconds, see Table 3.1. In contrast, to obtain the complete set of $70k$ solutions with ML algorithm requires 281 seconds. Thus, the ML construction is *very* efficient in this sense.

4.5 Different Variants of the Multilevel Clustering

One of the most important properties of the Multilevel algorithm is its flexibility. The merging and optimization steps can be altered in an arbitrary sequence or even deactivated if required. Thus, different variants of the algorithm are possible.

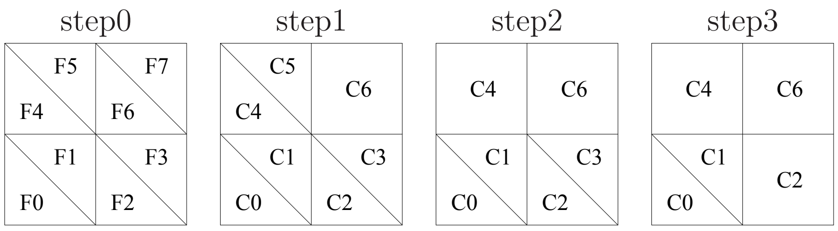


Figure 4.26: Example of possible mergings applied in the first steps.

In this context, it is also important to note that at the beginning the clusters are composed only of one face and the algorithm most likely merges only two by two of these single neighboring clusters, see in Figure 4.26 *step*0 to *step*3. Thus, depending on the addressed problem, it may be adequate to skip the optimization step for these first merging steps.

Indeed, regarding the energy behavior for spherical approximation in Figure 4.22 on page 83, the optimization can be deactivated for all levels without any quality loss. However, for the case presented in Figure 4.24 the optimization can be deactivated only for a specific number of merging steps, otherwise the quality of the final result will be lower. The same applies for planar approximation, see the energy behavior in Figure 4.20 on page 81.

However, speeding up the ML algorithm by deactivating the optimization step in different scenarios is not the only possibility. Remember that, to perform one merging step, one minimal cost ODE is identified in CA by performing one step of a bubble sort, see Section 4.2. This means iterating once trough the CA to perform *only one* such merging, which is, in general, time consuming.

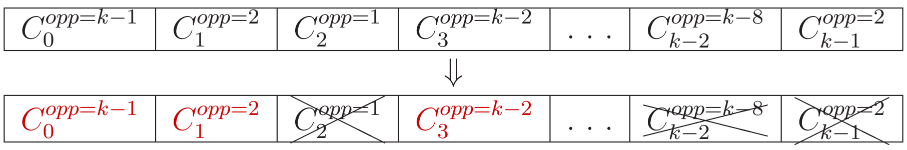


Figure 4.27: Example of the sequential merging of clusters in CA. The index *opp.* indicates the opposite merging cluster for a given ODE.

Instead, we propose to apply all possible mergings in CA sequentially up to a user defined parameter $p\%$ in *one* step and not in $p\%$ steps. Thus, after initialization, sequentially for each valid cluster `C` in the CA an ODE is identified and directly collapsed if the `C.ODE.opposite` cluster was not already involved in another merging operation. This step is repeated until $p\%$ of clusters are merged. An example of this operation is provided in Figure 4.27.

All steps of this variant of the ML approach are summarized in the Algorithm 4.2.

Algorithm 4.2. *(A p% Variant of the Multilevel Algorithm)*

```
1 Sequentially merging $p\%$ of clusters.
2 Loop until number of clusters equals 1 {
3   Collapse one ODE.
4   Apply optimization.
5 }
```

Table 4.4 presents the timing for the ML approach according to the Algorithm 4.2 for different values of $p\%$. Note that, this variant of the ML algorithm allows a substantial reduction of the computational cost compared to the standard ML approach (Table 4.3).

Model	# Faces (input mesh)	Energy Functional	p %	Time (sec.)
11Spheres	28k	CVD	50	4
11Spheres	28k	CVD	95	1
Horse	97k	CVD	50	200
Horse	97k	CVD	95	33

Table 4.4: Clustering time for a variation of the ML algorithm for different $p\%$ values.

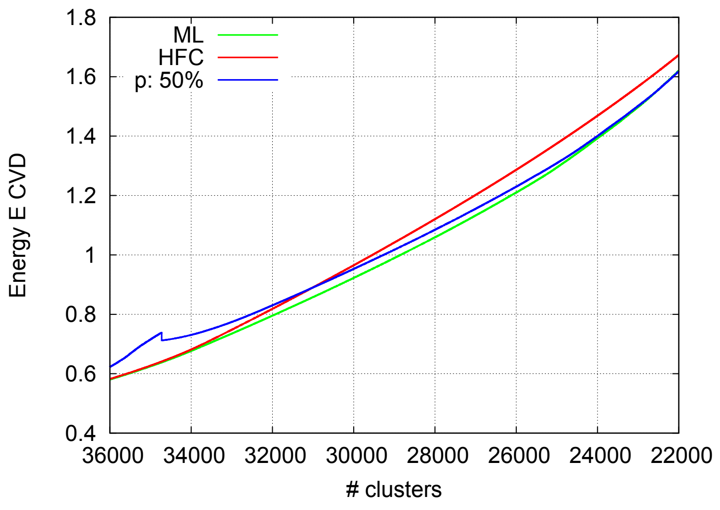
However, it is questionable how these sequential mergings affect the final quality of the result. In Figure 4.28 we show the dependency between the total CVD energy and the number of clusters for different clustering algorithms, applied to the Bunny model. The p factor refers to the first $p\%$ of sequential merges applied. Note the energy jumps which appear at the point where $p\%$ of the merging steps are done. Because these mergings are done sequentially, the total energy in this case is higher than that of the HFC algorithm. After this point the standard ML algorithm is applied, and as a result the energy starts to decrease approaching the energy of the ML algorithm.

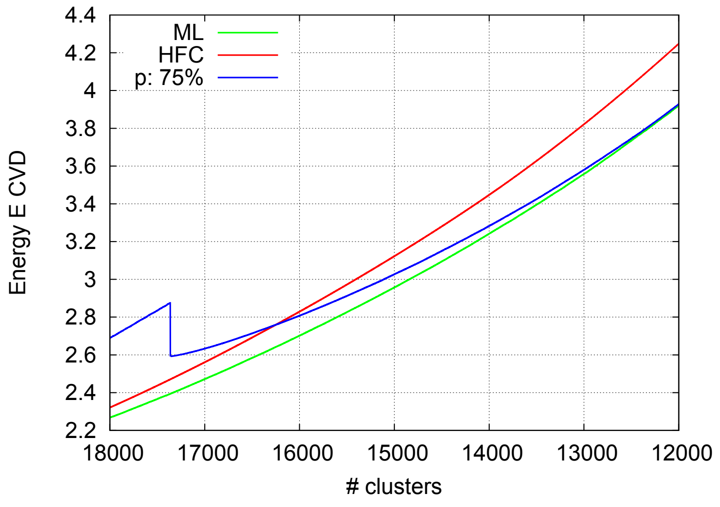
Table 4.5 depicts the time required for different algorithms to obtain the clustering/energy results presented in the Figure 4.28.

Algorithm	ML	HFC	$p = 50\%$	$p = 75\%$	$p = 95\%$
Time, sec.	281	218	52	18	9

Table 4.5: Clustering time for different clustering algorithms for Bunny model. ML: Multilevel mesh clustering. HFC: Hierarchical face clustering. (p:) Applying a variant of the ML algorithm according to the Algorithm 4.2.

Note, the energy of the ML and of the $p = 95\%$ in the Figure 4.28 in the range 2500 to 2000 clusters is identical. Thus, if the region of interest, regarding the number of clusters, is similar, then this variant of the algorithm, i.e. Algorithm 4.2 with $p = 95\%$, can be used to considerably speedup the computation without any substantial quality loss.





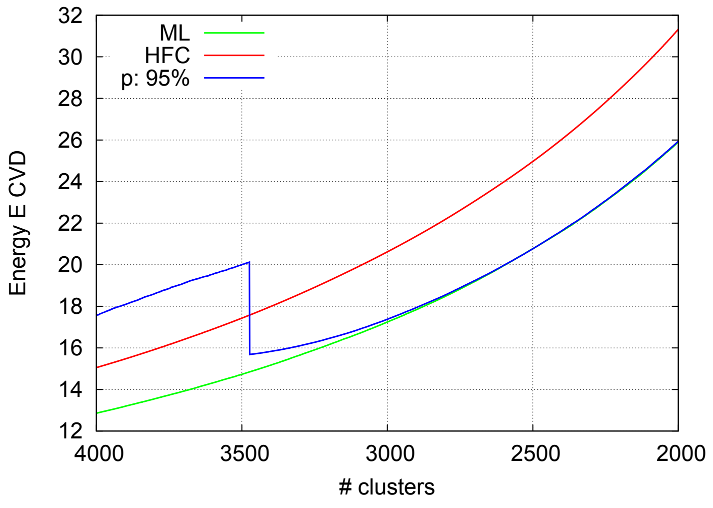


Figure 4.28: CVD Energy versus # of clusters for the Bunny model. ML: Multilevel mesh clustering. HFC: Hierarchical face clustering. (p:) Applying a variant of the ML algorithm according to the Algorithm 4.2.

Although the parameter p is a user specified one, for most of the problem a p between 50% *to* 70% proves to have no substantial influence on the results, yet allowing considerable speedup.

4.6 Conclusions

In this chapter we have presented a novel and versatile algorithm for performing Multilevel mesh clustering. As originally intended the algorithm resolves the inherent problems of Variational and hierarchical clustering. The algorithm neither uses any heuristics to obtain the final result nor any a-priori user specified parameters. It performs a complete mesh analysis with better, or at least the same, quality results.

We have provided a very efficient implementation and data structure for this algorithm. We showed that different variants of the algorithm are possible, thus allowing the user to choose between faster execution or higher quality of the final result.

This way, the Multilevel approach proves to be a powerful and reliable tool for mesh clustering. It allows a better mesh analysis and provides an indispensable control over clustering process.

We also have shown the generic nature of this approach by applying it to different tasks. However, we must point out that, because the EMLO algorithm developed in the last chapter is used in the optimization phase of the algorithm, the applicability of the ML algorithm is still restricted only to the set of energy functionals which can be represented in an incremental energy formulation.

In the next chapter, together with a solution for accelerating the ML construction on the GPU, we address this restriction by reformulating the EMLO algorithm such that it can accept any proxy-based energy functionals, thus allowing for a larger set of energy functionals.

Chapter 5

GPU-based Mesh Clustering

Fast and high quality clustering of large polygonal surface meshes still remains one of the most demanding fields in mesh processing. Because existing clustering algorithms are very time-consuming, the use of parallel hardware, i.e. the graphics processing unit (GPU), for speeding up these algorithms is a reasonable and at the same time a crucial task in this field.

However, up until now only a few approaches have been developed to utilize this kind of hardware for mesh clustering purposes. One of the first and most prominent works in GPU-based acceleration of the iterative (Variational, Lloyd's) clustering was the work of Hall and Hart [HH04]. Here, the GPU is *only* used to compute pairwise distance information and the proxy computation is done on the CPU. Thus only a small acceleration factor of 2 to 3 can be obtained. The work in [CTZ06] and later in [SDT08] proposed more efficient GPU-based solutions but only for k-means approaches. For more details on GPU-based processing see Section 2.3.

Although the solution proposed in [HH04] has been shown to work for polygonal meshes, *no* mesh connectivity information is actually used in the clustering process. The mesh is simply viewed as a triangle soup and the clustering is done on a k-means basis, which can be a major drawback in many mesh clustering applications. Figure 5.1 depicts such an example. Note that, if no mesh connectivity is used in the clustering process the clusters are split into two over different helix coils. In contrast, using the mesh connectivity in the clustering process aligns the clusters along helix coils.

The *only* existing GPU-based mesh clustering framework completely implemented on the GPU which employs the mesh connectivity (topology) in the clustering process is proposed in [CKCL09], [CK11]. In this chapter we describe[1] most of the details related to this framework.

In the first part of the chapter, i.e. Section 5.1, we develop new algorithms which obey a parallel formulation. In Section 5.1.1 a new Boundary-based mesh clustering algorithm is proposed as a new approach for parallel cluster optimization. It provides all necessary ingredients for a GPU-based implementation without introducing any special energy func-

[1]Parts of this chapter were published in [CKCL09] and [CK11].

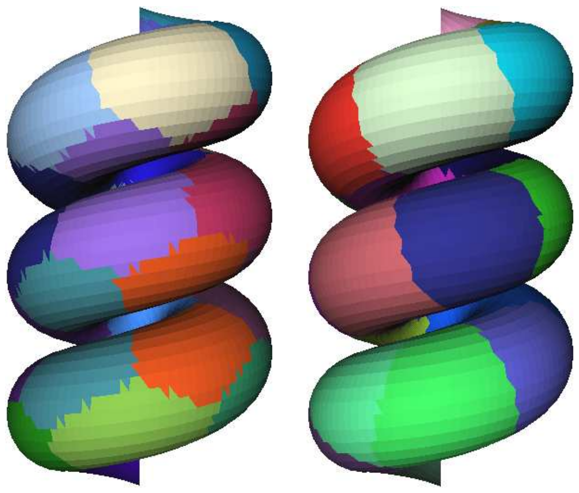


Figure 5.1: A clustering example without (left) and with (right) connectivity taken into account. Applied to a helix model for 25 clusters.

tional requirements. In Section 5.1.2 a new parallel Multilevel clustering concept, for which several Dual Edges are collapsed in each step, is proposed for speeding up the Multilevel mesh clustering on the GPU.

In the second part, i.e. Section 5.2, we address all GPU-specific implementation details and give some non-trivial OpenGL specific solutions. Section 5.2.1 describes general processing and data structures concepts. Section 5.2.2 describes how the mesh geometry and connectivity is encoded on the GPU. This gives all necessary means for a complete iterative or Multilevel GPU-based mesh clustering, which is described in Sections 5.2.3 - 5.2.5. In Section 5.3 we evaluate the proposed algorithms and their GPU-based implementations. Section 5.4 draws some final conclusions.

5.1 Parallel Mesh Clustering

A major problem that impedes the use of the GPU for mesh clustering and by that to exploit the computational power of this hardware, is the lack of parallel algorithms for mesh clustering. All approaches mentioned so far have more or less a *sequential* nature of processing. The Variational (Section 2.1.2) or hierarchical (Section 2.1.3) mesh clustering approaches use a global priority queue in order to identify the best face-cluster assignment or the best merge. The EMLO algorithm (Algorithm 3.3) requires a direct cluster data update to be propagated to subsequent steps after any face assignment. Thus, these algorithms can not be parallelized easily, resulting in a rather bad scalability for large meshes.

In this section we review these problems for both iterative and hierarchical mesh clustering approaches and propose parallel reformulations so that these can be implemented on the parallel hardware, i.e. on the GPU.

5.1.1 Boundary-based Mesh Clustering

In this section a new ***Boundary-based (BB)*** mesh clustering approach is described. It belongs to the same class of *iterative* approaches as the classical Variational clustering (VC), see Section 2.1.2, or the Energy Minimization by Local Optimization (EMLO), see Chapter 3. However, compared to both the proposed approach has a very important property – it has a parallel formulation.

Consider a mesh M that is clustered into k clusters C_i. As for VC, a shape proxy P_i is a local representative of the cluster C_i that best approximates a given cluster geometry. Accordingly, the proxy set $P = \{P_i\}$ approximates the whole mesh geometry, for more details see Section 2.1.2.

Now, suppose that a proxy-based energy functional $E(P)$ (Definition 2.2 on page 8) is provided:

$$E(P) = \sum_{i \in \Omega} E(C_i, P_i) = \sum_{i \in \Omega} \sum_{F_j \in C_i} E(F_j, P_i). \tag{5.1}$$

with $\Omega \in \{0, \ldots, k-1\}$.

$E(C_i, P_i)$ is the energy of the cluster C_i for a given proxy P_i and $E(F_j, P_i)$ is the positive semi-definite cost of assigning the face F_j to the cluster C_i with a corresponding proxy P_i.

Note, for the VC approach a *fixed precomputed* set of input proxies $\{P_i\}$ is used when assigning the faces F_j to the clusters C_i. Where the EMLO realizes indirectly proxy updates after *each* reassignment of a face F_j to the cluster C_i.

Now suppose that a Variational mesh clustering (Section 2.1.2) is done and a new set of proxies $\{P_i^{new}\}$ is fitted. The VC framework will continue with an identification of the initial seeds for the next partitioning phase. For each cluster C_i this is done by going once through all its faces F_j and identifying one face with the smallest energy $E(F_j, P_i^{new})$ and closest to the cluster center. These seeds, see Figure 5.2 (init grow), are then used to perform a new clustering *from scratch*.

However, in practice one can easily observe that the cluster configuration changes rapidly during the first iterations and then starts to settle slowly, i.e a significant spatial coherence can be observed. Thus, the cluster configuration is mostly affected on the cluster boundaries whereas the cluster's "interior" does not change, see Figure 5.2 (1 iter.) - (46 iter.). Thus, we find that in the process of cluster optimization, checking these "interior" regions for reassignment, as done in the case of the classical VC, is not necessary, and can be limited to boundary regions only.

To implement this idea a modified VC approach can be considered. Instead of performing the regrowing from scratch we have to cluster inwards from the boundary of each cluster and consider only boundary faces or faces which are in the vicinity of the cluster boundary. Although this sounds plausible it still does not avoid the use of a Priority Queue, which as already stated can be a major obstacle in parallelizing and implementing this algorithm on GPU.

Another alternative is to use a variant of the EMLO algorithm. Its *locality-based* check paradigm is well suited in this case. Additionally, as required, the EMLO performs and

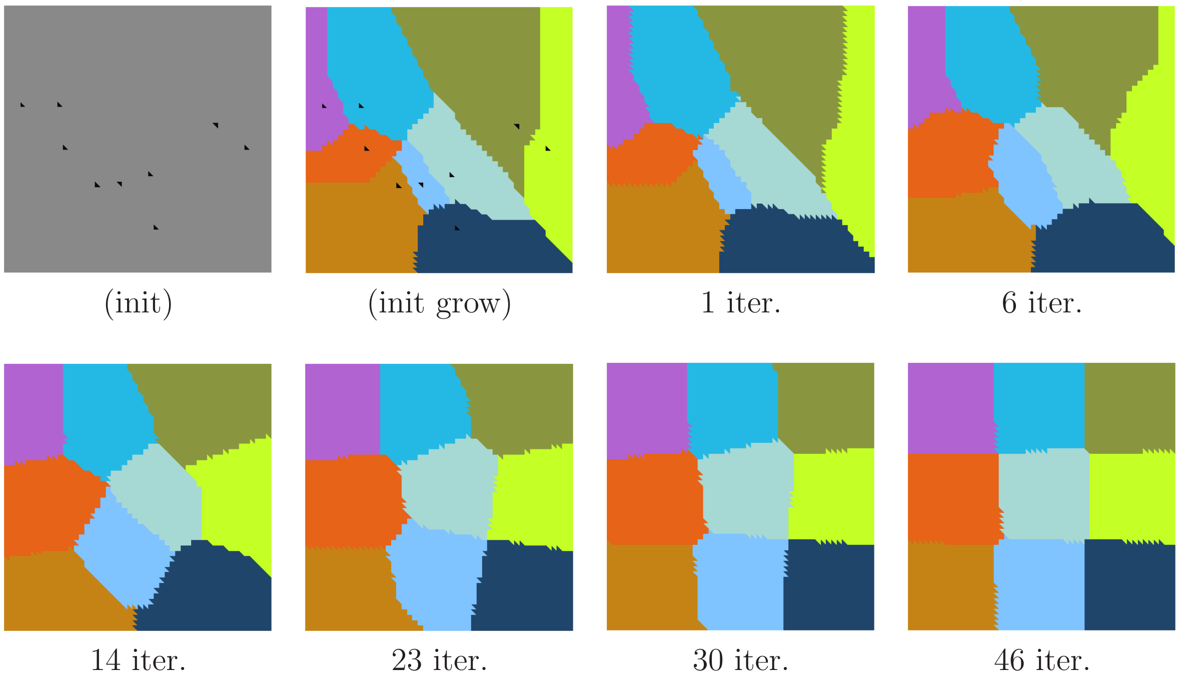


Figure 5.2: Boundary-based cluster optimization steps: (init) Black triangles are the starting seeds for initialization. (init grow) Initial cluster grow. (1 iter.)-(46 iter.) Clustering results after corresponding number of iterations.

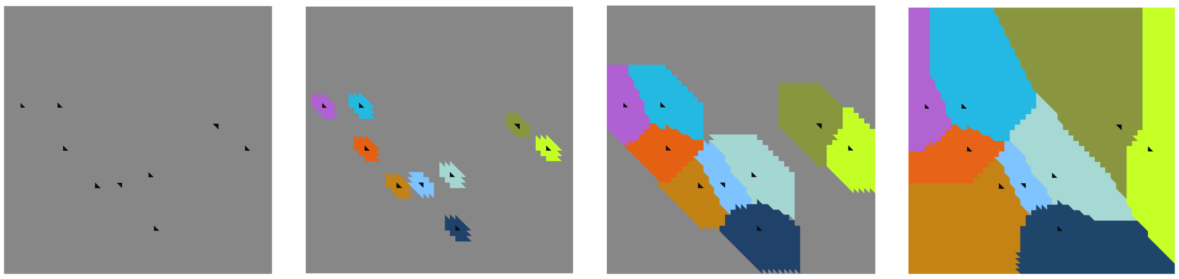


Figure 5.3: Initial cluster growth for given starting seeds (black triangles).

considers only the cluster's boundary faces. Despite this, it has a restriction on the energy functional to be in an incremental formulation for an efficient energy computation.

Instead, we propose a new clustering concept, namely the **Boundary-based** cluster optimization, by redefining the classical VC approach [CSAD04] and making use of the idea of local optimization from the EMLO approach [CK08].

Before performing a BB cluster optimization, an initial clustering configuration is required, i.e. a configuration that needs to be optimized. To obtain such a configuration the same procedure is used, as in the case of the EMLO mesh clustering approach, see Section 3.3.

Using a random initialization, k starting seeds are generated, see Figure 5.3 (black triangles). Then the initial cluster growth is done by iteratively assigning the free (not assigned to any cluster) neighboring face of a cluster's boundary edge, see Figure 5.3. Generally, this process is very fast because no energy computation is involved.

Boundary-based Cluster Optimization

Suppose that an initial clustering configuration, as depicted in Figure 5.2 (init grow), for k clusters C_i with proxies P_i and with corresponding energy $E(P) = \sum_i E(C_i, P_i)$ is given. Now, minimizing the total energy $E(P)$ of a given clustering can be done by reassigning the cluster's boundary faces to other clusters in such a way that the total energy $E(P)$ decreases.

An example of this process for two clusters C_q and C_p with proxies P_q and P_p and two neighboring faces F_m and F_n of a boundary edge e is presented in Figure 3.15 on page 54. For each boundary edge the following energy cases are considered:

- **Case 0:** Initial configuration with corresponding energy $E(P)^0$ where the face F_m belongs to C_q and F_n belongs to C_p.
- **Case 1:** A case when cluster C_q grows and C_p shrinks, thus both F_m and F_n belong to C_q, with corresponding energy $E(P)^1$.
- **Case 2:** A case when cluster C_p grows and C_q shrinks, thus both F_m and F_n belong to C_p, with corresponding energy $E(P)^2$.

Note, these energy cases are in the same spirit as considered for EMLO algorithm (Section 3.3) or as originally proposed in [VC04], however with a substantial difference in the way these energies are computed.

We propose to compute the energies $E(P)^0$, $E(P)^1$ and $E(P)^2$ as follows:

$$\begin{aligned}
E(P)^0 = & \sum_{F_j \in C_q \setminus \{F_m\}} E(F_j, P_q) + E(F_m, P_q) + \\
& \sum_{F_j \in C_p \setminus \{F_n\}} E(F_j, P_p) + E(F_n, P_p). \\
E(P)^1 = & \sum_{F_j \in C_q \setminus \{F_m\}} E(F_j, P_q) + E(F_m, P_q) + E(F_n, P_q) + \\
& \sum_{F_j \in C_p \setminus \{F_n\}} E(F_j, P_p). \\
E(P)^2 = & \sum_{F_j \in C_q \setminus \{F_m\}} E(F_j, P_q) + \\
& \sum_{F_j \in C_p \setminus \{F_n\}} E(F_j, P_p) + E(F_m, P_p) + E(F_n, P_p) \qquad (5.2)
\end{aligned}$$

The energies are computed with a fixed set of proxies, and no direct proxy update is considered. This is in the same spirit as done for the VC approach.

For comparing the energies $E(P)^0$, $E(P)^1$ and $E(P)^2$ in the above formulas, the sums are irrelevant, thus Eq. (5.2) can be simplified to:

$$\begin{aligned}{}^*E(P)^0 &= E(F_m, P_q) + E(F_n, P_p).\\ {}^*E(P)^1 &= E(F_m, P_q) + E(F_n, P_q).\\ {}^*E(P)^2 &= E(F_m, P_p) + E(F_n, P_p).\end{aligned} \tag{5.3}$$

Note that, comparing ${}^*E(P)$ in Eq. (5.3) is equivalent to comparing the energies $E(P)$ in Eq. (5.2), but at a lower computational cost because the irrelevant summations are dropped.

The complete cluster optimization is done according to the Algorithm 5.1.

Algorithm 5.1. *(Boundary-based (BB) Cluster Optimization)*

```
Loop until no configuration changes {
  Compute proxy set $\{P_i\}$
  For all Clusters $C_i$
    For all Boundary Loops $b$ of $C_i$
      Loop over boundary edges $e \in b$ {
        Compute energies ${}^*E(P)^0$, ${}^*E(P)^1$, ${}^*E(P)^2$
        Choose the smallest energy and update cluster configuration
      }
}
```

For a given boundary edge e the smallest energy ${}^*E(P)$ is chosen and a corresponding configuration is updated, i.e. cluster grow or shrink or no configuration changes. After the optimization, a new set of proxies $\{P_i\}$ is computed and used to perform another boundary-based energy minimization. Because the energy $E(P)$ is supposed to be positive semi-defined, any boundary modification lowers the energy and thus the algorithm converges in general[2], i.e. for a given set of proxies there are no boundary faces that can be reassigned to other clusters so that the energy decreases.

The most important property of this algorithm is that any computation of energy ${}^*E(P)$ can be done independently from any other, and there is no cluster data update required as for EMLO after a face is reassigned from one cluster to another. All this provides the base to perform the energy computations in parallel, making the Algorithm 5.1 suitable for a GPU-based implementation[3].

[2]Although not observed in practice, in some cases the algorithm may not converge due to numeric problems when recomputing the proxies. For these cases a predefined maximum number of iterations should be provided.

[3]**Remark:** Although, our intent is to implement the BB algorithm on the GPU, it can be also employed on the CPU with no restrictions. Similar to EMLO it can be used as a standalone algorithm or in the optimization stage of the ML algorithm. To accomplish this, the same strategy is used as described for the EMLO algorithm in Section 3.3.

It should be noted that, in contrast to [VC04] and [CK08], this approach imposes no further limitations on the energy functional, since the proxies are fixed during the optimization. Thus, any proxy-based energy functional (Definition 2.2) can be used, allowing for a large set of energy functionals.

Note that in comparison to the classical VC approach [CSAD04] there is *no* identification of initial seeds, as well as *no* local or global priority queue to determine the next assignment of a face to a new cluster. Thus, this approach reduces the computational costs and, furthermore, it is well suited to be implemented on a GPU. In this form the algorithm can also be used in the optimization phase of the Multilevel mesh clustering approach, see previous chapter, thus removing the requirement on the special representation of the energy functional for an efficient clustering with ML algorithm.

5.1.2 Parallel Multilevel (PML) Mesh Clustering

In the Multilevel mesh clustering approach each cluster has its own ODE, which represents the Dual Edge with the smallest merging cost out of all DEs of a given cluster, see for more details on this idea Section 4.2.1. In each step of the ML construction a *single* ODE with smallest cost out of all ODEs is identified and collapsed. This step is then followed by an optimization phase, see Algorithm 4.1.

As we aim at implementing the ML approach on the GPU, both steps (merging and optimization) must obey a parallel formulation. Regarding parallel cluster optimization, the Boundary-based cluster optimization algorithm described in the last section, see Algorithm 5.1, can be applied with no restrictions to achieve a fast parallel optimization.

However, for the hierarchical part of the ML construction, where a single ODE collapse is applied, *only* a parallel identification of these ODEs is possible. Each cluster can identify its own ODE independently from any other clusters. Thus, identifying the ODEs has a parallel formulation by default. As a result, the main steps of the ML approach can be performed as described in Algorithm 5.2.

Algorithm 5.2. *(The Multilevel Algorithm)*

```
Loop until number of clusters equals 1 {
  Forall clusters compute in parallel ODEs
  Collapse the ODE with smallest cost.
  Apply optimization.
}
```

However, the procedure of one Dual Edge collapse in each step limits the generally required parallelism. It can be easily recognized in Figure 5.4 that a single ODE collapse in each step is not the most optimal procedure in this example. Performing *more* ODE collapses in one step, as presented in Figure 5.5, can lead to the same clustering result but with a significant speedup. Thus, the key idea is to identify a set of independent and mutual ODEs, i.e. consisting of pair of ODEs which connect the same clusters. This set

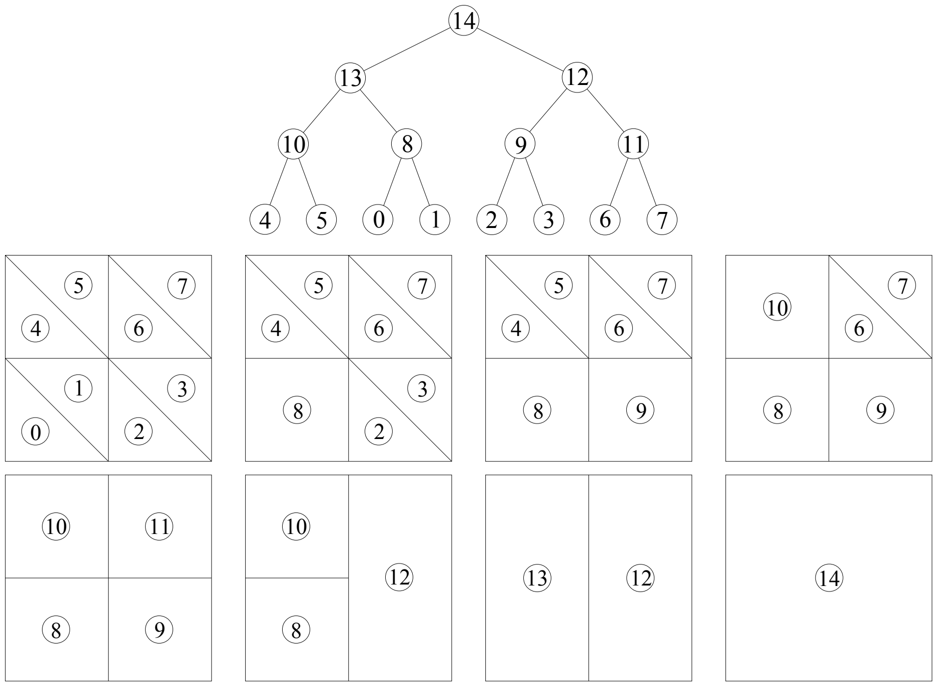


Figure 5.4: An example of sequential cluster merging and a resulting hierarchy (ML).

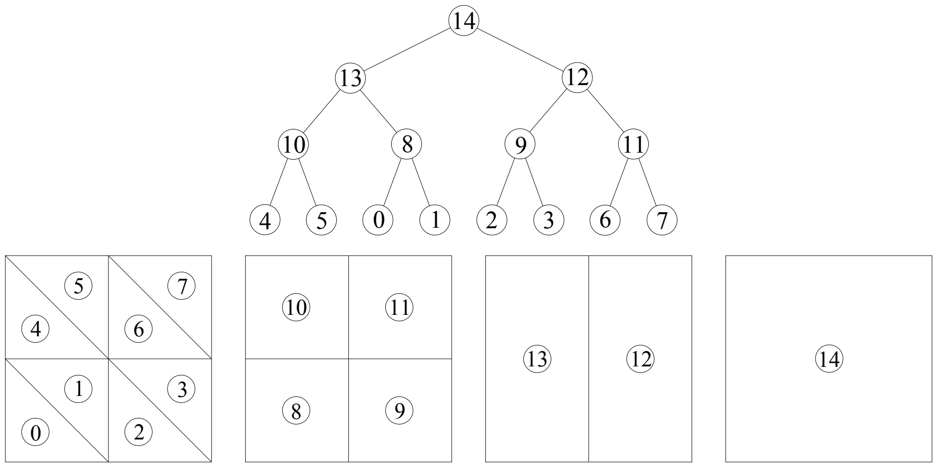


Figure 5.5: An example of parallel cluster merging and a resulting hierarchy (PML).

of ODEs can than be collapsed in parallel in one merging stepwithout introducing any inconsistency.

Based on this observation, we propose to apply multiple ODE collapses in parallel, yielding a *Parallel Multilevel (PML)*[4] clustering technique. The main steps of the PML approach are summarized in Algorithm 5.3.

Algorithm 5.3. *(The Parallel Multilevel Algorithm)*

```
Loop until number of clusters equals 1 {
  Forall clusters compute in parallel ODEs
  Collapse all mutual ODEs in parallel.
  Apply optimization.
}
```

In general, parallel cluster merging raises the question of whether or not this approach leads to worse results regarding the energy functional. To reflect on this question consider the variant of the ML mesh clustering algorithm proposed in Section 4.5. There it was shown that for a specific range of interest regarding the number of clusters, sequentially merging the first $p\%$ of the clusters without optimization leads to the same clustering quality, when the optimization is reactivated after these clusters have been merged. Thus, we expect the PML to have at least the same energy behavior. Indeed, as we will see in Section 5.3, applying PML only, the clustering is faster, however the clustering result is worse compared to ML. However, after switching from PML to ML, the energy rapidly converges against the ML variant and finally yields the same clustering result.

It should be pointed out that, the PML algorithm, as the ML algorithm, is flexible in applying the optimization and/or the parallel ODE collapse at any stage of the Multilevel mesh analysis. For example, for some clustering problems one could use the PML with optimization deactivated, i.e. perform a *parallel hierarchical* clustering. Or perform $p\%$ of ODE collapses with PML clustering and then continue with sequential ODE collapses for the remaining steps, see Section 5.3 for an example and discussions.

5.2 GPU-based Mesh Clustering

In this section we describe GPU-specific implementation details for the algorithms proposed in Section 5.1. We aim at performing the complete mesh clustering entirely on the GPU, reducing the data transfer between CPU and GPU to a minimum. The proposed GPU-framework is generic, thus we omit any energy functional related details.

Our implementation is OpenGL-based mainly due to the fact that the clustering framework is based on an existing generic GPU-framework [Cun09a], which was developed in the context of real-time particle systems for flow visualization [Cun09b]. This allowed for

[4]The ML and PML always refer to the version with *single* respectively *multiple* ODE merge, regardless of its implementation on the CPU or GPU.

an easy and efficient integration of all clustering algorithms. Although a CUDA-based implementation could provide a more structured implementation, our general expectation is that this will not provide substantial performance gains.

5.2.1 Processing Concepts and Data Structures

On the GPU, entities (e.g. vertices or fragments) are processed in parallel and independent from each other. Mesh clustering on a GPU must obey the same processing concept. A decision on whether a face must be assigned or reassigned to a different cluster must be made independently and in an arbitrary order from any decision made for its neighbors. The same is also true for cluster merging decisions. Thus a *per-face* or a *per-cluster* processing is applied. Depending on the applied energy functional the data necessary for energy computation is saved in associated *FaceData* or *ClusterData* textures, correspondingly.

For Multilevel clustering approaches the starting number of clusters k is identical to the number of faces m. However, in the case of Boundary-based clustering k is user specified and $k \leq m$ (usually $k \ll m$). Despite this we always assume $k_{max} = m$ and use a corresponding texture size, i.e. $texSize = ceil(sqrt(m))$. This simplifies all cluster data fetches, e.g. for a cluster with index r the data is located in a texture with size $texSize$ at the position $(mod(r, texSize), floor(r/texSize))$. For a *cluster*-defined texture a texel then simply refers to an individual cluster data.

In our current framework, only 2D 32-bit float textures are used to store the associated *FaceData* and *ClusterData*. On current graphics hardware the 2D texture maximum size limit is 8192, thus theoretically up to 8192^2 faces or clusters can be stored.

The input and the output textures on the GPU must be kept distinct. Thus, a double buffering approach is used to separate the input and the output textures [KLRS04], [SDK05]. Using a flip-flop technique, which exchanges the role of input and output, different operations can be performed on these textures.

Two basic techniques are used to perform most of the clustering tasks:

1. **Texture rasterization:** The process is triggered by rendering a quad which matches the output texture. During rasterization, for each output texel a fragment is generated. In the fragment shader the texture coordinates are used to read the input texture or other textures using texture fetching. This information is used to perform all necessary computations and finally assign a data to the fragment. If the output texture is also a render target then it is updated correspondingly.

2. **Vertex scattering:** The texture data are reinterpreted as a vertex stream. In the vertex shader, depending on loaded information, different texture fetches or computations are performed. The output vertices are rendered as points with the output positions computed in the vertex shader[5]. If an output texture is set as a render tar-

[5]Due to possible collisions in the scatter addresses, this process may not be efficient and it should be avoided if possible [Buc05].

get, then each rasterized point, i.e. fragment, will correspondingly update the texture at a precomputed position. This process fits into a single render pass [Buc05], [SH07].

5.2.2 Mesh Representation on GPU

Interestingly, a major problem that impedes the use of the GPU for mesh clustering, *even if a parallel algorithm formulation exists*, is the lack of the Half Edge data structure [Män88] on the GPU[6]. This restriction also explains why there is so little work done in this field.

As an example, the Boundary-based parallel algorithm proposed in Section 5.1.1 requires and heavily uses the Half Edge data structure to work on the boundary of the cluster. The ML algorithm in turn, see Section 5.1.2, requires the Boundary Loop of the cluster to identify the smallest cost ODE.

In its simplest form a mesh can be represented (reconstructed) by specifying the coordinates of the vertices together with a set of indices for faces, see Section 2.1.1. In this case, there is, in general, no possibility to obtain any neighboring information for a face without rebuilding the entire mesh. Due to processing constraints, rebuilding the mesh on the GPU and saving this information in some data structure is cumbersome. Additionally, it is also impossible to propagate any local changes to the neighbors, because each mesh entity is processed independently from each other.

Our solution to this problem [CKCL09] is to save the corresponding neighbors for each face in addition to the vertex coordinates and indices for faces. For a triangular mesh each face has exactly three neighbors, thus these indices $(ID_{n1}, ID_{n2}, ID_{n1})$ can be simply saved in a RGB channel, see example in Figure 5.6.

Observe that the face neighbors information allows navigating and collecting mesh information. Starting from one face any neighboring face can be obtained, by accessing the neighbors of the neighbors of a face. Using this representation one can compute any required information for each face: centroid, or normal, or curvature. Most importantly, it is also sufficient for performing any necessary clustering operations, as we will show later.

Moreover, due to the simplicity of the representation and because we mostly use textures to save any data in the GPU memory, we save the input mesh in an image based *pfs* format, see [MKMS07]. Three RGB frames are used to save, correspondingly, the vertex coordinates, vertex indices and face neighbors IDs, as sketched in Figure 5.7. The “face neighbors” frame contains for each face the indices for neighboring faces, as presented in Figure 5.6. In GPU memory these frames are also loaded in three corresponding textures.

Regardless of whether the mesh is clustered or not, each face belongs to a specific cluster with index ID_{cl}. In the beginning each face belongs to a “null” cluster, i.e. it has $ID_{cl} = -1$. We save this information together with face neighboring information (ID_{n1}, ID_{n2}, ID_{n3}) in a RGBA texture *FaceInfo*, as depicted in Figure 5.8. Here each texel corresponds to a face in the original mesh. Reassigning a face from a cluster m, i.e face has $ID_{cl} = m$, to cluster n means that the face will have $ID_{cl} = n$, as shown Figure 5.8.

Now, the most important element here is the notion of a cluster’s *boundary-face*:

[6]To our knowledge there is currently no such data structure representation on the GPU.

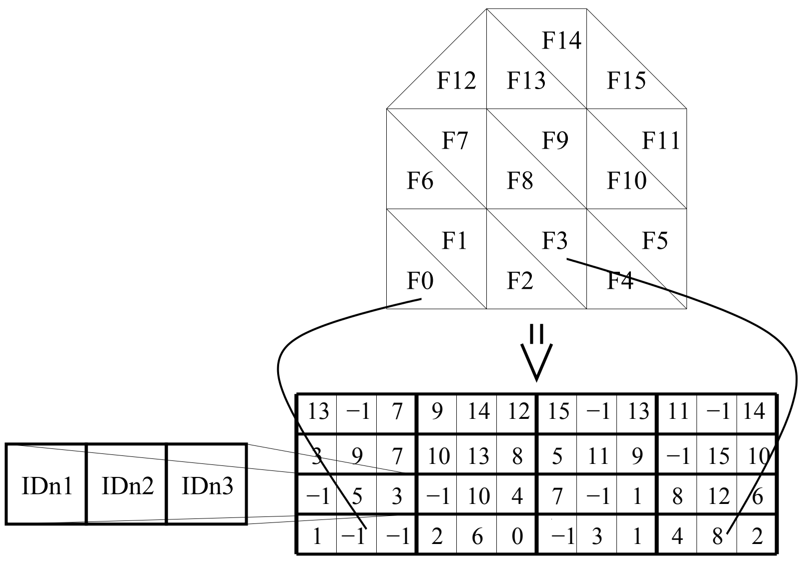


Figure 5.6: Example of stored face neighbors indices $(ID_{n1}, ID_{n2}, ID_{n1})$ for a given mesh.

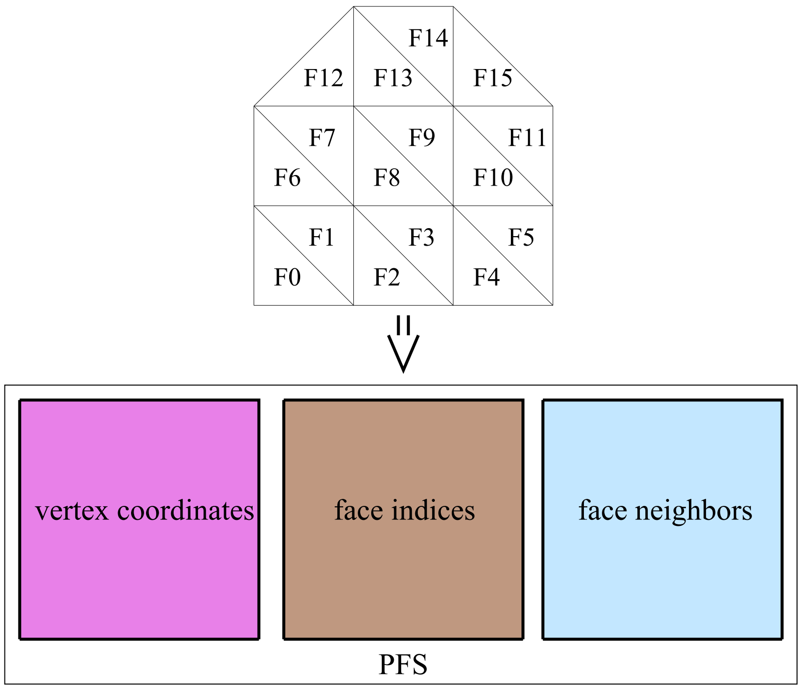


Figure 5.7: Mesh encoding in a PFS format with corresponding RGB frames.

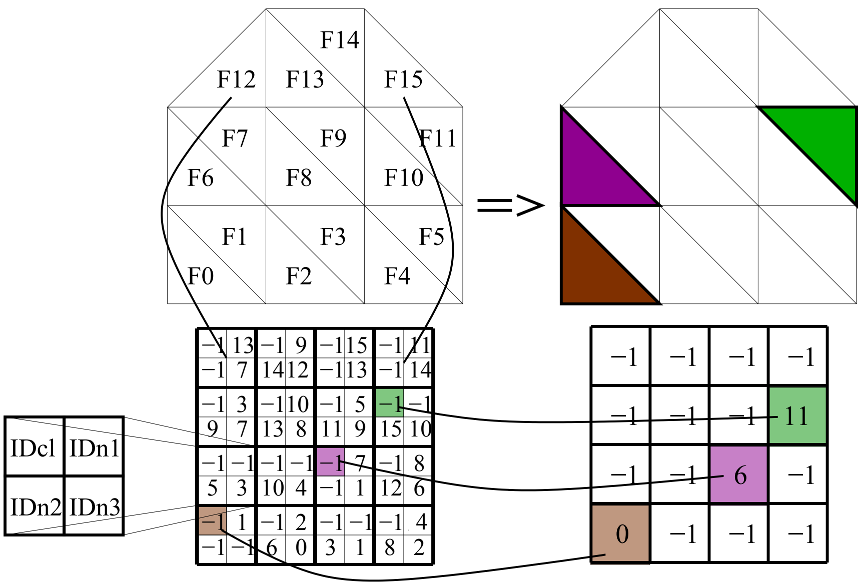


Figure 5.8: An example of the information stored in the $FaceInfo$ texture. *Top*: The mesh with corresponding IDs for faces. *Bottom*: The RGBA *FaceInfo* texture where each texel contains the $(ID_{cl}, ID_{n1}, ID_{n2}, ID_{n3})$ for each face. Any change of the face index ID_{cl} is propagated to the corresponding texel in the $FaceInfo$ texture.

Definition 5.1. *A face with* ID^*_{cl} *is a cluster* **boundary-face** *if at least one of its neighbors has* $ID_{cl} \neq ID^*_{cl}$.

As we will show later, using the notion of a boundary-face (Definition 5.1) the cluster growing, or the cluster optimization, or the ODE identification can be realized.

5.2.3 Boundary-based Mesh Clustering on GPU

For a user specified number of clusters k, the steps of the BB algorithm presented in Section 5.1.1 are implemented as follows:

Initialization:

Randomly generate or load from a file k starting IDs. Reset the face index ID_{cl} corresponding to these IDs in the *FaceInfo* texture. An example of this process is presented in Figure 5.8. Here three starting faces, i.e. $\{0;\ 6;\ 11\}$, are considered.

Initial Cluster Growth:

Regarded from the actual boundary of the cluster, there are two types of boundary faces, see Definition 5.1: interior $ID_{cl} > -1$ and exterior $ID_{cl} = -1$. Only *exterior boundary*

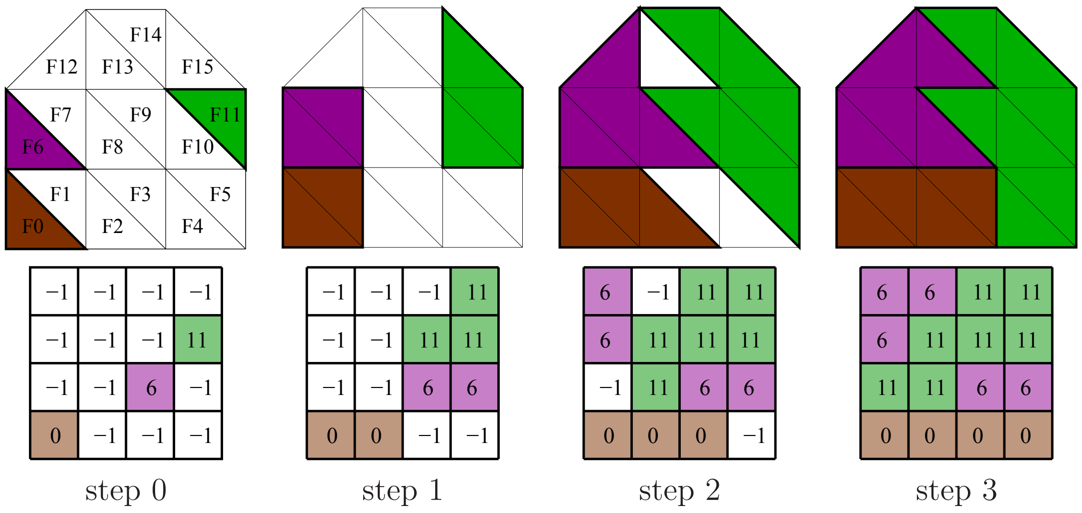


Figure 5.9: Example of an initial growing applied to a given mesh, with corresponding changes applied to the ID_{cl} channel of the $FaceInfo$ texture. Three render passes are required to obtain the final configuration.

faces must be added to a specific cluster in the growing phase, i.e. face ID_{cl} is changed from -1 to a corresponding cluster ID, see Figure 5.9.

This is achieved by rasterizing the $FaceInfo$ texture. In this case, fragments which correspond to different texels, i.e. faces, in $FaceInfo$ texture are generated. The fragments which correspond to the interior faces $ID_{cl} > -1$ or have all neighbors with $ID_{cl} == -1$ are simply discarded, the rest are considered for growing. In the case when an exterior boundary face can be added to more than one cluster we assign the face to a cluster with smallest ID, see Figure 5.9.

An initial cluster growing is finished if there are no exterior boundary faces left for further assignment, i.e. all fragments are discarded. An occlusion query can be used in this case to check if any fragment has been written, i.e. if any face ID_{cl} in $FaceInfo$ texture changed or not.

Cluster Optimization:

The GPU-based implementation steps of the Boundary-based cluster optimization, according to the Algorithm 5.1, are summarized in Algorithm 5.4.

Algorithm 5.4. *(GPU BB clustering steps)*

```
Loop while $samples$ != 0 {
    GatherClusterData()
    ComputeClusterProxy()
    samples = OptimizeBoundaryEnergy()
}
```

The *GatherClusterData()* subroutine is used to gather all necessary cluster data to compute the cluster proxy. To perform this, the *FaceInfo* texture is loaded into a vertex stream. In a vertex shader the corresponding face data is fetched for each vertex, i.e. face, from *FaceData* texture. This information is then scattered, using the index ID_{cl} of the face, to a correct cluster position in the *ClusterData* texture. Having the additive blending activated, the individual cluster's data is summed up from each face.

Using the data from *ClusterData* texture the cluster's proxy can be computed (*ComputeClusterProxy()*) and the information saved in the *ClusterProxy* texture.

The *OptimizeBoundaryEnergy()* subroutine is implemented as presented in the Algorithm 5.5. Where N_1, N_2, N_3, i.e. N_j, refer to the index ID_{cl} of the neighboring faces $(ID_{n1}, ID_{n2}, ID_{n3})$.

Algorithm 5.5. *(Optimize Boundary Energy)*

```
$faceWillGoToCluster = -1$; //set default to discard
$shrinkEnergy$ = DBL_MAX; //set to the maximum value

rasterize $FaceInfo$ texture
for each fragment $f$ { //represents a face
  $ID_{cl} = f.getClusterID()$;
  if ( $(N_1 == ID_{cl})$ && $(N_2 == ID_{cl})$ && $(N_3 == ID_{cl})$ ) discard;
  else{
    //we are on the cluster boundary
    for each neighbor $N_j$ {
      compute energies $^*E(P)^0$, $^*E(P)^1$, $^*E(P)^2$; //according to Eq. (5.3)
      if ( $^*E(P)^2$ smallest and smaller than $shrinkEnergy$) {
        $shrinkEnergy = {}^*E(P)^2$;
        $faceWillGoToCluster = N_j$;
      }
    }

    if( $faceWillGoToCluster == -1$ ) discard; //no change to the boundary
    else set $f[ID_{cl}] = faceWillGoToCluster$; //do the shrink operation
  }
}
```

To perform a cluster boundary optimization, see Figure 5.10, the *FaceInfo* texture is rasterized. For each fragment which corresponds to a boundary face the $^*E(P)^0$, $^*E(P)^1$, $^*E(P)^2$ energies are computed according to Eq.(5.3). The non-boundary corresponding fragments are simply discarded. Remember that as in the case of initial cluster growth the cluster can change its configuration only through boundary faces, i.e the interior boundary faces in this case.

The case with the smallest energy must be chosen and the configuration correspondingly updated. Thus if $^*E(P)^0$ is the smallest energy the fragment is simply discarded. If $^*E(P)^2$

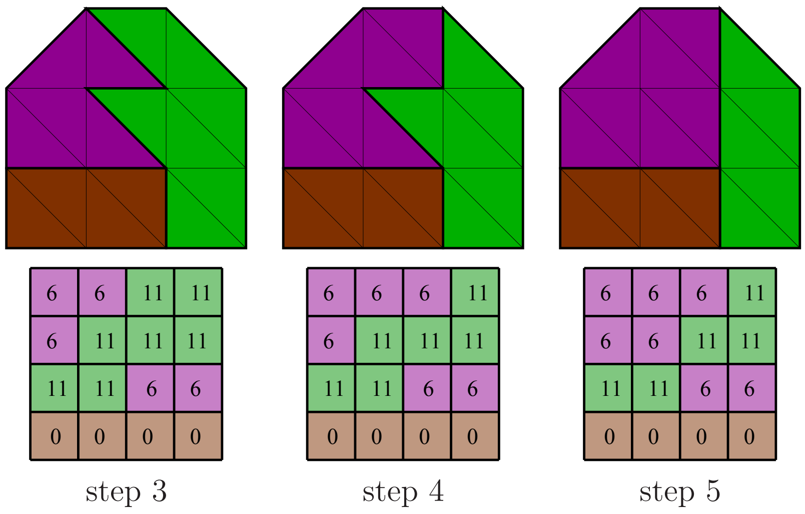


Figure 5.10: An example of a GPU-based BB cluster optimization. (Top) Clustered mesh. (Bottom) Shows how the face's ID_{cl} changes in *FaceInfo* texture.

is the smallest, i.e. cluster shrinks, then the fragment ID_{cl} is set to the ID of the opposite cluster. However, if ${}^{*}E(P)^{1}$ is the smallest energy, i.e. cluster must grow, the fragment is also discarded because the ID_{cl} for the neighboring face *can not* be reset from this point in the program. This limitation has *no* influence on the optimization, because any cluster growth can be seen as a shrink of the opposite cluster, i.e. required cluster growth will be performed by shrinking the opposite cluster.

After all fragments are processed, a new clustering configuration is obtained. This is used as a starting configuration to complete a new optimization step according to the Algorithm 5.4, i.e. gather cluster data, compute cluster proxies and apply a boundary optimization. This process is repeated until there is no change in the clusters configuration, i.e. no fragment changed its ID_{cl}. An occlusion query is used in this case to check how many fragments (*samples*) were written. An example of this process is depicted in Figure 5.10.

Note that, the bigger the clusters the more fragments corresponding to interior (non-boundary) cluster faces are discarded. Thus, this Boundary-based optimization process is very fast, see also the timing in Table 5.1.

5.2.4 Multilevel Mesh Clustering on GPU

The ML approach performs according to the Algorithm 5.2. As a starting configuration each mesh face is considered as an individual cluster. For a given clustering configuration the algorithm starts with the identification of Optimal Dual Edges (ODEs), see Definition 4.2. This is followed by the collapse operation, where *one* smallest cost ODE is collapsed.

However, to identify the cluster's ODE requires the Boundary Loop, Definition 3.1, or at least the boundary Half Edges (HEs) of the cluster, see Section 4.2.1. None of these elements are stored and thus available on the GPU.

Despite this, note that a boundary HE is one that is adjacent to two boundary faces, see Figure 5.11. On the GPU the boundary faces, see Definition 5.1, can be easily identified using the stored information. Thus, a Dual Edge (DE) can be generated for a boundary face if its neighboring face belongs to a different cluster. An example of this operation is depicted in Figure 5.11. Using this idea yields a very efficient DE generation on the GPU without storing any additional information.

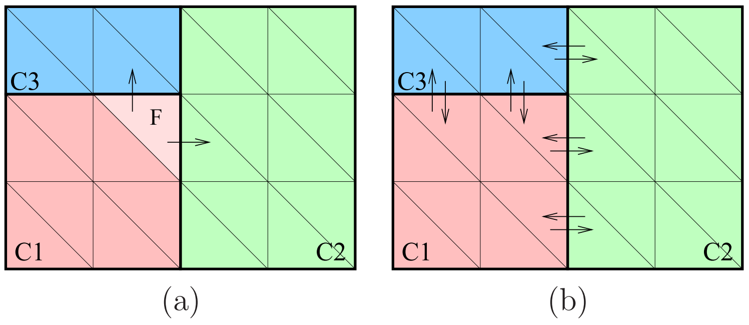


Figure 5.11: Example of a Dual Edge (DE) identification on the GPU. An arrow indicates a DE. (a) For a given face F two DEs are generated, because it has two adjacent boundary faces. (b) For a given clustering configuration 10 DEs are generated.

The Algorithm 5.6 shows the main implementation steps for ML approach on the GPU.

Algorithm 5.6. *(GPU ML clustering steps)*

```
Loop while numberOfCluster > 1 {
  findMinDE();
  collapseMinDE(); // numberOfCluster−−;
  minimizeEnergy(); //according to the Algorithm 5.4
}
```

In the *findMinDE()* subroutine one DE with the smallest merging cost out of all generated DEs is identified. A workflow of this process is given in Figure 5.12. The *FaceInfo* texture is loaded into a vertex stream. The vertex program (1) discards all vertices corresponding to the non-boundary faces. For vertices which pass this test, i.e. for each boundary face, a DE is computed/generated. A DE contains the (virtual) collapse cost $E_{r,s}$ and the indices (ID_r, ID_s) of two merging clusters. All generated DEs are scattered into a single pixel, see Figure 5.12. For each fragment, which correspond to a DE, the DE cost $E_{r,s}$ is set as a fragment depth and the indices (ID_r, ID_s) as a fragment color. Using a depth test, the smallest cost DE with corresponding (ID_r, ID_s) can be identified.

In the *collapseMinDE()* subroutine the collapse operation is applied, i.e two clusters referenced by (ID_r, ID_s) are merged. This operation is simple as it requires only reseting all faces with $ID_{cl} == ID_s$ to $ID_{cl} = ID_r$ in the *FaceInfo* texture.

In the *minimizeEnergy()* subroutine the obtained clustering configuration is optimized. For this the Boundary-based cluster optimization approach is applied, as described in Section 5.2.3.

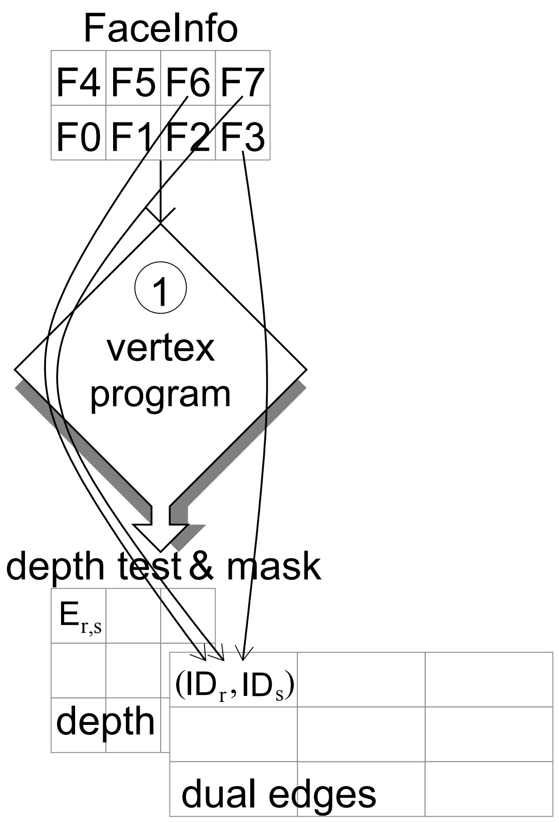


Figure 5.12: A workflow example for the Multilevel (ML) Dual Edge identification. Arrows indicate the DEs, $E_{r,s}$ denotes the energy for merging the cluster ID_r with cluster ID_s.

5.2.5 Parallel Multilevel Mesh Clustering on the GPU

For performing the parallel Multilevel (PML) clustering all mutual Dual Edges must be identified. This is in contrast to the ML approach where only one minimal cost DE is identified. Mutual DEs are pairs of DEs that connect the same clusters, e.g. if one DE merges cluster C_1 with cluster C_2 then the mutual DE will merge C_2 with C_1, see Figure 5.13 (b).

In general, identifying if two DEs are mutual can be done only after all DEs are generated. Thus, first, for each cluster the ODE is identified. This is followed by a collapse operation in which it is tested if two ODEs are mutual and, respectively, can be collapsed. The main implementation steps for the PML approach on the GPU are shown in the Algorithm 5.7.

Algorithm 5.7. *(GPU PML clustering steps)*

```
Loop while numberOfCluster > 1 {
  findODEForEachCluster();
  collapseAllMutualODEs();
  minimizeEnergy(); //according to the Algorithm 5.4
}
```

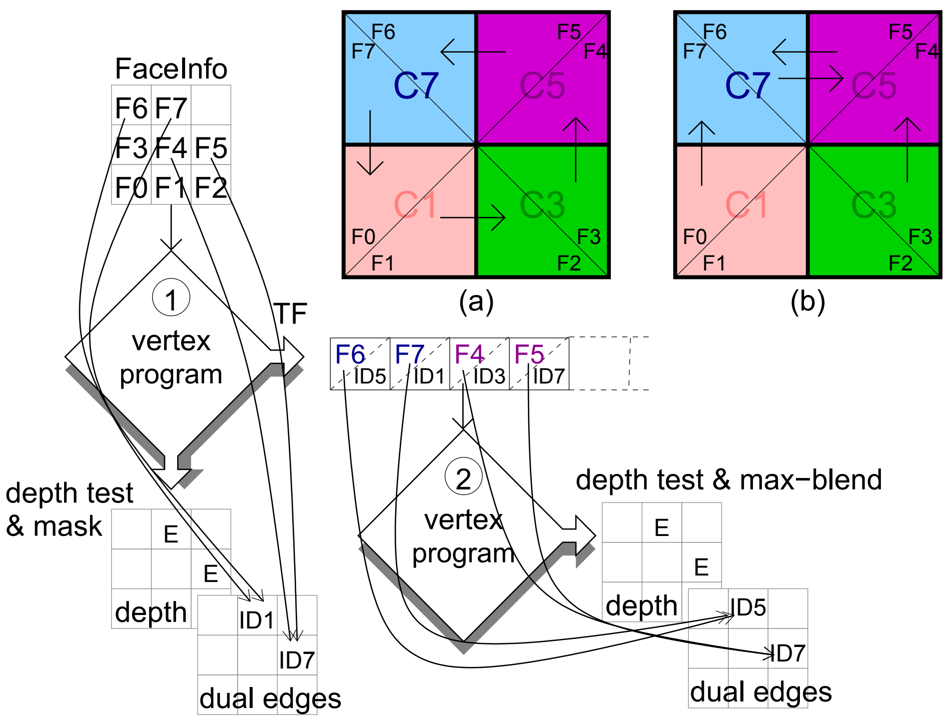


Figure 5.13: A workflow example for Parallel Multilevel (PML) dual edge identification. (a)-(b): A mesh consisting of 8 faces and clustered in 4 clusters C_i. Arrows indicate the DEs, E denotes the merging energy, and ID_n denotes the opposite merging cluster index. Note, that applying the second pass results in case (b) with two mutual DEs instead of case (a) with no mutual DEs.

In the *findMinDEForEachCluster()* subroutine for each cluster a DE with the smallest merging cost out of all generated DEs for this cluster is identified, i.e. an Optimal DE (Definition 4.2). The workflow in this case is identical to the ML clustering, see Figure 5.12. Where the vertex scattering (vertex program (1)) with depth test activated, i.e. *glEnable(GL_DEPTH_TEST)* and *glDepthMask(GL_TRUE)*, is applied. However, with the difference that the DEs are not scattered into only one pixel but to different pixels which correspond to the boundary face's ID_{cl}, i.e. which correspond to different clusters ID, see Figure 5.13. Remember from last section that the number of generated DEs for each cluster is approximately equal to the number of boundary faces. Now, each of these DEs must be scattered to its individual cluster position, and the one with smallest cost, i.e minimal depth, will be reported.

However, there can be cases where no mutual ODEs exist although there are possibilities for merging, see for an example Figure 5.13 (a). This mostly happens in the regions with identical merging cost, because here the direction of the merge is arbitrary. The same problem was pointed out indirectly in [WLR88], where it is proposed to select the neighbor with the largest ID. In our implementation we adopt the same selection rule.

To have this selection implemented efficiently we propose to use the transform feedback (TF) feature of the GPU to read back all generated DEs without recomputing them again, see the sketch in Figure 5.13. These DEs can be scattered (vertex program (2)) to the correct cluster position exactly as done in the (vertex program (1)). However, to choose a neighbor with the largest ID, some additional settings are required for this second pass:

1. We set no depth buffer update, i.e. *glDepthMask(GL_FALSE)*, and set the depth comparison function to *glDepthFunc(GL_EQUAL)*. Thus, using the depth buffer data to test and consider only the DEs which have identical minimal cost, i.e depth, as that already saved in the depth buffer.

2. Apply a maximum blending to the ODE's cluster ID. Thus, from all fragments that correspond to DEs with minimal cost only one with maximum cluster ID will be saved for each cluster in part.

This way, at least one pair of mutual DEs can be obtained, see Figure 5.13 (a) - (b).

After all ODEs are computed, the mutual ones need to be identified and collapsed in *one* collapse step. This is implemented in a subroutine *collapseAllMutualODEs()* as described in Algorithm 5.8. Note that, if two mutual ODEs exist we have to apply the merging only in one direction, i.e. avoid any double resettings. The order in this case is not important, and we let only clusters with largest ID to apply merging.

Algorithm 5.8. *(GPU Collapse All Mutual ODEs)*

```
Texture FaceInfo;//contains the current clustering, i.e. $ID_{cl}$ for each face
Texture ClusterODE; //contains ODE for each cluster

rasterize FaceInfo texture
for each fragment $f$ { //represents a face
  $ID_{cl} = f.getClusterID()$;
  mergingClusterID_1 = get data at $ID_{cl}$ in ClusterODE;
  mergingClusterID_2 = get data at mergingClusterID_1 in ClusterODE;

  if( mergingClusterID_2 != $ID_{cl}$ ) discard; //no mutual ODEs

  //choose only largest ID clusters to perform the collapse
  if( $ID_{cl}$ < mergingClusterID_1 ) discard;

  $ID_{cl}$ = mergingClusterID_1; //perform the collapse
}
```

5.3 GPU-based Mesh Clustering Results

All the results presented in this and the next chapter are generated using a 3GHz Intel Core(TM)2 Duo CPU PC with a GeForce GTX 280 (1024MB) graphics card.

To evaluate the algorithms presented in this chapter we show how an approximated Centroidal Voronoi Diagram (CVD) can be performed. This energy functional is chosen mostly due to the fact that it can be easily implemented and because it requires few cluster data.

The energy of an approximated CVD can be written as, see Section 3.1.3:

$$E_{CVD} = \sum_{i=0}^{k-1} E_i = \sum_{i=0}^{k-1} \sum_{F_j \in C_i} \rho_j \|\boldsymbol{\gamma}_j - \overline{\boldsymbol{\gamma}}_i\|^2. \tag{5.4}$$

where $\boldsymbol{\gamma}_j$ and ρ_j is the centroid and the weighted area of the face F_j, respectively. $\overline{\boldsymbol{\gamma}}_i = \sum_{F_j \in C_i} \rho_j \boldsymbol{\gamma}_j / \sum_{F_j \in C_i} \rho_j$ is the cluster centroid (proxy). Thus for computing the ${}^*E(P)^0$, ${}^*E(P)^1$, ${}^*E(P)^2$ energies in Eq. (5.3) one uses:

$$E(F_s, P_r) = \rho_s \|\boldsymbol{\gamma}_s - \overline{\boldsymbol{\gamma}}_r\|^2. \tag{5.5}$$

The Dual Edge collapse cost between two clusters C_1 and C_2 is computed as $DE_{cost} = E_{12} - E_1 - E_2$, see Section 4.1. For an easier computation of the merging energy, Eq. (5.4) can be written in the form:

$$E_i = \sum_j \rho_j \|\boldsymbol{\gamma}_j\|^2 - 2\overline{\boldsymbol{\gamma}}_i \cdot (\sum_j \rho_j \boldsymbol{\gamma}_j) + \|\overline{\boldsymbol{\gamma}}_i\|^2 \sum_j \rho_j.$$

Thus for each face in the *FaceData* texture we only need to keep the values $\rho_j \|\boldsymbol{\gamma}_j\|^2$, $\rho_j \boldsymbol{\gamma}_j$ and ρ_j. Correspondingly, the *ClusterData* texture stores for each cluster the following information: $\sum_j \rho_j \|\boldsymbol{\gamma}_j\|^2$, $\sum_j \rho_j \boldsymbol{\gamma}_j$ and $\sum_j \rho_j$.

Boundary-based GPU Mesh Clustering Results:

Figure 5.14 depicts the result of a CVD construction for the Armadillo model. Observe how during optimization the clusters (the green cluster is the most prominent) move from the left leg to different parts of the model. This shows that the Boundary-based algorithm performs very well even when starting with a “bad” initialization. At the same time observe the perfect symmetry (good visual quality) in the final clustering.

Figure 5.15 shows the CVD energy behavior for the BB algorithm compared to the EMLO algorithm (Section 3.1.4) for different number of clusters between $2k$ and $1k$. Note that using the Boundary-based approach a slightly higher energy is obtained compared to the EMLO algorithm. This is due to the fact that the BB algorithm optimizes the complete boundary at once in parallel, where the EMLO algorithm has a more selective boundary optimization as it reupdates the cluster data after each local optimization.

Similar behavior was observed for most of the tested models, although for some models and specific number of clusters the results are identical. Such an example is the clustering

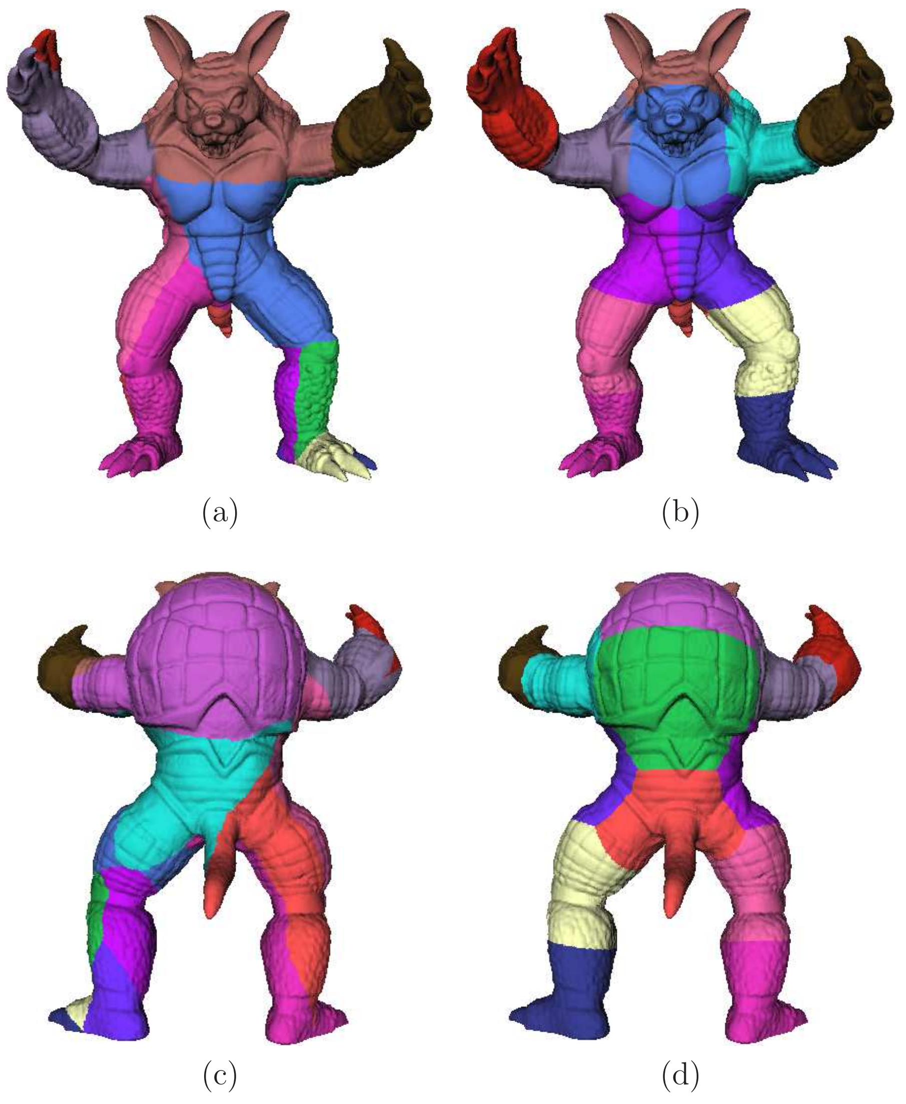


Figure 5.14: A CVD construction for 15 clusters. (a) & (c) Results of the initialization. (b)&(d) Results after applying Boundary-based mesh clustering.

Model	# Faces	# Clusters	CPU *EMLO* (sec.)	GPU *BB* (ms)	Speedup Factor
Bunny	70k	1k	3	172	17
Bunny	70k	3k	3	187	16
Horse	97k	1k	3	187	16
Horse	97k	2k	4	218	18
Armadillo	346k	2k	12	1140	10
Armadillo	346k	5k	16	891	18

Table 5.1: Clustering time for building a uniform CVD with CPU- vs. GPU Boundary-based (BB) cluster optimization.

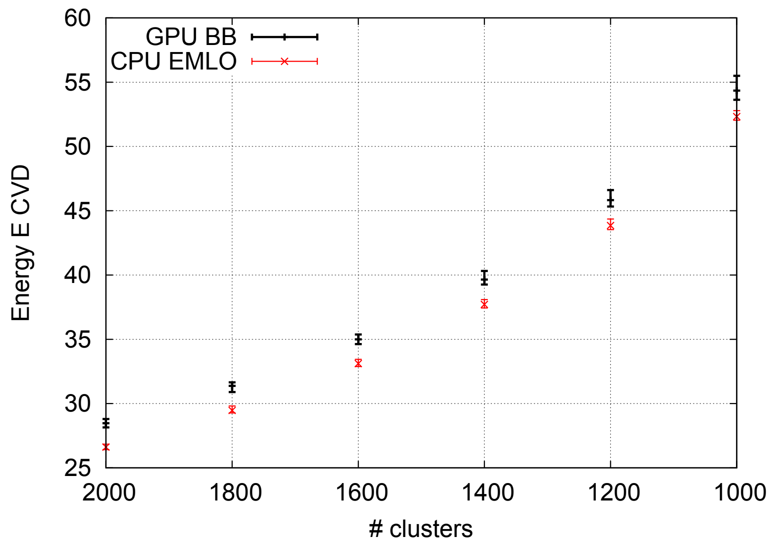


Figure 5.15: CVD energy as a function of the number of clusters for the Bunny model. (BB) Boundary-based; (EMLO) Energy Minimization by Local Optimization (Section 3.1.4). To obtain the energy variation limits the algorithms are performed 100 times with different random initializations.

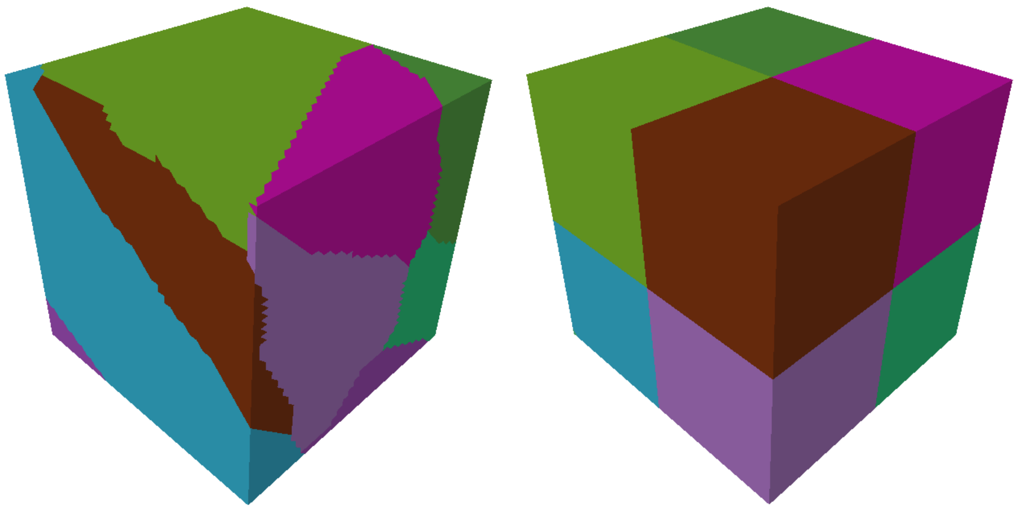


Figure 5.16: A GPU-based CVD clustering for 8 clusters. (a) Result of the initialization. (b) Result after applying Boundary-based mesh clustering.

of a cube model with 8 clusters, as presented in Figure 5.16. For this configuration both algorithms provide the same clustering result.

Table 5.1 provides a timing comparison between CPU- and GPU-based clustering results, i.e. for the EMLO and the BB approaches, for different meshes. Here considerable speedups from 10 to 18 times are observed.

Multilevel Mesh Clustering Results:

Figure 5.17 shows the CVD energy variation for GPU- vs. CPU-based clustering algorithms for different number of clusters between $2k$ and $1k$ for the Bunny model.

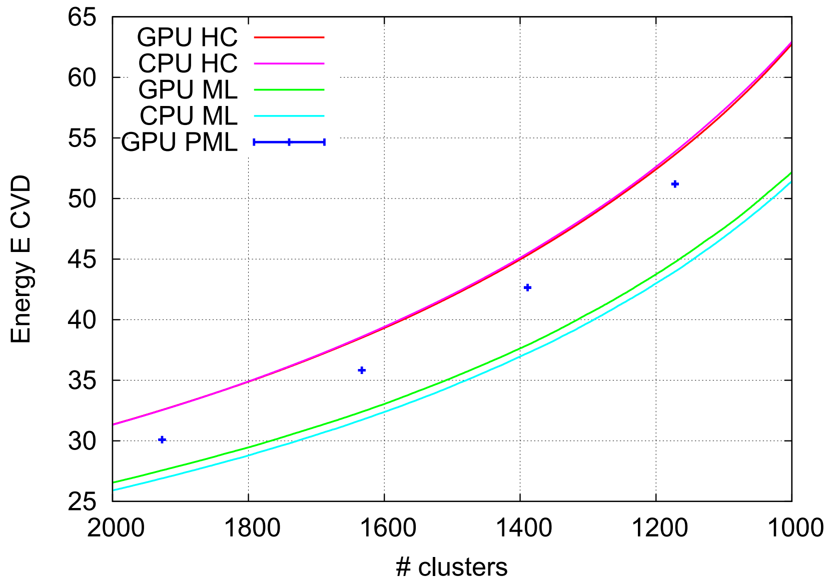


Figure 5.17: CVD energy versus number of clusters for Bunny model. (HC) Hierarchical clustering; (ML) Multilevel; (PML) Parallel ML.

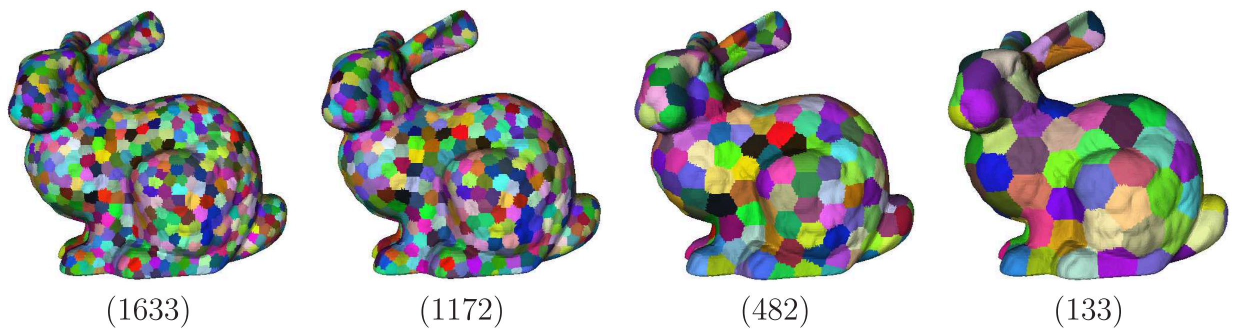


Figure 5.18: A CVD GPU-based PML clustering result for different number of clusters.

Model	CPU, ML (sec.)	GPU, ML (sec.)	Speedup Factor
Bunny	281	183	1.5
Horse	710	283	2.5
Armadillo	10856	6633	1.6

Table 5.2: Clustering time for Multilevel (ML) mesh clustering. The results are given using a CPU and respectively a GPU based implementation.

Model	CPU, ML (sec.)	GPU, PML (sec.)	Speedup Factor
Bunny	281	9	31
Horse	710	5	142
Armadillo	10856	73	149

Table 5.3: Clustering time for GPU-based Parallel Multilevel (PML) vs. to the CPU ML mesh clustering.

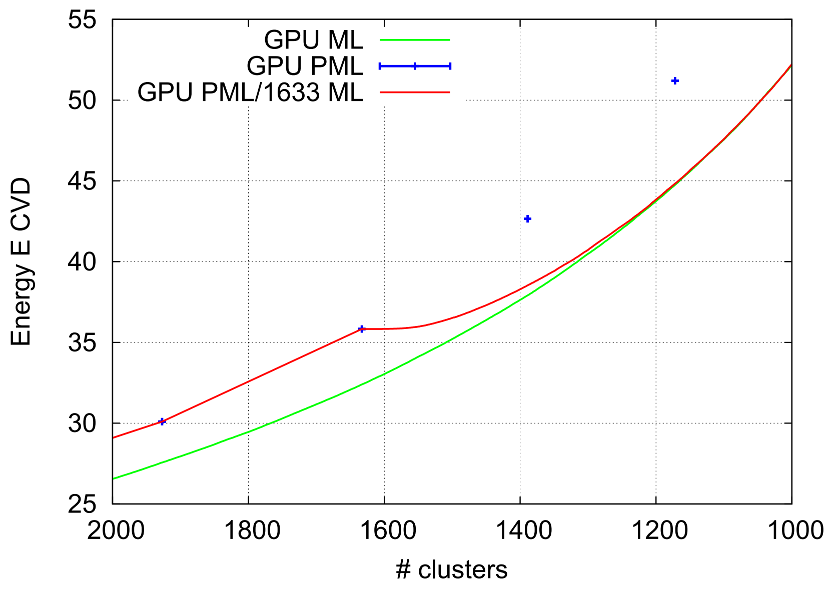


Figure 5.19: CVD energy as function of the number of clusters for the Bunny model. (ML) Multilevel; (PML) Parallel ML. (PML/1633 ML) Performing PML up to 1633 clusters, then applying ML clustering.

As expected, the hierarchical clustering energy, which is obtained by deactivating the optimization, is identical in both cases, see Figure 5.17 (GPU HC & CPU HC). This is because both algorithms must perform mergings in the same order.

However, the energy of the (GPU ML) compared to the (CPU ML) approach is slightly higher, see Figure 5.17. This is due the optimization behavior of the Boundary-based algorithm which is used in the optimization phase, as presented in Figure 5.15.

Figure 5.17 also depicts the energy of the GPU-based parallel ML (PML) algorithm. The PML energy is lower than that of the hierarchical clustering but higher than that of the ML algorithm. Note also that there are approximately 200 cluster mergings in one merging step. Figure 5.18 shows, correspondingly, the GPU-based PML clustering result for Bunny model for different number of clusters.

Table 5.2 provides a timing comparison between CPU- and GPU-based clustering results for ML approach. Due the fact that only one Dual Edge is collapsed in each step for the GPU ML clustering a speedup factor of only 1.5 to 2.5 is achieved.

However, using the GPU-based parallel ML clustering approach speedup factors of 30 to 150 can be achieved, as presented in Table 5.3.

It is important to note that, as in the case of the CPU-based ML algorithm, see Section 4.5, different variants of the GPU-based ML algorithm are possible. The PML algorithm is very fast, but the clustering result is worse compared to ML. In this case one can combine both algorithms to speedup the ML approach with no quality sacrifice. Figure 5.19 depicts such an example. If the ML algorithm is activated after the PML, the energy rapidly converges against the standard ML variant. Here the PML approach was applied up to 1633 clusters, followed by the ML clustering. Note that, the clustering energy from 1200 clusters is identical to that of the ML clustering. Thus for a specific region of interest, regarding the number of clusters (in this example from 1200 to 1 clusters), this variation of the algorithm can be used to perform a fast ML construction.

5.4 Conclusions

In this chapter we have described new mesh clustering concepts, which provide a parallel redefinition of the existing standard approaches. Thus, we reviewed the Variational and the hierarchical (or in our case the Multilevel approach because it is more generic) algorithms and proposed the Boundary-based and the parallel ML mesh clustering algorithms. The Boundary-based approach is parallelizable and can accept any proxy-based energy functionals, thus allowing for a wider range of applications. The proposed parallel ML mesh clustering approach is very flexible. It has no limitation in the way in which parallel ML clustering can be performed.

The major algorithmic elements are *boundary-based queries*, which strongly incorporate the spatial coherence present in the optimization and the cluster merging steps. Our formulation is free from any global data structures. Thus, it provides all necessary ingredients for a GPU-based implementation.

We showed how both concepts can be *entirely* implemented on the GPU and also gave some non-trivial GPU-specific technical details. To perform the clustering on the GPU, we proposed a new mesh connectivity encoding. This gives all necessary means for a complete Variational, Hierarchical or Multilevel GPU-based mesh clustering. It must be noted that, this is the first GPU-based mesh clustering framework which employs mesh connectivity in the clustering process.

We tested the approach by building a Centroidal Voronoi Diagram. Using this framework we showed that considerable speedup can be obtained.

This GPU-based framework constitutes an important building block for an efficient mesh clustering. Together with the Multilevel approach, which we discussed in the previous chapter and which proved to provide high quality clustering results, they constitute the main tools to perform efficient and high quality mesh clustering.

Chapter 6

GPU-based Data Clustering

Generalizing the *Multilevel (ML)* mesh clustering concept described in the previous two chapters to general data clustering is tempting in several ways:

1. As shown in Chapter 4 for meshes, the ML approach resolves the inherent problems of the standard iterative and hierarchical algorithms. For iterative (k-means) approaches the ML solves the dependency of the result on the initial number and selection of seeds, which strongly affects the quality of the result. For hierarchical approaches the non-optimal shape of the clusters in the hierarchy, which is due to the strict containment property in the cluster hierarchy, is also overcome. Thus, any application which uses the standard hierarchical or k-means clustering algorithms can profit, i.e. provide higher quality results, when using the ML construction.

2. There is a rapid growth not only in the amount but also in the dimensionality of the data to be clustered. Thus, fast processing and clustering of these large data sets is an essential task for many applications. Realizing the ML data clustering on the GPU, as done for meshes [CKCL09], will make accelerated and scalable clustering available for a large number of applications.

In this chapter we present a generalization of the Multilevel method to data clustering and its GPU-based implementation[1]. All advantages of the mesh-based technique conveniently carry over to the generalized data clustering approach.

In Section 6.1 we show how the problem of the missing topological information, which is inherent to general data clustering, is solved by dynamically tracking cluster neighborhoods. This tracking allows the identification of cluster neighborhoods, required for cluster merging and for cluster optimization, without enforcing a disadvantageous storage of per-element neighborhoods. This leads to a new *Local Neighbors* k-means algorithm, described in Section 6.2. In Section 6.3 we describe how Multilevel data clustering can be performed. The proposed approaches provide all necessary ingredients to support a GPU-based implementation, the details of which are described in Section 6.4. Finally, Section 6.5 and Section 6.6 presents different evaluation results. Section 6.7 draws some final conclusions.

[1]Parts of this chapter were published in [CK11].

6.1 Neighborhood Identification and Tracking

Realizing a GPU-based Multilevel data clustering for "point clouds" comprises significant challenges:

1. There is no "natural" neighborhood for the data elements as for the faces in a mesh.

2. As a consequence to this, there is no concept of a cluster boundary which is required for the Boundary-based and ML clustering approaches. Thus, any algorithm described in the last two chapters can not be carried over to data clustering directly.

Now, regarding the missing neighbors, one could reconstruct this neighborhood by identifying the k nearest neighbors for each element [GDB08], see Figure 6.1. However, doing this would require a static storage of this information on a per-point level. The computation and storage of this information on the GPU will most probably have a negative impact on the performance and scalability.

Furthermore, there is even a more severe problem with this idea. A careful analysis shows that the identification of an appropriate value for k is rather difficult and is even data dependent. Figure 6.1 (a) shows such an example for $k = 3$. Due to missing neighbor information no merging between the final two clusters in the ML construction is possible. Note that this may happen at any level in the ML construction. Increasing k, see Figure 6.1 (b), may solve the problem for this specific case. However, using a larger k is no longer appropriate for the case presented in Figure 6.1 (c).

To deal with these problems, we propose a new neighborhood identification approach based *only* on spatial relations between clusters, thus realizing a kind of "indirect" point-cluster neighborhood. This is in contrast to the mesh clustering approach where per-face, i.e. per-point, neighbor information is used.

Remember that in the case of a polygonal mesh, an Optimal Dual Edge (ODE) is identified by looping over the Boundary Loop (BL) of the cluster, see Figure 6.2 (a). A DE with smallest cost from all DEs is then chosen as an ODE. In the case of a discrete point set no connectivity is present, thus there is no BL of the cluster to proceed in the same way.

However, we can use the same analogy as for meshes to identify the cluster's neighborhood on a point set. For a given mesh, a loop over the BL of the cluster (Figure 6.2 (a)) can be seen as sweep of the 2D space and a search for possible neighbors. Thus, if we subdivide the point set space into a fixed number of subspaces, see Figure 6.2 (b), and try to identify in each of these subspaces the closest cluster as neighbor we can obtain approximately the same effect as for meshes, Figure 6.2 (c).

To establish a cluster neighborhood for the general case, we propose to subdivide the space at the centroid $\overline{C_i}$ of cluster i into $2d$ subspaces $S_n^i,\ n = 1, \ldots, 2d$ in the d-dimensional case according to:

$$S_n^i = \{Q \ : \ 2m(Q,i) - ((Q - \overline{C_i})_{m(Q,i)} > 0) = n\}, \tag{6.1}$$

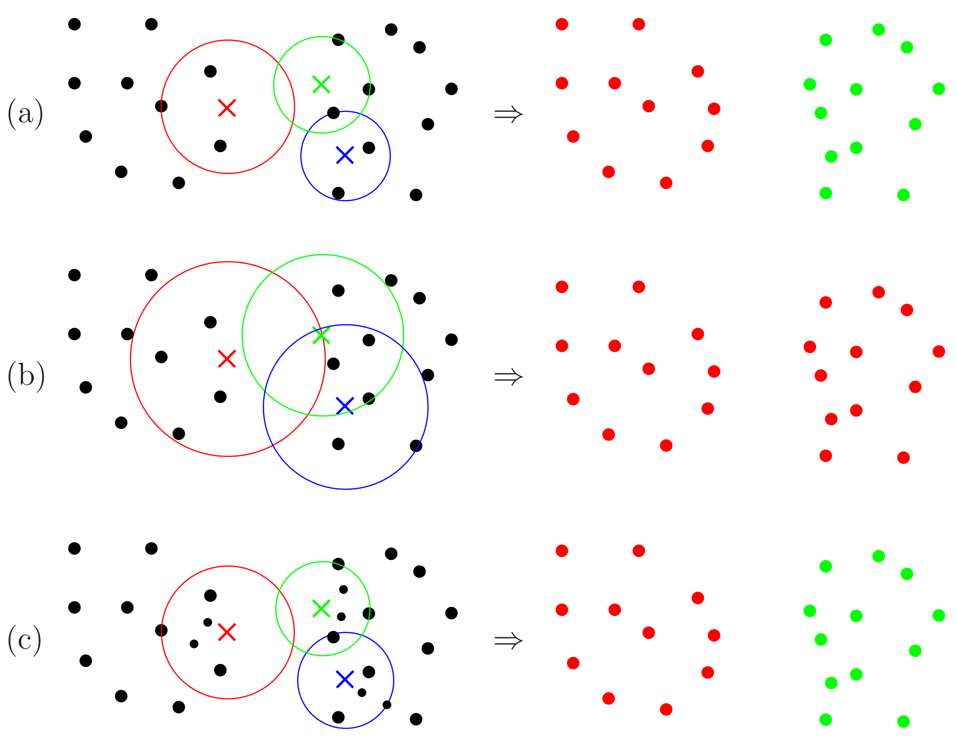


Figure 6.1: Left: Example of identified k nearest neighbors (points inside each circle) for: (a) $k = 3$, (b) $k = 5$ and (c) $k = 5$. Corresponding query data points are represented by $\times$. Right: Final results of the ML construction. (a) & (c) The final two clusters cannot be merged into one, because no neighbors connect them.

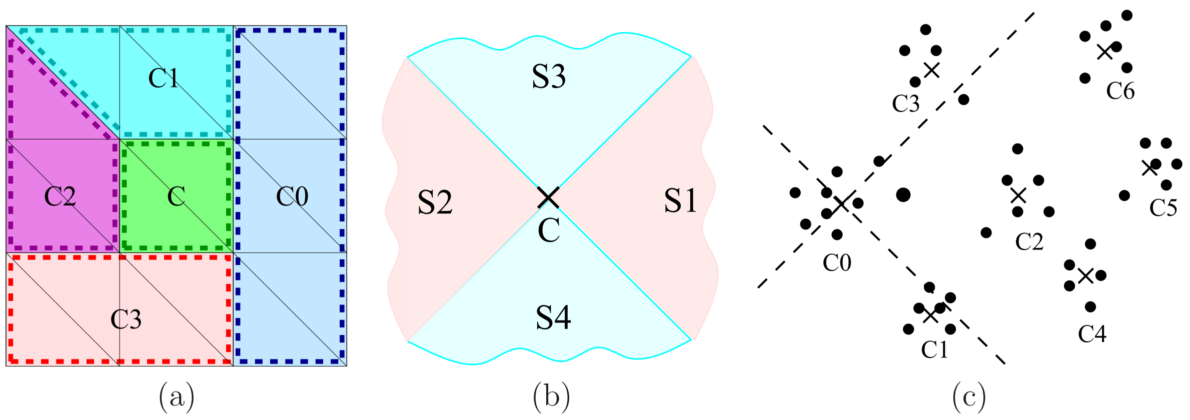


Figure 6.2: A 2D case: (a) Cluster C with its Boundary Loop (dashed line). (b) Space subdivision with respect to a given cluster centroid C. (c) The ODE identification. The local neighbors of the cluster C_0 are $N_1^0 = C_2, N_2^0 =$null, $N_3^0 = C_3, N_4^0 = C_1$.

where $m(Q,i) = \arg\max_{j=1,\ldots,d}(\left|Q - \overline{C_i}\right|_j)$, and $((Q - \overline{C_i})_{m(Q,i)} > 0)$ is a boolean expression.

For each of these subspaces the closest cluster is identified and referenced as *local neighbor* N_n^i, where i refers to the cluster C_i for which the neighbors are identified and n to the subspace index. Note, that there may be no closest cluster in a region S_n^i, in this case N_n^i is a null reference, for an example see Figure 6.2 (c).

With such a neighborhood definition, the ODE of a given cluster C_i is easily obtained by checking only its local neighbors N_n^i from each subspace, see Figure 6.2 (c). This yields a very fast, i.e. efficient, ODE identification. Additionally, it overcomes the problem of the k nearest neighborhoods described and exemplified in Figure 6.1.

As a result each cluster has a fixed number of maximum $2d$ neighbors, which must be updated after any operation that changes a given cluster configuration, e.g. cluster merge or optimization.

A cluster local neighbors update, implemented as the *UpdateClusterLocalNeighbors()* subroutine, is done by considering all second order neighbors, i.e. all current neighbors and their neighbors. Each of these neighbors is first checked to find the subspace in which it is located, according to Eq. (6.1). It is assigned as a local neighbor if it is the closest to a considered cluster.

One may argue, that the specific neighborhood tracking may not catch up with the cluster motion and thus "better" assignment of neighbors may get lost. However, since the cluster motion slows down rapidly after a few optimization cycles and because we use for update not only the neighbors of a given cluster but also all second order neighbors, it is expected that the neighborhood information will always correct itself perfectly. Even though there is no guarantee for this, we did not encounter any false clustering result due to (possibly partially existing) incorrect cluster neighborhoods.

6.2 Local Neighbors K-Means

Although the standard k-means algorithm (Section 2.2.1) is parallelizable and can be directly implemented on the GPU, as we show in Section 6.4.4, this is still a brute-force k-means algorithm. More precisely, the classical k-means will perform $k \cdot m$ distance computations and comparisons for k clusters and m points. However, it can be easily observed that in the clustering process a given data point Q most probably will be reassigned only to its local neighboring clusters, see the example in Figure 6.3. Here, the point Q should be checked only against clusters C_0, C_1, C_2 and C_3 but not against C_4, C_5 and C_6. This observation is based on spatial coherence, similar to the Boundary-based algorithm proposed in Section 5.1.1, where a boundary triangle is checked only against its neighboring clusters. Thus, an efficient data clustering algorithm must use this spatial coherence in the clustering process.

Based on this observation, we propose a new algorithm, as a counterpart to the classical brute-force k-means, namely the *Local Neighbors k-means*.

Suppose that a set of m discrete points Q_j is provided. Clustering this point set into k

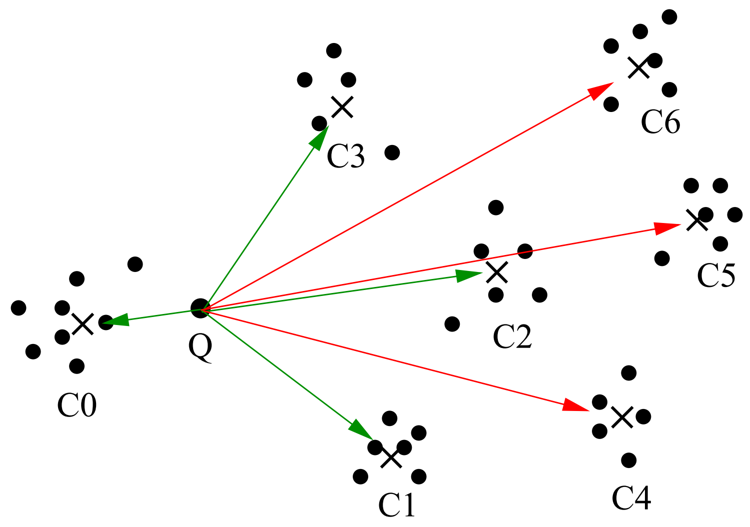


Figure 6.3: An efficient reassignment check. The green and the red arrows indicate the clusters that need and that do not need to be checked, respectively.

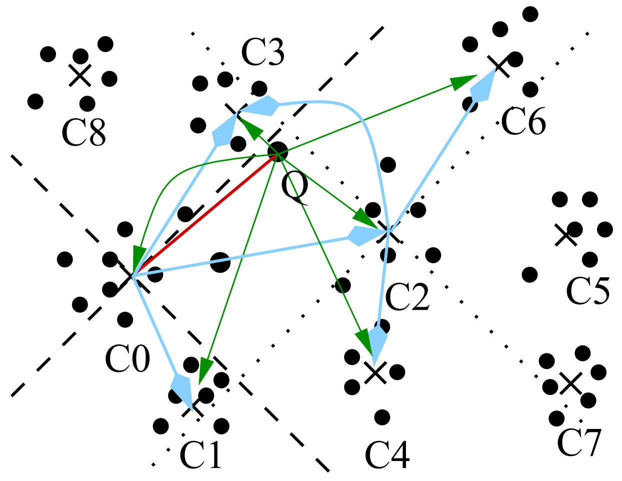


Figure 6.4: 2D example of the required checks in order to reassign the point Q to a closer cluster. The green arrows indicate the clusters which must be checked, the red line shows the current assignment of Q, the blue arrows indicate the 2-neighboring clusters.

clusters means partitioning the set such that points belonging to one cluster are "closer" by some energy measure to this cluster than to any other cluster.

Now suppose that an initial configuration is provided, where each point Q_j is already assigned to some cluster not necessarily the closest, see Figure 6.4. According to this configuration each cluster C_i has m_i points and a centroid $\overline{C_i}$. Additionally, using the neighborhood definition proposed in Section 6.1, each cluster C_i has a reference to its $2d$ local neighboring clusters N_n^i.

The basic idea is to optimize a given clustering configuration by checking for each point

Q only the “relevant” subspaces and the respective neighboring clusters, thus reducing significantly the number of clusters to be checked. The steps of this approach are summarized in Algorithm 6.1.

Algorithm 6.1. *(Local Neighbors k-means)*

```
while no point reassignment happens
{
  //update clusters data
  ComputeClusterCentroid();
  UpdateClusterLocalNeighbors();

  foreach point $Q$ in the data set
  {
    // check current cluster of $Q$
    $C_i$ = cluster containing $Q$
    check($Q$, $C_i$);

    // clusters in the subspace of $Q$ w.r.t. $C_i$
    // and its neighboring subspaces
    $S^i_k$ = subspace containing $Q$ in cluster $C_i$
    foreach subspace $S^i_l$ of $C_i$
      if ($S^i_k$ not opposite to $S^i_l$ AND $N^i_l$ != null ) check($Q$, $N^i_l$);

    // clusters in the subspace of $Q$ w.r.t. $N^i_k$
    // and its neighboring subspaces
    $C_j = N^i_k$ = neighboring cluster for $C_i$ in $S^i_k$
    if ( $C_j$ == null ) continue;
    $S^j_m$ = subspace containing $Q$ in cluster $C_j$
    foreach subspace $S^j_n$ of $C_j$
      if ( $S^j_m$ not opposite to $S^j_n$ AND $N^j_n$ != null ) check($Q$, $N^j_n$);

    // final assignment
    assign $Q$ to the cluster with least energy
  }
}
```

The *ComputeClusterCetroid()* and *UpdateClusterLocalNeighbors()* subroutines recompute the cluster centroids and update the local neighbors of each cluster (Section 6.1), respectively. The *check(Q_j, C_i)* subroutine computes the energy of point Q_j with respect to a cluster C_i and references the cluster with the least energy so far. The process is alternately repeated until no point reassignment happens. The result, as in the case of the k-means algorithm, is a clustering that minimize the total energy E.

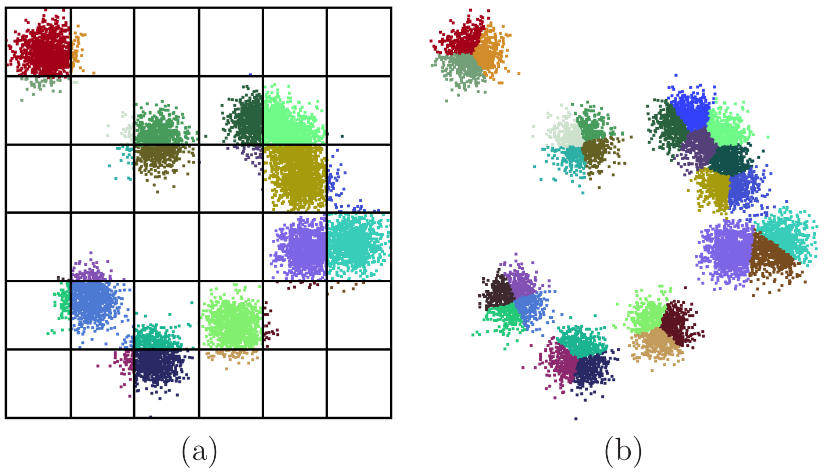


Figure 6.5: 2D example of a Local Neighbors k-means clustering. (a) Initialization. (b) Final clustering result.

Note, that this approach checks *only* relevant neighboring clusters in the same subspace S_k^i as the point Q or in the corresponding neighboring subspaces. The opposite subspace is omitted. Furthermore, the neighbors of cluster N_k^i are checked, if this cluster exists. Again, the subspace opposite to the one containing Q is omitted, resulting in at most $4d - 1$ clusters to be checked for each point. Thus, for $k \gg d$, this algorithm performs significantly less checks than the brute force variant of the k-means and the checks are still independent from each other, i.e. they can be performed in parallel. Note that, in the case of a specific neighborhood situation, clusters may be checked several times, as it can be seen in Figure 6.4. Here, the point Q is finally assigned to cluster C_3.

Up to now, we assumed that the initial configuration is given, i.e. each point is assigned to a specific cluster and each cluster has reference to its local neighbors. To have a valid and a good initial configuration we propose to do the initialization as follows, see Figure 6.5 (a) for an example:

1. Subdivide the bounding box spanned by the data points into $l = a^d$ uniform voxels, i.e. a voxels in each dimension.

2. Scatter points to corresponding voxels.

3. Consider each voxel as a starting cluster, with side voxels as local neighbors.

4. Merge zero sized clusters with non zero sized clusters.

Based on this initialization, the neighbors are directly assigned as the $2d$ voxels, i.e. clusters, sharing a common face. As a result we obtain a valid initial configuration where each point is assigned to a cluster and each cluster has $2d$ local neighbors.

Performing such an initialization has visible advantages over the random initialization of the k-means algorithm. The starting seeds are distributed uniformly and in some way simulates the farthest-point initialization heuristic. Such an initialization always leads to the same clustering result with no fluctuations and will give a number of clusters equal to a^d minus zero sized clusters, where a is the step-size. Thus, if a specific number of clusters k is desired, the Multilevel approach must be used to obtain the level k, see next section. Currently, a fixed step-size for all dimensions is used, which could be improved using adaptive techniques.

6.3 Multilevel Data Clustering

In the hierarchical phase of the Multilevel construction an Optimal Dual Edge is collapsed, i.e. two clusters with minimal merging cost are merged. A brute-force implementation of this step in the case of data clustering requires k^2 computations and comparisons for k clusters. However, it can be again easily observed in Figure 6.6 that a cluster most probably merges with one of its closest (local) neighbors. For example C_0 should be checked for merging only with C_1, C_2 and C_3 but not with C_4, C_5 and C_6. Thus, as in the case of the k-means algorithm, an efficient implementation of this step for ML data clustering must make use of the neighboring information of each cluster.

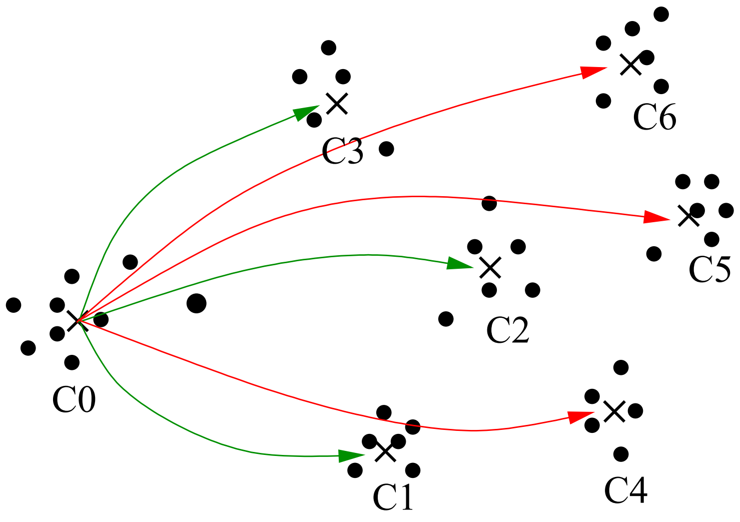


Figure 6.6: More efficient Optimal Dual Edge identification. The green and the red arrows indicate the clusters that need and that do not need to be checked, respectively.

The neighborhood definition proposed in Section 6.1 provides the base for performing the ODE identification efficiently. For a given cluster an ODE is identified by checking only its $2d$ local neighbors and choosing one with the smallest merging cost. This approach obeys the spatial coherence in the process of cluster merging, i.e. only spatially near clusters can merge.

Thus, the ML (Algorithm 5.2) and parallel ML (Algorithm 5.3) mesh clustering approaches smoothly carry over to data clustering. All these techniques can be applied in a flexible way, e.g. optimization can be deactivated to perform a classical hierarchical data clustering, or one can switch between sequential ML and PML.

The initialization performed for the Local Neighbors k-means, see Section 6.2, already yields a-priori well shaped clusters, if the classical Euclidean distance measure is used as energy functional. Thus, starting with a very fine step-size and merging without optimization or in parallel, i.e. using PML, yields very similar results compared to starting with a larger step-size, providing a configuration where the number of points in each voxel is $\gg 1$.

Therefore, to obtain all our results in Section 6.5 we use a variant of a Multilevel approach that starts with a larger step-size a and applies only the sequential ML clustering afterwards. In this case, the expectation is that the energy behavior will be similar to that depicted in Figure 5.19 on page 115 for mesh clustering.

6.4 GPU Implementation Details

In this section we describe GPU-specific implementation details for the algorithms proposed in Section 6.2 and 6.3. As for meshes, we aim at performing the complete clustering entirely on the GPU, reducing any data transfer between CPU and GPU to a minimum.

With small changes, the techniques for GPU-based mesh clustering carry over to GPU-based data clustering. Most of the implementation details described in Section 5.2 apply equally to data clustering. Thus, the reader who is not familiar with these is referred to read Section 5.2 first. In this chapter we address only the new elements which are related to data clustering.

6.4.1 Data Representation and Processing

As for meshes, on the GPU a *per-point* and a *per-cluster* processing applies. Correspondingly, 2D 32-bit textures are used to store the associated *PointData* and *ClusterData*.

Given a d-dimensional data set with m points, we save the input data set in an image based *pfs* format [MKMS07]. This is done in order to obey the same input format as for meshes. The number of channels in the *pfs* file corresponds to the d dimensions of the data set. Note that no additional information besides this is saved in the *pfs* data file.

In contrast to meshes which are 3D, the data usually tend to be high dimensional, i.e. $d > 4$. To cope with this, the Multiple Render Targets (MRT) mechanism is used to store and process any point or cluster data information. Current hardware can support up to 8 RGBA targets. Thus, theoretically up to 32-dimensional data can be stored and processed using $ceil(d/4)$ render targets. Since each cluster needs to store $2d$ neighbors, we can only process data up to 16 dimensions in the current implementation.

However, it must be recognized that this limit only applies to the current implementation. Thus, if the neighbors of each cluster are saved in two texels instead of one, i.e. doubling the size of the *NeighborsInfo* texture, then up to 32 dimensions can be supported. Interestingly, this technique can be used to deal with even higher number of dimensions.

Now, regardless of whether the data point is clustered or not, each data entity belongs to a specific cluster with index ID_{cl}. In the beginning all data belong to the "null" cluster, i.e. they have $ID_{cl} = -1$. This information is saved in a *PointInfo* texture, where each texel corresponds to a data entity in the original data set. As for meshes, reassigning a data point which belongs to the cluster m, i.e. with $ID_{cl} = m$, to a cluster n means that the data ID_{cl} index changes to $ID_{cl} = n$, see Figure 5.8 on page 103.

6.4.2 Initial Clustering Configuration

Both the Local Neighbors k-means and the Multilevel algorithms proposed in Section 6.2 - 6.3 start with an initial configuration as described in Section 6.2.

Given a step-size a the bounding box spanned by the data set is subdivided into $l = a^d$ voxels. Each voxel correspond to a starting cluster. Thus the cluster texture size is set to $clusterTexSize = ceil(sqrt(l))$. From the bounding box limits for each dimension k the $voxelSize_k$, as the voxel size for a specific dimension, is computed as $(limitMax_k - limitMin_k)/a$.

However, since the subdivisions are done in a d-dimensional space but we work on 2D textures, we need a mapping from the subdivided d-dimensional space to the 2D texture space. This is achieved using the following formula:

$$ID_{cl} = \sum_{k=0}^{d-1} (base)^k m_k. \tag{6.2}$$

where *base* is the basis from which the mapping is done and the m_k are the corresponding numerals for each dimension with values between 0 and $base - 1$. As an example, for *ClusterData* textures we have $base = clusterTexSize$ and the numerals m_k are exactly the texture coordinates.

If the cluster index ID_{cl} is known the numerals m_k are obtained as:

$$m_k = \frac{ID_{cl}}{base^k} \ \% \ base. \tag{6.3}$$

for k between 0 and $d - 1$, and % as modulo operation.

If we want to map from one base to another, the cluster ID_{cl} is computed first in one basis using Eq. (6.2) and then mapped to a different one using the numerals computed according to the Eq. (6.3).

Having a valid initialization means that each d-dimensional data point has assigned a correct cluster index ID_{cl} and that each cluster with index ID_{cl} has $2d$ valid local neighbors. This is done as follows:

1. Assigning the d-dimensional data points to correct starting cluster ID_{cl} is done using a fragment shader. For each fragment which corresponds to a texel in *PointInfo* texture an ID_{cl} is computed according to Eq. (6.2) with $base = a$ and base vectors $m_k = Point[k]/voxelSize_k$. Where $Point[k]$ is the k-th point coordinate component and $voxelSize_k$ is the voxel size for a given k-th dimension.

2. Initially, the neighbors are assigned according to the voxel-based spatial segmentation, see Section 6.2 and Figure 6.5 on page 125. For a given cluster with index ID_{cl} the numerals m_k in basis a can be computed according to Eq. (6.3), see the *getNumerals(int forID)* subroutine in Algorithm B.1 in the Appendix B. These numerals are used to identify the indices ID^*_{cl} of the neighboring clusters. Figure 6.7 depicts a 2D example. By increasing and correspondingly decreasing by one the value m_k for each dimension (see the *getNeighbors(int forDim)* subroutine in Algorithm B.1 in the Appendix B) different cluster's indices according to Eq. (6.2) can be computed. These are assigned as the indices of the neighboring clusters.

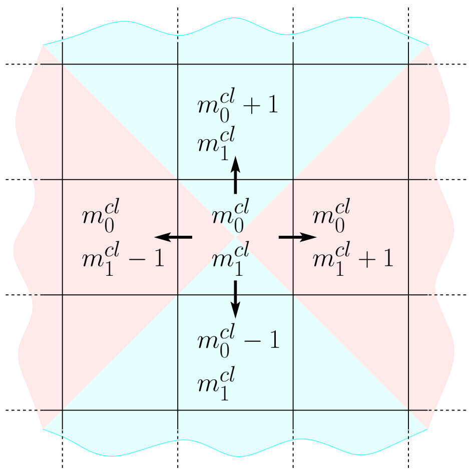


Figure 6.7: A 2D example for identifying the indices of the neighboring clusters. For a given cluster with index ID_{cl} and corresponding numerals (m_0^{cl}, m_1^{cl}), the indices for four neighbors (indicated by arrows) are computed according to Eq. (6.2) using four corresponding numerals pairs.

6.4.3 Data Clustering

Local Neighbors k-means:

The approach, see Section 6.2, is implemented similarly to the Boundary-based algorithm, as described in Section 5.2.3. The *PointInfo* texture is rasterized and for each fragment, which corresponds to a given point Q_j, the closest cluster is identified according to Algorithm 6.1. The points which remain assigned to the same clusters are simply discarded. In this case an occlusion query can be used to count the number of written fragments. If all fragments are discarded the algorithm stops, meaning that no points can be reassigned to other neighboring clusters such that the energy decreases.

Multilevel clustering:

The workflow and the way in which the ML data clustering works is the same as for mesh clustering, for additional details refer to Section 5.2.4. Here, the *NeighborsInfo* texture is used instead of the *FaceInfo* texture. In the *NeighborsInfo* texture each cluster C_i has a reference to its N_k^i neighbors. Thus, for each of these neighbors a DE can be computed, resulting in maximally $2d$ DEs.

6.4.4 Brute-force K-Means on the GPU

In order to compare the newly proposed Local Neighbors k-means, Section 6.2, with the classical brute-force k-means algorithm, see Section 2.2.1, we integrated the latter into our framework.

This can be achieved very easily. As described in Section 5.2.3, the *PointInfo* is used to update the clusters centroid, i.e by applying *GatherClusterData()* and *ComputeCluster-Proxy()* subroutines.

To perform the optimization step we load the current cluster's centroid and the cluster ID into a vertex stream. In the geometry shader we generate for each vertex, i.e. cluster, a quad that covers the complete *PointCoordinate* texture. For each generated fragment the "point to cluster energy" is computed. Using as fragment depth the computed energy and performing the depth test one obtains for each point in the *PointInfo* texture the ID of the closest cluster, i.e a new *PointInfo* texture. This process, i.e. computing clusters centroids and reassigning the points to the closest clusters, is repeated until no points are reassigned to other clusters.

6.5 GPU-based Data Clustering Results

It is known that there is a strong link between the Centroidal Voronoi Diagram [DFG99] and the k-means clustering, see Section 2.2.1. Both minimize the within-cluster variance, i.e. the squared distance between cluster's centroid and its assigned data points. Thus, the energy functional Eq. (5.4) equally applies to data clustering and can be written as:

$$E = \sum_{i=0}^{k-1} E_i = \sum_{i=0}^{k-1} \sum_{Q_j \in C_i} \|\mathbf{Q}_j - \overline{\mathbf{C}_i}\|^2. \tag{6.4}$$

where $\mathbf{Q}_j$ and $\overline{\mathbf{C}_i}$ is the data point and cluster's centroid coordinates, respectively. The cluster's centroid is computed as $\overline{\mathbf{C}_i} = \sum_{Q_j \in C_i} \mathbf{Q}_j / n_i$, where n_i is the total number of points in cluster C_i.

The Dual Edge collapse cost between two clusters C_1 and C_2 is computed as proposed in [CK08] using:

$$DE_{cost} = E_{1\cup 2} - E_1 - E_2. \tag{6.5}$$

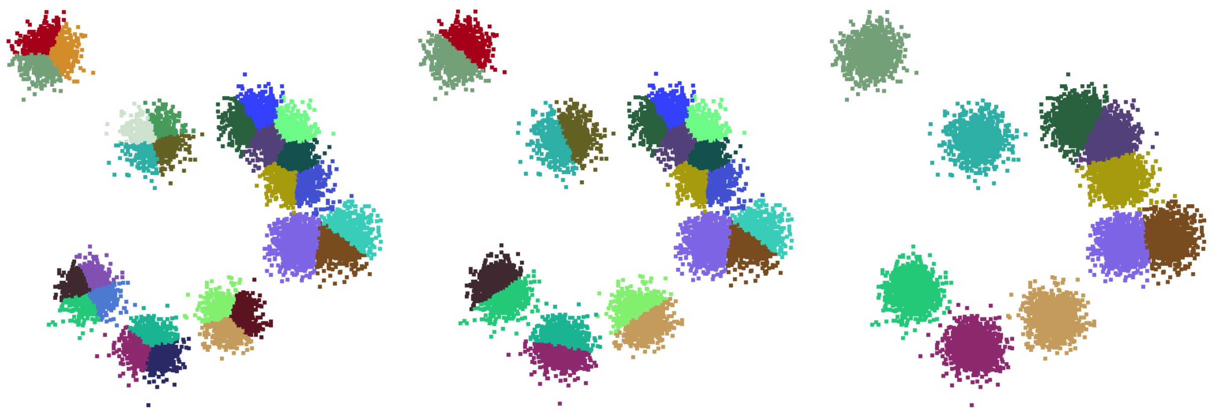


Figure 6.8: 2D example of ML data clustering. (start): The starting ML configuration as shown in Figure 6.5. (20 cl.) and (10 cl.): ML clustering results for different number of clusters.

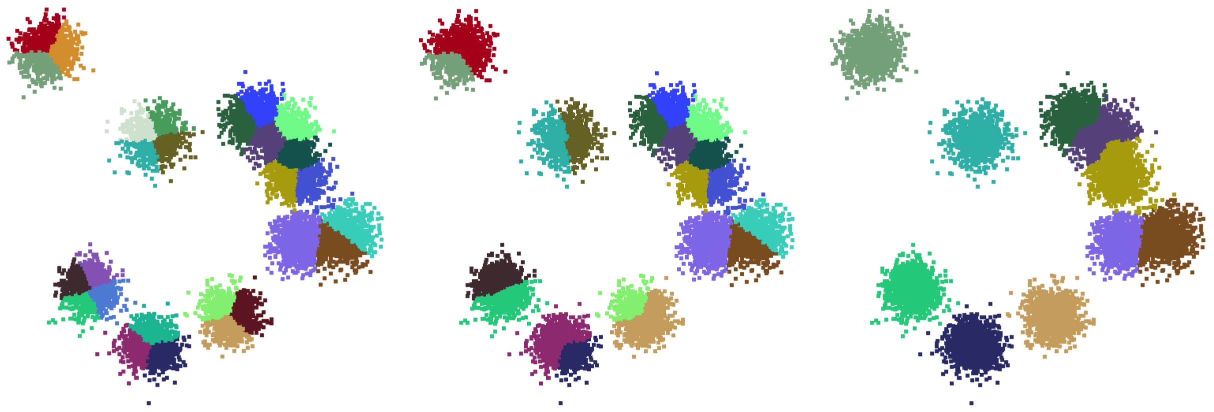


Figure 6.9: 2D example of hierarchical (HC) data clustering. (start): The starting HC configuration. (20 cl.) and (10 cl.): HC clustering results for different number of clusters.

However, we must point out that, there are other merging cost definitions which for different data clustering problems are considered to be more appropriate, e.g. some take the cluster size into account, see for an overview [XW08] and [GMW07].

For an easier computation of the merging energy, Eq. 6.4 can be written in the form:

$$E_i = \sum_j \|\mathbf{Q}_j\|^2 - 2\overline{\mathbf{C}_i} \cdot (\sum_j \mathbf{Q}_j) + n_i \|\overline{\mathbf{C}_i}\|^2. \tag{6.6}$$

Thus in the *PointData* texture, see Section 6.4.1, we only need to store the values $\|\mathbf{Q}_j\|^2$, $\mathbf{Q}_j$. Correspondingly, the *ClusterData* texture stores the following information: $\sum_j \|\mathbf{Q}_j\|^2$, $\sum_j \mathbf{Q}_j$ and n_i.

Figure 6.5 and Figure 6.8 shows the clustering result for $10k$ 2D data points which represent 10 randomly distributed Gaussians. Figure 6.5 depicts the result after applying the Local Neighbors k-means algorithm for a step-size equal to 6 resulting in 27 valid clusters. Note that, similar to mesh clustering, the perfect symmetry present in the final clustering. This indicates that the algorithm performs very well regardless of the initial configuration. The visual quality observed for ML clustering results at 20 and 10 number of clusters is also very pleasing, see Figure 6.8.

In contrast, the result of the classical hierarchical data clustering, as presented in Figure 6.9, is worse compared to the Multilevel approach. This is due to a strict containment of the lower hierarchy levels in the upper ones.

Figure 6.10 shows the energy behavior of the ML clustering compared with the k-means clustering, obtained for $100k$ 3D data points which represent 50 randomly distributed Gaussians. As expected, in the region of interest the average energy of the k-means is significantly higher compared to that of ML clustering. It is close to the ML energy only when the number of seeds is much larger than the real number of clusters present in the data set, i.e. at 102 clusters.

Due to a random initialization, large fluctuations are obtained in the final result of the k-means algorithm, whereas the Local Neighbors k-means always leads to the same final configuration due to its fixed initial spatial subdivision. Note, that there are cases where the k-means energy, not the average energy, is lower than that of the ML. This *only* happens after the first optimization in the ML construction, see Figure 6.10 at 102 and 57 clusters. This is due to the fact that the k-means (and that only by chance) may have a better initial configuration compared to the uniform sampling initialization used for the ML data clustering.

In order to correctly judge the difference in behavior between the newly proposed Local Neighbors k-means and the brute-force k-means algorithms (which usually has a random initialization), we generate the initial set of seeds for the k-means from the initial configurations of the Local Neighbors k-means. Then the point closest to the obtained cluster centroid is assigned as seed. This ensures that both algorithms start with the same initial configuration.

Figure 6.11 shows the resulting timing for both algorithms for different numbers of clusters k. For a fixed point data set with m points, the Local Neighbors k-means is not affected by increasing the number of clusters, where the computation time for the k-means increases linearly per iteration. In each iteration, the Local Neighbors k-means performs $m(4d-1)$ point-to-cluster computations, thus increasing the number of clusters does not affect the number of these computations. At the same time the k-means requires mk point-to-cluster computations. This is a very important property of the Local Neighbors k-means showing that the newly proposed algorithm scales much better than k-means with increasing number of clusters.

To see how the algorithm scales with increasing number of points, we have generated different data sets with different numbers of points for fixed positions of the Gaussians. Figure 6.12 shows the timing for these different configurations. It can be observed that the Local Neighbors k-means scales much better than the k-means, it is approximately two times faster.

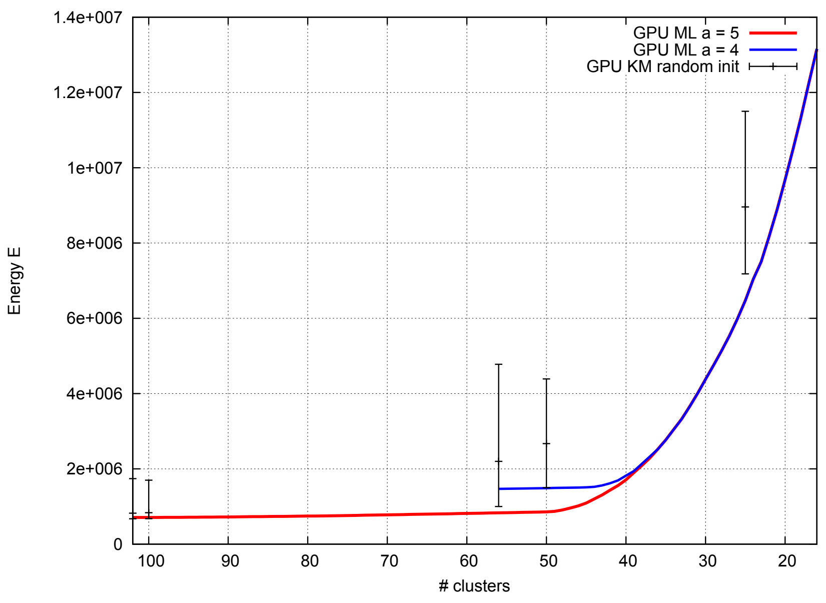


Figure 6.10: k-means energy as a function of the number of clusters. (GPU ML) Energy of the Multilevel data clustering. (GPU KM) Energy of the brute-force k-means. a - the number of voxels per dimension. The error bars were obtained by repeating the k-means 100 times with different random initializations.

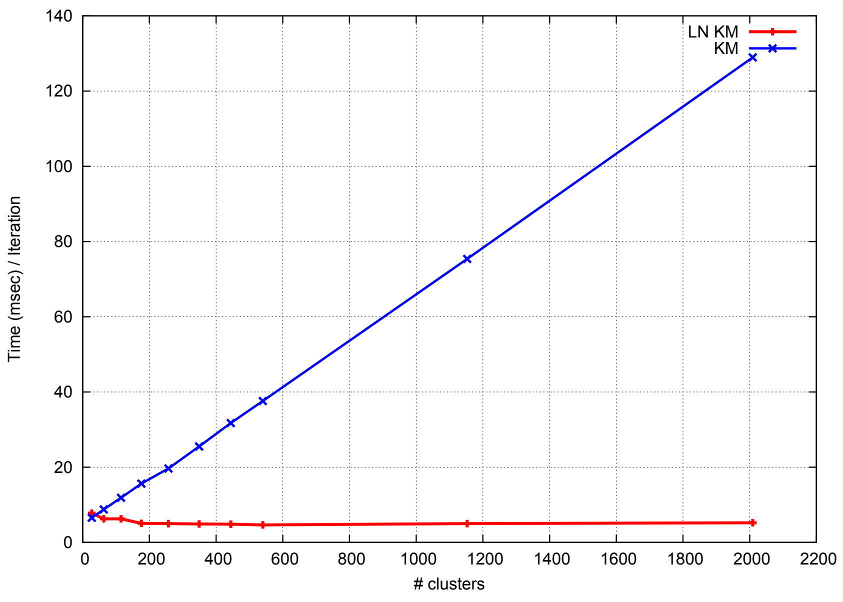


Figure 6.11: Local Neighbors k-means (LN-KM) vs. k-means (KM) timing for different number of clusters at fixed $100k$ number of points.

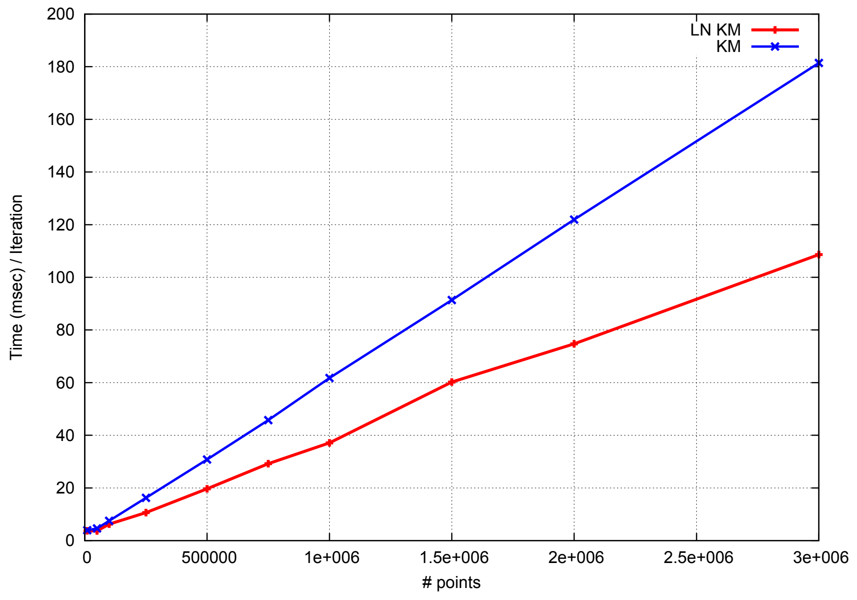


Figure 6.12: Local Neighbors k-means (LN-KM) vs. k-means (KM) timing for different number of points at fixed ≈ 100 number of clusters.

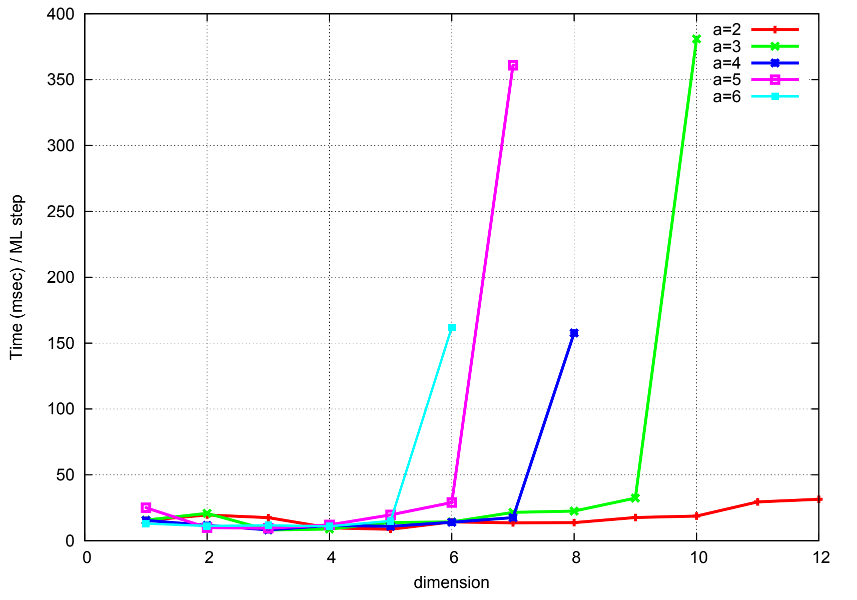


Figure 6.13: Average time of one ML step (merging and optimization) vs. dimensions.

Data clustering usually requires high-dimensional processing. Therefore, it is important to investigate how the algorithm processing time scales with the increasing number of dimensions. Due to hardware restriction, our current implementation can support up to 16 dimensions, see the discussions in Section 6.4.1. To make a correct judgment we first generate a data set with 50 Gaussians each containing $1k$ points for the highest supported dimension $d = 16$. To obtain data sets for lower dimensions $d < 16$ we simply project the first d dimensions on the original data set. Figure 6.13 presents the final result. It is interesting to note that the average time to perform one ML step, which includes merging two clusters and applying the Local Neighbors k-means, is nearly constant. This indicates that the ML scales very well with increasing number of dimensions as well. The large timing values which can be seen for different values of a are due to GPU memory management problems. Remember that in the initialization, see Section 6.4.2, we start with $l = a^d$ voxels, therefore with increasing number of dimensions the GPU memory limit is reached very quickly as more memory resources are required.

6.6 Identifying the Number of Clusters

One of the major goals of data clustering is to identify the "true" number of clusters that best describes a given data set. The ML approach does not provide a direct answer to this question – however it provides the complete set of all solutions. Depending on a specific problem, the algorithm can be designed to choose one best solution out of all possible solutions.

The simplest approach in this case is to use the energy measure, e.g. as done in Section 4.4 to identify different types of shapes present in the model. It is observed that as long as the data is not naturally clustered smaller increases in the energy can be observed as the number of clusters decreases, see Figure 6.10. However, if the data naturally clusters then the energy very quickly increases – the so-called *elbowing effect* [DGJW06].

Another approach is to use the Bayesian Information Criterion (BIC) as done in [PM00] for X-means algorithm[2]. The algorithm can then report the solution with highest BIC or stop when the BIC score starts decreasing[3].

Figure 6.14 shows the BIC values for different ML levels for the data set shown in Figure 6.8. Here we compute the BIC score exactly as described in [PM00]. As expected, the highest BIC score is obtained for 10 clusters. This way the ML algorithm can be designed to report the solution with highest BIC or even stop and report the best solution if the BIC score starts decreasing[4].

In contrast to meshes for which there are no top-down approaches, for data clustering such algorithms exist, see Section 2.2.2. Thus, it is important to compare the ML algorithm to these.

[2]For more details on the BIC computation see Section 2.2.1.

[3]Note, that the same principle can be used with other measures. See [XW08] for other model selection criteria.

[4]The results in Figure 6.15 were obtained using this idea.

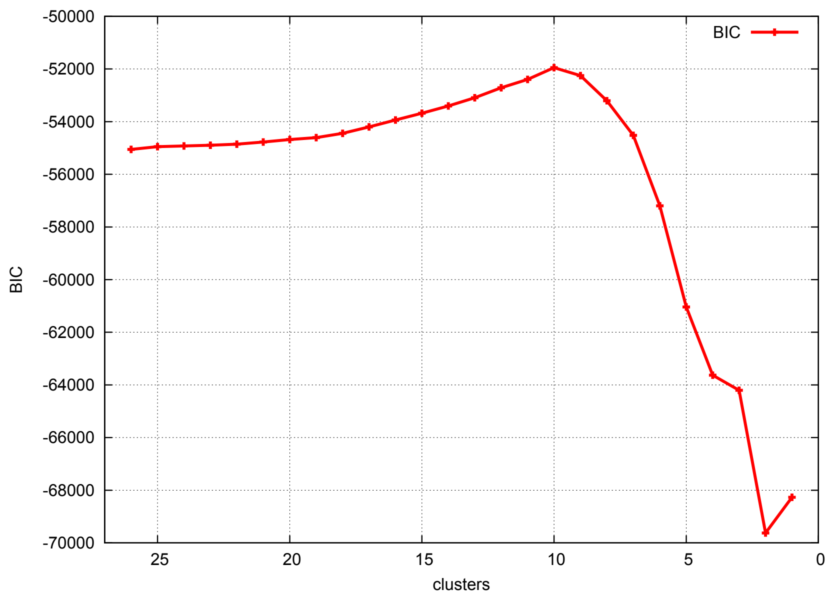


Figure 6.14: The BIC score at different ML levels for data set presented in Figure 6.8.

The bisecting k-means [SKK00], [SB04] could be considered in this case. However, as already pointed out in Section 2.2.2, it provides a hierarchy of clusters (a binary tree). As a result, it suffers from the strict containment of upper levels in the lower ones, although less than agglomerative hierarchical clustering.

In contrast, the X-means algorithm [PM00] applies an optimization step after any bisecting step, therefore the X-means leads to superior results compared to the bisecting k-means. Thus, we choose to compare the ML approach with the X-means algorithm [PM00]. Even more, the X-means algorithm is specially designed to stop and report the "true" number of clusters.

To compare how well both algorithms can identify the "true" number of clusters present in the data, we generated different data sets with different numbers of spherical Gaussians with a standard deviation of one, each with 500 points, in a fixed 3D volume with limits between $[-25; 25]$. The ML algorithm stops when the BIC value starts to decrease. The result of this experiment[5] is presented in Figure 6.15. As expected, as long as the clusters are well separated both algorithms are able to identify the original number of clusters present in the data. However, starting from 100 clusters as more Gaussians begin overlapping, the output of both algorithms start to diverge. The X-means always identifies with large fluctuations less clusters as originally generated in the data, which is more or less in agreement with the results presented in [PM00]. In contrast, the ML approach is always very close to the original number of clusters present in the data. Thus, ML proves to be better at revealing the "true" number of clusters compared to the X-means, which also reflects on the quality of the clustering result.

[5]To obtain the results for X-means we used a publicly available implementation of the algorithm [PM].

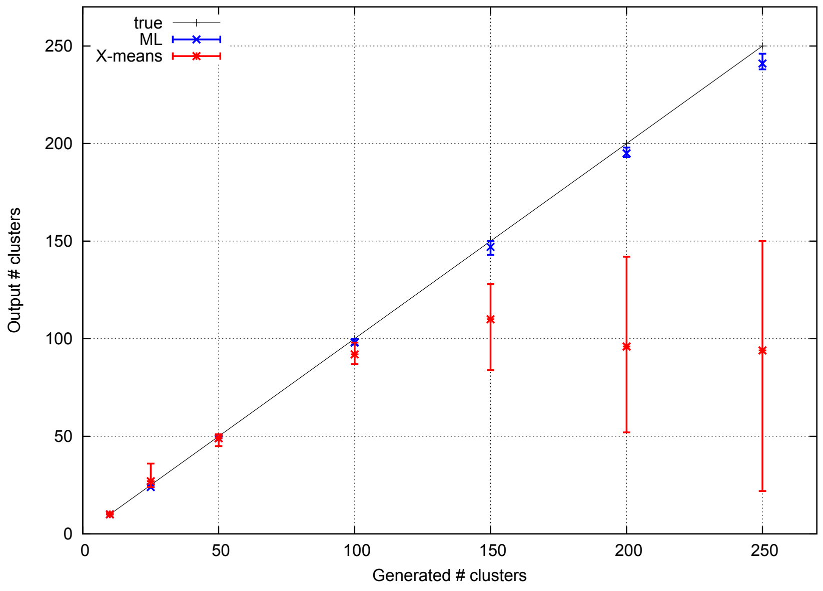


Figure 6.15: The output number of clusters for different number of generated Gaussians. Each Gaussian contains 500 3D points. The error bars were obtained using 10 different data sets.

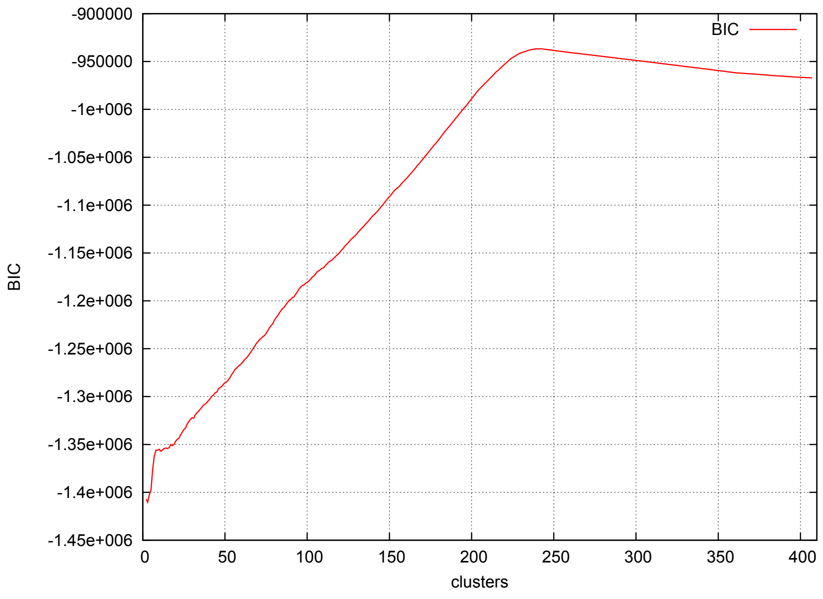


Figure 6.16: The BIC score at different ML levels for 250 generated Gaussians.

The explanation of this result can be seen in Figure 6.16 where the BIC score for different ML levels for a data set consisting of 250 Gaussians is shown. Note that, in the range from 1 to 100 clusters, there are several local maxima present in the BIC values. Thus, the X-means most probably will be trapped in such regions and provides unsatisfactory results. In this example the X-means reports 22 clusters. In contrast, doing an agglomerative ML there are no such fluctuations in the BIC values and the maximum appears at 242 clusters. This again emphasize the methodological advantage of the Multilevel approach, in this case over the top-down approaches, i.e. the X-means algorithm.

6.7 Conclusions

In this chapter we have generalized the Multilevel approach to data clustering. As a result, the major advantages of the ML technique, discussed in the previous two chapters, can now readily apply to data clustering. As in the case of mesh clustering, the ML approach has shown to be a good strategy in overcoming the stated problems of the k-means and hierarchical clustering methods.

In this context, we have presented a method to resolve the missing topological information by dynamically tracking cluster neighborhoods. This tracking allows the identification of cluster neighbors required for cluster merging and for cluster optimization, without enforcing a disadvantageous storage of per-element neighborhoods.

The clustering technique is formulated as parallel algorithm, which incorporates local neighbor checks. Both the ML and the local neighborhood approach form the basis for a framework solely based on GPU resources, thus making it more attractive. Using this framework we have shown that considerable speedup can be obtained.

As a result the Multilevel approach becomes a powerful tool for mesh as well as data clustering and analysis. It is very flexible in terms of enabling and disabling the optimization and/or parallel cluster merging. Compared to classical approaches, our technique generates results with at least the same clustering quality, and is better at revealing the number of clusters present in the data set.

Furthermore, our approach scales very well, currently being limited only by the available amount of graphics memory.

Summary and Future Work

In this dissertation we have presented a new generic Multilevel clustering technique. The framework was applied to mesh clustering and to general data clustering. From a conceptual point of view, the topological information for meshes has been replaced by dynamic cluster tracking in the case of general data clustering. The major algorithmic elements are *boundary-based queries*, which strongly incorporate the spatial coherence present in the optimization and the cluster merging steps without having any restriction regarding the energy functional.

Our clustering technique has several methodological advantages:

1. The approach has been shown to be a good strategy to overcome the stated problems of the iterative and hierarchical clustering methods, thus providing robust and high quality clustering results.

2. Our formulation is free from any global data structures. This allows an efficient GPU-based implementation and a further step in parallelization.

The technique is formulated as a parallel algorithm, which incorporates local neighbor checks. This conceptual combination of optimization with hierarchical techniques and its formulation as a parallel algorithm yields a multilevel clustering approach, that optionally performs the cluster merging also in parallel. Both the parallel and the local neighborhood approach form the basis of a framework solely based on GPU resources, resulting in an algorithm which scales very well.

Using this framework we have shown that considerable speedups can be obtained, yielding at least the same clustering quality compared to standard techniques. The proposed parallel multilevel clustering approach is very flexible. It has no limitation on the way in which parallel Multilevel clustering can be performed.

In the field of mesh processing the GPU-based Multilevel approach is an important building block, since it is the first GPU-based approach that performs a “true” mesh clustering. In this context, because of a steadily growing GPU power, we expect even more techniques to emerge based on the concepts presented in this dissertation. New application areas need to be identified by testing more standard energy functionals or developing new ones.

In the field of data clustering the Multilevel approach proved to be a better strategy compared to the top-down approaches, such as the X-means algorithm, at revealing the number of clusters present in the data. Its GPU-based implementation makes it even more attractive, as considerable speedup can be obtained.

Nonetheless, we have to recognize that in both cases there is still room for optimization. This regards the algorithm as well as the GPU-specific implementation parts. Thus, we need to identify new GPU-based techniques to even further speed up the clustering process. Here the newest hardware architecture specification must be taken into account.

Finally, we expect to see the Multilevel approach applied in many other clustering-related applications, such as gene analysis, document clustering, image segmentation, etc. Thus, the Multilevel concept together with a GPU-based implementation becomes more generic and can be established as a standard approach for any clustering tasks.

Appendix A

Proof of Proposition 3.1

PROPOSITION 3.1 . *Given a set of n different seeds $\{\mathbf{z}_i\}_{i=0}^{n-1}$ with associated positive weights $\{w_i\}_{i=0}^{n-1}$ and a density function $\rho(\mathbf{x})$ in the domain Ω. Let $\{D_i^{mw}\}_{i=0}^{n-1}$ denote any tessellation of Ω into n regions. Define:*

$$E^{mw} = \sum_{i=0}^{n-1} \int_{D_i^{mw}} \rho(\mathbf{x})\ w_i \left|\mathbf{x} - \mathbf{z}_i\right|^2 d\mathbf{x}$$

E^{mw} is minimized if and only if $\{D_i^{mw}\}_{i=0}^{n-1}$ is a Multiplicatively Weighted Centroidal Voronoi Diagram (MWCVD).

Proof: Given a set of n different seeds $\{\mathbf{z}_i\}_{i=0}^{n-1}$ with associated positive weights $\{w_i\}_{i=0}^{n-1}$, a positive density function $\rho(\mathbf{x})$ in the 2D domain Ω and a tessellation of Ω into n regions $\{D_i^{mw}\}_{i=0}^{n-1}$. The E^{mw}, according to Eq. (3.22), is minimized if and only if the regions D_i^{mw} are the MW-Voronoi regions associated to the generators $\mathbf{z}_i$ and, simultaneously, the generators $\mathbf{z}_i$ are the centroids $\mathbf{z}_i^*$ of the regions D_i^{mw}, see Eq. (3.21).

We follow a similar argument as in [DFG99]:

First, assuming that E^{mw} is minimized, we have to show that the generators $\{\mathbf{z}_i\}_{i=0}^{n-1}$ are the centroids corresponding to the regions $\{D_i^{mw}\}_{i=0}^{n-1}$.

Examine the variation of E^{mw} with respect to a fixed $\mathbf{z}_i$, namely $E^{mw}(\mathbf{z}_i+\epsilon\mathbf{v})-E^{mw}(\mathbf{z}_i)$, where $\mathbf{z}_i + \epsilon\mathbf{v} \in \Omega$. Now, dividing by ϵ and taking the limit as $\epsilon \to 0$:

$$\lim_{\epsilon\to 0} \frac{E^{mw}(z_i + \epsilon\mathbf{v}) - E^{mw}(z_i)}{\epsilon} = \frac{\int \rho(\mathbf{x})w_i \left|\mathbf{x} - \mathbf{z}_i - \epsilon\mathbf{v}\right|^2 d\mathbf{x} - \int \rho(\mathbf{x})w_i \left|\mathbf{x} - \mathbf{z}_i\right|^2 d\mathbf{x}}{\epsilon} = 0$$

yields:

$$\mathbf{z}_i = \frac{\int_{D_i^{mw}} \mathbf{x}\rho(\mathbf{x})d\mathbf{x}}{\int_{D_i^{mw}} \rho(\mathbf{x})d\mathbf{x}}$$

Second, we show that E^{mw} is minimized if $\{D_i^{mw}\}_{i=0}^{n-1}$ are MW-Voronoi regions associated with sites $\{\mathbf{z}_i\}_{i=0}^{n-1}$.

Recall the fact that due to the construction of the MW-Voronoi diagram for $\mathbf{x} \in D_i^{mw}$ we get for $i \neq j$:

$$\begin{aligned} & w_i \left|\mathbf{x} - \mathbf{z}_i\right| < w_j \left|\mathbf{x} - \mathbf{z}_j\right| \\ \Longleftrightarrow \quad & \rho(\mathbf{x}) w_i \left|\mathbf{x} - \mathbf{z}_i\right|^2 < \rho(\mathbf{x}) w_j \left|\mathbf{x} - \mathbf{z}_j\right|^2 \end{aligned}$$

Thus, for a *fixed* set of sites $\{\mathbf{z}_i\}_{i=0}^{n-1}$ the energy E^{mv} defined according to Eq. (3.22) will be smaller compared to the case when the tessellation is not a MW-Voronoi diagram.

Appendix B

Compute Cluster's Local Neighbors.

Algorithm B.1. *(GLSL code to compute cluster's local neighbors.)*

```
int numerals[dimension]; //global for numerals

//compute numerals with the base=subdivisions
void getNumerals(int forID) {
  int temp = 1;
  for(int d=0; d<dimension; d++) {
    numerals[d] = (forID/temp)%subdivisions;
    temp *= subdivisions;
  }
}

//generic is good, but this is faster
int getClusterID2D(ivec2 texIndices)
{ return clustersTextureSize*texIndices[1] + texIndices[0]; }

//get the cluster index with an offset at given dimension
int compClusterID(int atDim, int offset) {
  if( (numerals[atDim] + offset) < 0
      || (numerals[atDim] + offset) == subdivisions ) return -150;
  else {
    int temp=0; int multBase=1;
    for(int d=0; d<dimension; d++) {
      if( d != atDim ) temp += multBase*numerals[d];
      else temp += multBase*(numerals[d] + offset);

      multBase*=subdivisions;
    }
    return temp;
  }
}

//compute bottom and upper neighbors for a given dimension
ivec2 getNeighbors(int forDim) {
  ivec2 outVec=ivec2(0, 0);

  if( forDim<dimension ) {
    outVec[0] = compClusterID(forDim, 1);
    outVec[1] = compClusterID(forDim, −1);
  }
  return outVec;
}

//The main entry point of this GPU program
void main() {
  //get cluster id
  int clusterID = getClusterID2D( ivec2(floor(gl_TexCoord[0].xy)) );

  //get the numerals in a new base
  getNumerals(clusterID);

  //get new neighbors
  for(int rt=0; rt<clustersNeighborsRenderTargets; rt++)
  gl_FragData[rt] = vec4(getNeighbors(rt*2+0), getNeighbors(rt*2+1));
}
```

Bibliography

[AdVDI03] Alliez P., de Verdière E. C. d., Devillers O., Isenburg M.: Isotropic Surface Remeshing. In *Proc. of the Shape Modeling International (SMI)* (2003), IEEE Computer Society.

[AdVDI05] Alliez P., de Verdière E. C., Devillers O., Isenburg M.: Centroidal Voronoi diagrams for isotropic surface remeshing. *Graph. Models 67*, 3 (2005), 204–231.

[AE84] Aurenhammer F., Edelsbrunner H.: An optimal algorithm for constructing the weighted Voronoi diagram in the plane. *Pattern Recognition 17*, 2 (1984), 251–257.

[AFS06] Attene M., Falcidieno B., Spagnuolo M.: Hierarchical mesh segmentation based on fitting primitives. *The Visual Computer 22*, 3 (2006), 181–193.

[AKM*06] Attene M., Katz S., Mortara M., Patane G., Spagnuolo M., Tal A.: Mesh Segmentation - A Comparative Study. In *Proc. of the IEEE Int. Conf. on Shape Modeling and Applications (SMI)* (2006), IEEE Computer Society, p. 7.

[AMD02] Alliez P., Meyer M., Desbrun M.: Interactive Geometry Remeshing. *ACM Transactions on Graphics 21* (2002), 347–354.

[APP*07] Agathos A., Pratikakis I., Perantonis S., Sapidis N., Azariadis P.: 3D Mesh Segmentation Methodologies for CAD applications. *Computer-Aided Design and Applications 4*, 6 (2007), 827 – 841.

[Bau72] Baumgart B. G.: *Winged edge polyhedron representation.* Tech. rep., Stanford, CA, USA, 1972.

[BPK*08] Botsch M., Pauly M., Kobbelt L., Alliez P., Lvy B.: Geometric Modeling Based on Polygonal Meshes. In *Eurographics Tutorial* (2008).

[Buc05] Buck I.: *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation.* Addison Wesley, 2005, ch. Taking the Plunge into GPU Computing, pp. 509–519.

[CGF09] Chen X., Golovinskiy A., Funkhouser T.: A benchmark for 3D mesh segmentation. *ACM Transactions on Graphics (Proc. SIGGRAPH) 28*, 3 (2009).

[CK06] Chiosa I., Kolb A.: Mesh Coarsening based upon Multiplicatively Weighted Centroidal Voronoi Diagrams. In *Technical Report, Computer Graphics Group, Institute for Vision and Graphics (IVG), University of Siegen* (2006).

[CK08] Chiosa I., Kolb A.: Variational Multilevel Mesh Clustering. In *Proc. IEEE Int. Conf. on Shape Modeling and Applications (SMI)* (2008), pp. 197–204.

[CK11] Chiosa I., Kolb A.: GPU-based Multilevel Clustering. *IEEE Transactions on Visualization and Computer Graphics (TVCG) 17*, 2 (2011), 132–145.

[CKCL09] Chiosa I., Kolb A., Cuntz N., Lindner M.: Parallel Mesh Clustering. In *Proc. Eurographics Symp. on Parallel Graphics and Visualization (EGPGV)* (2009), pp. 33–40.

[CKS98] Campagna S., Kobbelt L., Seidel H.-P.: Directed Edges - A Scalable Representation for Triangle Meshes. *Journal of Graphics Tools 3* (1998).

[Coh99] Cohen J. D.: Concepts and algorithms for polygonal simplification. *SIGGRAPH Course Tutorial: Multiresolution Surface Modeling Course* (1999).

[CP05] Cazals F., Pouget M.: Estimating Differential Quantities using Polynomial fitting of Osculating Jets. *Computer Aided Geometric Design 22* (2005), 121–146.

[CSAD04] Cohen-Steiner D., Alliez P., Desbrun M.: Variational shape approximation. In *Proc. SIGGRAPH* (2004), pp. 905–914.

[CSM03] Cohen-Steiner D., Morvan J.-M.: Restricted delaunay triangulations and normal cycle. In *Proc. of symposium on Computational geometry (SCG)* (2003), ACM, pp. 312–321.

[CTZ06] Cao F., Tung A. K., Zhou A.: Scalable clustering using graphics processors. In *Lecture Notes in Computer Science* (2006), vol. 4016, Springer, pp. 372–384.

[Cun09a] Cuntz N.: gp3tools. http://www.cg.informatik.uni-siegen.de/Programming/hase3d, 2009.

[Cun09b] Cuntz N.: *Real-Time Particle Systems.* PhD Thesis, Computer Graphics Group, Institute for Vision and Graphics (IVG), University of Siegen, 2009.

[DBvKOS00] De Berg M., van Kreveld M., Overmars M., Schwarzkopf O.: *Computational geometry: algorithms and applications*. Springer, 2000.

[DFG99] Du Q., Faber V., Gunzburger M.: Centroidal Voronoi Tessellations: Applications and Algorithms. *SIAM Review 41*, 4 (1999), 637–676.

[DGJ03] Du Q., Gunzburger M. D., Ju L.: Constrained centroidal voronoi tessellations for surfaces. *SIAM Journal on Scientific Computing 24*, 5 (2003), 1488–1506.

[DGJW06] Du Q., Gunzburger M., Ju L., Wang X.: Centroidal voronoi tessellation algorithms for image compression, segmentation, and multichannel restoration. *J. Math. Imaging Vis. 24*, 2 (2006), 177–194.

[DW05] Du Q., Wang D.: Anisotropic centroidal voronoi tessellations and their applications. *SIAM J. Sci. Comput. 26*, 3 (2005), 737–761.

[EDD*95] Eck M., DeRose T., Duchamp T., Hoppe H., Lounsbery M., Stuetzle W.: Multiresolution analysis of arbitrary meshes. In *Proc. SIGGRAPH* (1995), pp. 173–182.

[For65] Forgy E.: Cluster analysis of multivariate data: efficiency versus interpretability of classifications. *Biometrics 21* (1965), 768–780.

[FRCC08] Farivar R., Rebolledo D., Chan E., Campbell R. H.: A parallel implementation of k-means clustering on gpus. In *Proc. of Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA)* (2008), pp. 340–345.

[Gar99] Garland M.: Multiresolution modeling: Survey & future opportunities. *EUROGRAPHICS - State of the Art Report* (1999), 111–131.

[GDB08] Garcia V., Debreuve E., Barlaud M.: Fast k nearest neighbor search using GPU. In *CVPR Workshop on Computer Vision on GPU* (2008).

[GG04] Gelfand N., Guibas L. J.: Shape segmentation using local slippage analysis. In *Proc. of Eurographics/ACM SIGGRAPH symp. on Geometry processing* (2004), ACM, pp. 214–223.

[GG06] Gatzke T., Grimm C.: Estimating curvature on triangular meshes. *International Journal of Shape Modeling 12*, 1 (June 2006), 1–29.

[GGH02] Gu X., Gortler S., Hoppe H.: Geometry images. In *Proc. SIGGRAPH* (2002), pp. 355–261.

[GH97] Garland M., Heckbert P. S.: Surface simplification using quadric error metrics. In *Proc. SIGGRAPH* (1997), pp. 209–216.

[GHJ*97] Gieng T. S., Hamann B., Joy K. I., Schussman G. L., Trotts I. J.: Smooth hierarchical surface triangulations. In *Proc. of the conf. on Visualization (VIS)* (1997), IEEE Computer Society Press, pp. 379–386.

[GMW07] Gan G., Ma C., Wu J.: *Data Clustering: Theory, Algorithms, and Applications.* ASA-SIAM Series on Statistics and Applied Probability, 2007.

[GWH01] Garland M., Willmott A., Heckbert P. S.: Hierarchical face clustering on polygonal surfaces. In *Proc. Symp. on Interactive 3D graphics (I3D)* (2001), pp. 49–58.

[Ham94] Hamann B.: A data reduction scheme for triangulated surfaces. *Comput. Aided Geom. Design 11*, 2 (1994), 197–214.

[HE02] Hamerly G., Elkan C.: Alternatives to the k-means algorithm that find better clusterings. In *Proc. of the Int. Conf. on Information and knowledge management (CIKM)* (2002), pp. 600–607.

[HG97] Heckbert P. S., Garland M.: Survey of polygonal surface simplification algorithms. *Proc. SIGGRAPH: Multiresolution Surface Modeling Course* (1997).

[HH04] Hall J. D., Hart J. C.: GPU acceleration of iterative clustering. In *Manuscript accompanying poster at GP^2. The ACM Workshop on General Purpose Comp. on Graph. Processors, and SIGGRAPH 2004 poster* (2004).

[Hop96] Hoppe H.: Progressive meshes. In *Proc. SIGGRAPH* (1996), pp. 99–108.

[Hop97] Hoppe H.: View-dependent refinement of progressive meshes. In *SIGGRAPH: Proc. of conf. on Computer graphics and interactive techniques* (1997), ACM Press/Addison-Wesley Publishing Co., pp. 189–198.

[HtLlDt*09] Hong-tao B., Li-li H., Dan-tong O., Zhan-shan L., He L.: K-means on commodity gpus with cuda. *World Congress on Computer Science and Information Engineering 3* (2009), 651–655.

[JKS05] Julius D., Kraevoy V., Sheffer A.: D-charts: Quasi-developable mesh segmentation. *Proc. EUROGRAPHICS 24*, 3 (2005), 581–590.

[KJ01] Kolb A., John L.: Volumetric Model Repair for Virtual Reality Applications. In *EUROGRAPHICS Short Presentation* (2001), University of Manchester, pp. 249–256.

[KJKZ94] Katsavounidis I., Jay Kuo C.-C., Zhang Z.: A new initialization technique for generalized lloyd iteration. *IEEE Signal Processing Letters 1*, 10 (1994), 144–146.

[Kle98] Klein R.: Multiresolution representations for surfaces meshes based on the vertex decimation method. *Computers and Graphics 22*, 1 (1998), 13 – 26.

[KLRS04] Kolb A., Latta L., Rezk-Salama C.: Hardware-based Simulation and Collision Detection for Large Particle Systems. In *Proc. Graphics Hardware* (2004), pp. 123–131.

[KVLS99] Kobbelt L. P., Vorsatz J., Labsik U., Seidel H.-P.: A Shrink Wrapping Approach to Remeshing Polygonal Surfaces. *Computer Graphics Forum 18*, 3 (1999), 119 – 130.

[KW03] Krüger J., Westermann R.: Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics (TOG) 22*, 3 (2003), 908–916.

[LE97] Luebke D., Erikson C.: View-dependent simplification of arbitrary polygonal environments. In *SIGGRAPH: Proc. of conf. on Computer graphics and interactive techniques* (1997), ACM Press/Addison-Wesley Publishing Co., pp. 199–208.

[Lin00] Lindstrom P.: Out-of-core simplification of large polygonal models. In *Proc. SIGGRAPH* (2000), pp. 259–262.

[Llo82] Lloyd S. P.: Least squares quantization in PCM. *IEEE Transactions on Information Theory 28*, 2 (1982), 129–137.

[LRC*02] Luebke D., Reddy M., Cohen J., Varshney A., Watson B., Huebner R.: *Level of Detail for 3D Graphics.* Elsevier Science Inc., 2002.

[LS03] Labelle F., Shewchuk J. R.: Anisotropic voronoi diagrams and guaranteed-quality anisotropic mesh generation. In *Proc. symposium on Computational geometry (SCG)* (2003), ACM, pp. 191–200.

[LSS*98] Lee A. W. F., Sweldens W., Schröder P., Cowsar L., Dobkin D.: MAPS: Multiresolution adaptive parameterization of surfaces. In *Proc. of SIGGRAPH* (998), pp. 95–104.

[Mac67] MacQueen J. B.: Some methods for classification and analysis of multivariate observations. In *Proc. of the Berkeley Symp. on Mathematical Statistics and Probability* (1967), vol. 1, University of California Press, pp. 281–297.

[Män88] Mäntylä M.: *An Introduction to Solid Modeling.* Computer Science Press, 1988.

[MGAK03] Mark W. R., Glanville R. S., Akeley K., Kilgard M. J.: Cg: a system for programming graphics hardware in a c-like language. In *ACM SIGGRAPH* (2003), pp. 896–907.

[MKMS07] Mantiuk R., Krawczyk G., Mantiuk R., Seidel H.-P.: High Dynamic Range Imaging Pipeline: Perception-Motivated Representation of Visual Content. In *Human Vision and Electronic Imaging XII* (2007), vol. 6492, SPIE, p. 649212.

[MS09] Moriguchi M., Sugihara K.: *Generalized Voronoi Diagram: A Geometry-Based Approach to Computational Intelligence*. Springer, 2009, ch. Constructing Centroidal Voronoi Tessellations on Surface Meshes, pp. 235–245.

[NT03] Nooruddin F. S., Turk G.: Simplification and Repair of Polygonal Models Using Volumetric Techniques. *IEEE Transactions on Visualization and Computer Graphics 9*, 2 (2003), 191–205.

[OBS92] Okabe A., Boots B., Sugihara K.: *Spatial Tesselations: Concepts and Applications of Voronoi Diagrams*. Wiley Publishing, 1992.

[OHL*08] Owens J. D., Houston M., Luebke D., Green S., Stone J. E., Phillips J. C.: GPU Computing. *Proceedings of the IEEE 96*, 5 (May 2008), 879–899.

[OLG*07] Owens J. D., Luebke D., Govindaraju N., Harris M., Krüger J., Lefohn A. E., Purcell T. J.: A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum 26*, 1 (2007), 80–113.

[Ols95] Olson C. F.: Parallel algorithms for hierarchical clustering. *Parallel Computing 21*, 8 (1995), 1313–1325.

[OS08] Ochotta T., Saupe D.: Image-based surface compression. *Computer graphics forum 27*, 6 (2008), 1647–1663.

[PC04] Peyré G., Cohen L.: Surface segmentation using geodesic centroidal tesselation. In *Int. Symp. on 3D Data Proc., Vis. and Transmission (3DPVT)* (2004), IEEE Computer Society, pp. 995–1002.

[PC06] Peyré G., Cohen L. D.: Geodesic remeshing using front propagation. *Int. J. Comput. Vision 69*, 1 (2006), 145–156.

[PM] Pelleg D., Moore A.: Auton Lab: K-means and x-means implementation. http://www.cs.cmu.edu/∼dpelleg/kmeans.html.

[PM00] Pelleg D., Moore A.: X-means: Extending K-means with Efficient Estimation of the Number of Clusters. In *Proc. of the Int. Conf. on Machine Learning* (2000), Morgan Kaufmann, pp. 727–734.

[RB93] Rossignac J., Borrel. P.: Multi-resolution 3D approximations for rendering complex scenes. In *Modeling in Computer Graphics: Methods and Applications* (1993), pp. 279–286.

[Ros06] Rost R. J.: *OpenGL(R) Shading Language (2nd Edition)*. Addison-Wesley Professional, 2006.

[Rus04] Rusinkiewicz S.: Estimating curvatures and their derivatives on triangle meshes. In *Proc. of the 3D Data Processing, Visualization, and Transmission (3DPVT)* (2004), IEEE Computer Society, pp. 486–493.

[SAG03] Surazhsky V., Alliez P., Gotsman C.: Isotropic remeshing of surfaces: A local parameterization approach. In *In Proc. of Int. Meshing Roundtable* (2003), pp. 215–224.

[SB04] Savaresi S. M., Boley D. L.: A comparative analysis on the bisecting K-means and the PDDP clustering algorithms. *Intell. Data Anal. 8*, 4 (2004), 345–362.

[Sch78] Schwarz G.: Estimating the Dimension of a Model. *The Annals of Statistics 6*, 2 (1978), 461–464.

[SDK05] Strzodka R., Doggett M., Kolb A.: Scientific Computation for Simulations on Programmable Graphics Hardware. *Simulation Practice & Theory 13*, 8 (2005), 667–680.

[SDT08] Shalom S. A., Dash M., Tue M.: Efficient K-Means Clustering Using Accelerated Graphics Processors. In *Proc. of the Int. conf. on Data Warehousing and Knowledge Discovery (DaWaK)* (2008), Springer-Verlag, pp. 166–175.

[SDTW09] Shalom S. A., Dash M., Tue M., Wilson N.: Hierarchical Agglomerative Clustering Using Graphics Processor with Compute Unified Device Architecture. *Signal Processing Systems, International Conference on 0* (2009), 556–561.

[SG03] Surazhsky V., Gotsman C.: Explicit surface remeshing. In *Proc. Symp. on Geometry Processing* (2003), ACM/Eurographics, pp. 20–30.

[SH07] Scheuermann T., Hensley J.: Efficient histogram generation using scattering on gpus. In *Proc. of the symposium on Interactive 3D graphics and games (I3D)* (2007), ACM, pp. 33–37.

[Sha04] Shamir A.: A formulation of boundary mesh segmentation. In *Int. Symp. on 3D Data Proc., Vis. and Transmission* (2004), pp. 82–89.

[Sha06] Shamir A.: Segmentation and shape extraction of 3D boundary meshes. *EUROGRAPHICS - State of the Art Reports* (2006), 137–149.

[Sha08] Shamir A.: A survey on mesh segmentation techniques. *Computer Graphics Forum 27*, 6 (2008), 1539 – 1556.

[She01] Sheffer A.: Model simplification for meshing using face clustering. *Computer-Aided Design 33*, 13 (2001), 925 – 934.

[SKK00] Steinbach M., Karypis G., Kumar V.: A comparison of document clustering techniques. In *KDD Workshop on Text Mining* (Boston, MA, 2000), pp. 109–111.

[SS05] Simari P. D., Singh K.: Extraction and remeshing of ellipsoidal representations from mesh data. In *Proc. of Graphics Interface* (2005), Canadian Human-Computer Communications Society, pp. 161–168.

[SSG03] Sifri O., Sheffer A., Gotsman C.: Geodesic-based surface remeshing. In *IMR* (2003), pp. 189–199.

[SSGH01] Sander P. V., Snyder J., Gortler S. J., Hoppe H.: Texture mapping progressive meshes. In *SIGGRAPH: Proc. of Computer graphics and interactive techniques* (2001), ACM, pp. 409–416.

[SZL92] Schroeder W. J., Zarge J. A., Lorensen W. E.: Decimation of triangle meshes. In *SIGGRAPH: Proc. of Computer graphics and interactive techniques* (1992), ACM, pp. 65–70.

[TK04] Takizawa H., Kobayashi H.: Multi-grain parallel processing of data-clustering on programmable graphics hardware. In *ISPA* (2004), pp. 16–27.

[VC04] Valette S., Chassery J.-M.: Approximated Centroidal Voronoi Diagram for Uniform Polygonal Mesh Coarsening. *EUROGRAPHICS 23*, 3 (2004), 381–389.

[VCP08] Valette S., Chassery J. M., Prost R.: Generic remeshing of 3D triangular meshes with metric-dependent discrete voronoi diagrams. *IEEE Transactions on Visualization and Computer Graphics 14*, 2 (2008), 369–381.

[VKC05] Valette S., Kompatsiaris I., Chassery J.-M.: Adaptive Polygonal Mesh Simplification With Discrete Centroidal Voronoi Diagrams. *Proc. Int. Conf. on Machine Intelligence ICMI* (November 2005), 655–662. Tozeur, Tunisia.

[WK05] Wu J., Kobbelt L.: Structure Recovery via Hybrid Variational Surface Approximation. *Proc. Eurographics 24*, 3 (2005), 277–284.

[WLR88] Willebeek-LeMair M., Reeves A. P.: Region growing on a hypercube multiprocessor. In *Proc. of the conf. on Hypercube concurrent computers and applications* (1988), ACM, pp. 1033–1042.

[WZS*06] Wang R., Zhou K., Snyder J., Liu X., Bao H., Peng Q., Guo B.:. *The Visual Computer 22*, 9 (2006), 612–621.

[XV96] Xia J. C., Varshney A.: Dynamic view-dependent simplification for polygonal models. In *Proc. of conf. on Visualization (VIS)* (1996), IEEE Computer Society Press, pp. 327–ff.

[XW08] Xu R., Wunsch D. C.: *Clustering.* Wiley-IEEE Press Series on Computational Intelligence, 2008.

[YLW06] Yan D.-M., Liu Y., Wang W.: Quadric surface extraction by variational shape approximation. In *Geometric Modeling and Processing - GMP (Lecture Notes in Computer Science)* (2006), pp. 73–86.

[ZG09] Zechner M., Granitzer M.: Accelerating k-means on the graphics processor via cuda. *Intensive Applications and Services, International Conference on 0* (2009), 7–15.

[ZZ06] Zhang Q., Zhang Y.: Hierarchical clustering of gene expression profiles with graphics hardware acceleration. *Pattern Recogn. Lett. 27*, 6 (2006), 676–681.