Analysis of Global Properties of Shapes

Aleksey Golovinskiy

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science
Adviser: Thomas Funkhouser

June 2010

Abstract

With increasing amounts of data describing 3D geometry at scales small and large, shape analysis is becoming increasingly important in fields ranging from computer graphics to robotics to computational biology. While a great deal of research exists on local shape analysis, less work has been done on global shape analysis. This thesis aims to advance global shape analysis in three directions: symmetry-aware mesh processing, part decomposition of 3D models, and analysis of 3D scenes.

First, we propose a pipeline for making mesh processing algorithms "symmetry-aware", using large-scale symmetries to aid the processing of 3D meshes. Our pipeline can be used to emphasize the symmetries of a mesh, establish correspondences between symmetric features of a mesh, and decompose a mesh into symmetric parts and asymmetric residuals. We make technical contributions towards two of the main steps in this pipeline: a method for symmetrizing the geometry of an object, and a method for remeshing an object to have a symmetric triangulation. We offer several applications of this pipeline: modeling, beautification, attribute transfer, and simplification of approximately symmetric surfaces.

Second, we conduct several investigations into part decomposition of 3D meshes. We propose a hierarchical mesh segmentation method as a basis for consistently segmenting a set of meshes. We show how our method of consistent segmentation can be used for the more specific applications of symmetric segmentation and segmentation transfer. Then, we propose a probabilistic version of mesh segmentation, which we call a "partition function", that aims to estimate the likelihood that a given mesh edge is on a segmentation boundary. We describe several methods of computing this structure, and demonstrate its robustness to noise, tessellation, and pose and intra-class shape variation. We demonstrate the utility of the partition function for mesh visualization, segmentation, deformation, and registration.

Third, we develop a system for object recognition in 3D scenes, and test it on a large point cloud representing a city. We make technical contributions towards three key steps of our system: localizing objects, segmenting them from the background, and extracting features that describe them. We conduct an extensive evaluation of the system: we perform quantitative evaluation on a point cloud consisting of about 100 million points, with about 1000 objects of interest belonging to 16 classes. We evaluate our system as a whole, as well as each individual step, trying several alternatives for each component.

Acknowledgements

I would first like to thank my advisor, Thomas Funkhouser, who put more time, energy, and work into this thesis than could reasonably be asked for, and then some. The high standard he set shamed me into trying to keep up.

I was fortunate to have had the opportunity to work with other computer graphics faculty at Princeton: Adam Finkelstein and Szymon Rusinkiewicz. I learned a great deal from their different approaches to research. I also had the benefit of working with and learning from my other co-authors: Heather Stoddart Barros, Xiaobai Chen, Forrester Cole, Vladimir G. Kim, Alex Limpaecher, Wojciech Matusik, Hanspeter Pfister, Joshua Podolak, and Philip Shilane. Beyond my co-authors, I had help from too many people to cite, but Aleksey Boyko and Yaron Lipman deserve a special mention for useful discussions and code. Additional thanks must go to Forrester Cole and Christopher DeCoro for helping to create videos for me, and to the Tiggraph group for reviewing preliminary drafts.

For providing 3D surface models, we thank Aim@Shape, Viewpoint, Stanford, Cyberware, Lior Shapira, Marco Attene, Daniela Giorgi, Ayellet Tal, and others. For providing LIDAR data, we thank Neptec, John Gilmore, and Wright State University and thank Kelly Hageman for helping to truth the data. We also thank DARPA's URGENT program, and BAE Systems for including us, especially Erik Sobel, Matt Antone, and Joel Douglas.

I thank the following funding sources for sponsoring this work: National Science Foundation under grants CCR-0093343, 11S-0121446, IIS-0612231, CCF-0347427, CCF-0702672, CNS-0406415, and CNS-0831374, the Air Force Research Laboratory under grant FA8650-04-1-1718, the Sloan Foundation, and Google.

Finally, I cannot help but thank people who supported me outside of work. My friends in Princeton made for a fun graduate school experience. My grandparents, aunt, uncle, and cousins were always inspiring and supportive. My parents, having bravely immigrated to a different continent, encouraged and enabled me to follow my interests, and made graduate study possible for me. Rebecca shared with me love, support, and many Indian dinners, while showing uncharacteristic patience towards my pre-deadline schedules.

To my family, who made things easy, and to Rebecca, who made things fun.

Contents

Chapter 1

Introduction

1.1 Motivation

A broad goal of computer science is the design of algorithms and systems that allow computers to interpret and understand data. This goal is important in a number of domains: text documents, images, and audio, to offer a few. In this spirit, this thesis describes algorithms that advance the analysis of another important domain: 3D geometry. The need for algorithms for processing geometry is driven on the one hand by the many application domains that use such data, and on the other hand by the improvements in 3D scanning and modeling technologies that have led to an explosion of available data that promises to continue.

3D models have long been used in the entertainment industry for games, movies, and virtual reality. The use of geometrical data for medical and biological applications has been growing: analysis of CAT scans, MRI scans, fMRI scans is a busy topic of research. In another branch of biology, the repository of known protein geometry has been growing, and there is a great deal of research investigating the use of this data. Computer Aided Design has many applications that benefit from geometry analysis. At a larger scale, 3D scans of entire cities are becoming available. It seems increasingly likely that the geometry of much of the world, at scales small and large, will be captured within a decade. This explosion of 3D data, together with its usefulness to multiple domains, underlines the fundamental need for the development of algorithms that analyze 3D geometry.

1.2 Research Context

A great deal of research has been done in shape analysis. Because this thesis is concerned with a variety of applications, it is useful to group related algorithms by their scale of influence, ranging from local to global methods.

Much previous work in shape analysis focuses on local, patch-scale properties. For example, discrete estimates of local curvature have been used for applications ranging from sampling points to finding correspondences to shape simplification. Discrete variants of the Laplace-Beltrami operator [108] and first and second fundamental form [71] have been proposed for applications that involve manipulating meshes while preserving local shape. The concept of mesh saliency [66] was introduced to measure "importance" of mesh surfaces in a way inspired by low-level human visual system cues. A variety of local geometry descriptors, such as spin images [51] can be used to characterize the local patch of geometry. However, due to their small support size, local properties alone have a limited power in describing the geometry of an object.

At a larger scale, many shape analysis methods compute global shape structures such as skeletons and segmentations through a sequence of steps that consider local properties. Mesh simplification using QSlim [35], for example, operates through a series of edge collapses ordered by the distance to the original surface. Similarly, iterative merging of faces has been used to cluster polygonal surfaces [36]. Laplacian mesh contraction [13] was proposed to create a mesh skeleton. These methods share the feature that each step is performed in a local, greedy manner, but the result contains global information about the geometry; some of our methods follow this approach.

Finally, at the largest scale, some shape analysis methods extract properties that are inherently global, which is where much of this thesis focuses. Because these properties consider the entirety of an object, they are often able to robustly extract information that is difficult to discern by looking at small patches of geometry at a time. Below we summarize categories of global shape properties that have been used in shape analysis.

One example of an inherently global cue is global symmetry. Information about how symmetric an object is with respect to a set of transformations depends on relationships between distant parts of the objects. Therefore, both detection and use of symmetry in mesh analysis and processing require non-local approaches. Methods have been proposed for detecting symmetry in 3D objects (e.g. [92, 83]), as well as for discovering structural regularity ([91]). Such methods have been used for a variety of applications, including model repair ([91, 112]), compression([83]), and segmentation([83, 92]).

Spectral methods offer another class of inherently global properties for shape analysis. This class of methods for 3D shapes is analogous to Fourier techniques for signals. Spectral methods take advantage of eigenvectors and eigenvalues of linear operators, usually related to the Laplace-Beltrami operator. These quantities have several benefits: they can be interpreted as harmonics of the surface, decomposing it into frequency bands; they reveal global information about the shape (eigenvalues of graph Laplacians in general are related to global graph properties [127]); and, with appropriate choice of operator, they reveal intrinsic shape properties that are invariant to articulation. Spectral methods have been used for applications including mesh compression [53], segmentation [73], remeshing [26], and registration [50].

Global shape descriptors form another set of large-scale shape properties. The difference between these and local shape descriptors often lies in application rather than technique: most local descriptors can be scaled up and applied to entire models. However, the scope of these properties is entirely different: rather than describe a local patch, these descriptors aim to describe an entire object. Such descriptors include histograms of properties such as distances to center of mass [8] and pairwise point distances [88], spherical functions describing maximal distance from center of mass [100], renderings of shapes from positions uniformly centered on a sphere [22], and the Gaussian of the distance transform of a shape and its rotationally invariant representation [57].

Finally, another branch of research in global shape properties focuses on defining pairwise distances between points on or in a shape. Such a pairwise distance measure is useful for applications including segmentation, skeletonization, registration, point sampling, and shape matching. Geodesic distances have long been proposed (e.g. in [55]) for applications such as segmentation; the local maxima of these distances has been useful for finding "extremal" points. Diffusion distances [25] offer a more robust distance between a pair of points by considering the average distance along the paths connecting these points. [74] proposed a metric that was explicitly constructed to increase more along potential part boundaries. Finally, an intrinsic distance between points inside a shape was proposed in [99].

1.3 Overview

In this thesis, we seek to advance algorithms that help computers interpret shape by extracting global properties. We focus on three directions: analysis of symmetry in objects, analysis of part decomposition of objects, and analysis of 3D scenes.

1.3.1 Symmetry

Perfect, partial, and approximate symmetries are pervasive in 3D surface meshes of real-world objects. However, current digital geometry processing algorithms generally ignore them, instead focusing on local shape features and differential surface properties. We investigate how detection of large-scale symmetries can be used to guide processing of 3D meshes. We establish a framework for mesh processing that includes steps for symmetrization (applying a warp to make a surface more symmetric) and symmetric remeshing (approximating a surface with a mesh having symmetric topology). These steps can be used to enhance the symmetries of a mesh, to decompose a mesh into its symmetric parts and asymmetric residuals, and to establish correspondences between symmetric mesh features. Applications are demonstrated for modeling, beautification, and simplification of nearly symmetric surfaces.

1.3.2 Part Decomposition

Decomposition of 3D objects into function parts is a fundamental problem in computer graphics as well as computer-aided design. A part decomposition yields information about the structure of the underlying object, and can also by used to guide some mesh processing algorithms, including shape-based retrieval, texture mapping, skeleton extraction, and modeling. Despite considerable research, part decomposition remains a difficult, unsolved problem. We describe several advances in this direction.

We introduce an hierarchical segmentation algorithm for meshes, and extend this algorithm to consistently segment a set of meshes; for example, to extract all the legs, seats, and backs of a group of chairs. We show how to use our system to symmetrically segment meshes, and also how to use it to transfer segmentations.

Even modern mesh segmentation algorithms (including ones we propose) often do not produce desirable results. We investigate a method that randomizes segmentation algorithms to produce a more robust, continuous version of a mesh segmentation. The general strategy is to generate a random set of mesh segmentations and then to measure how often each edge of the mesh lies on a segmentation boundary in the randomized set. The resulting "partition function" defined on edges provides a continuous measure of where natural part boundaries occur in a mesh, and the set of "most consistent cuts" provides a stable list of global shape features. We describe methods for generating random distributions of mesh segmentations, studies sensitivity of the resulting partition functions to noise, tessellation, pose, and intra-class shape variations, and investigates applications in mesh visualization, segmentation, deformation, and registration.

1.3.3 Scene Analysis

With the increasing amount of data available describing outdoor 3D scenes, it is important to develop geometry analysis algorithms that deal not only with single objects, with with scenes. Towards this end, we investigate the design of a system for recognizing objects in 3D point clouds of urban environments.

The system is decomposed into four steps: locating, segmenting, characterizing, and classifying clusters of 3D points. Specifically, we first cluster nearby points to form a set of potential object locations (with hierarchical clustering). Then, we segment points near those locations into foreground and background sets (with a graph-cut algorithm). Next, we build a feature vector for each point cluster (based on both its shape and its context). Finally, we label the feature vectors using a classifier trained on a set of manually labeled objects. The paper presents several alternative methods for each step. We quantitatively evaluate the system and trade-offs of different alternatives in a truthed part of a scan of Ottawa that contains approximately 100 million points and 1000 objects of interest. Then, we use this truth data as a training set to recognize objects amidst approximately 1 billion points of the remainder of the Ottawa scan.

1.3.4 Goal

While the above topics have slightly different domains, they share a common goal: the extraction of global, high-level properties and investigation of applications that benefit from these properties. In our investigation on symmetry, we focus on extracting and using correspondences induced by symmetries, which are global features relating far-away parts of objects. Segmentation is a high-level abstraction capturing the global structure of an object; through our partition function, we introduce a continuous version of segmentation, which also captures global features. Finally, in our work on scene analysis, we use global shape descriptors to recognize objects.

The remainder of the thesis is organized into the following chapters. In Chapter 2, introduce a pipeline for processing 3D objects in a "symmetry-aware" manner. In Chapter 3, we carry out two investigations related to part decomposition of objects: consistent segmentation of a set of objects, and a structure representing a continuous version of a part decomposition. In Chapter 4, we describe a framework for object recognition in outdoor scenes. Finally, in Chapter 5 we present some avenues for future work.

Chapter 2

Symmetry-Aware Processing

2.1 Introduction

Symmetry is ubiquitous in our world. Almost all man-made objects are composed exclusively of symmetric parts, and many organic structures are nearly symmetric (e.g., bodies of animals, leaves of trees, etc.). It is almost impossible to find a real-world object that does not have at least one nearly perfect symmetry and/or is not composed of symmetric parts. Moreover, symmetry is an important cue for shape recognition [30], as humans readily notice departures from perfect symmetry.

For decades, however, mesh reconstruction and processing algorithms in computer graphics have largely ignored symmetries. Most algorithms operate as sequences of mesh processing operations based on local shape features and/or differential surface properties. As a result, they have difficulty reproducing and preserving global shape properties, such as symmetry.

Consider simplification, for example – when presented with an input mesh for a nearly symmetric object (e.g., a face), a simplification algorithm should produce a nearly symmetric mesh. However, to our knowledge, there is no current algorithm that satisfies this basic requirement. Certainly, if the input is perfectly symmetric, then the problem is trivial – simply process half of the mesh and then copy the result. However, if the underlying surface is symmetric but the mesh topology is not, or if the underlying surface is only approximately symmetric, then standard simplification algorithms fail to preserve symmetries present in the underlying object (Figure 2.10c). The result is potential artifacts in physical simulations, manufacturing processes, animations, and rendered images (e.g., asymmetric specular highlights).

Recently, researchers have introduced several methods for detecting and characterizing the symmetries in 3D data. For example, Zabrodsky et al. [125] provided a measure of approximate symmetry with respect to any transformation, and Mitra et al. [83] and Podolak et al. [92] have described algorithms for extracting the most significant approximate and partial symmetries of a 3D mesh. While symmetry analysis methods like these have been used to guide high-level geometric processing operations, such as registration, matching, segmentation, reconstruction, reverse engineering, editing, and completion, they have only begun to be incorporated into low-level mesh processing algorithms.

The main goal of this section is to investigate ways in which symmetry analysis can guide the representation and processing of 3D surface meshes. To support this goal, we make the following contributions. First, we describe an algorithm for geometric symmetrization – i.e., deforming a surface to respect a given set of symmetries while retaining its shape as best as possible. Second, we describe an algorithm for topological symmetrization – i.e., remeshing a surface so that symmetric regions have consistent mesh topology. Third, we propose a "symmetry-aware" mesh processing framework in which geometric and/or topological symmetrization algorithms provide high-level shape information (symmetric correspondences and asymmetric residuals) that can guide mesh processing applications to produce more symmetric results for approximately symmetric inputs. Finally, we demonstrate applications of this framework for surface beautification, symmetry enhancement, attribute transfer, and simplification.

2.2 Background and Previous Work

Understanding the symmetries of shapes is a well studied problem with applications in many disciplines. Perfect symmetries are common in CAD models and used to guide compression, editing, and instancing [77]. However, only considering perfect symmetries is of limited use in geometric processing, in general. First, the presence of noise, numerical round-off error, or small differences in tessellation can cause models of objects that are in fact symmetric to lack perfect symmetry. Second, many asymmetric objects are composed of connected parts with different symmetries. Finally, most organic objects exhibit near, but imperfect, symmetries (leaves of trees, human bodies, etc.), and understanding those types of symmetry is important, too. Thus, it is useful to have methods to detect and utilize partial and approximate symmetries.

Towards this end, Zabrodsky et al. [125] defined the *symmetry distance* of a shape with respect to a transformation as the distance from the given shape to the closest shape that is perfectly symmetric with respect to that transformation. They provide an algorithm to find the symmetry distance for a set of connected points for any given reflective or rotational transformation. Mitra et al. [83] and Podolak et al. [92] find a set of prominent symmetries by having points on a mesh vote for symmetries in a process similar to a Hough transform.

Measures for partial and approximate symmetry of this type have been used in a variety of computer vision applications. Perhaps the earliest example is by [111], who used deformable models with symmetry-seeking forces to reconstruct 3D surfaces from 2D images. Zabrodsky et al. used a continuous measure of symmetry for completing the outline of partially-occluded 2D contours [124], for locating faces in an image, determining the orientation of a 3D shape [125], for reconstructing 3D models from images, and for symmetrizing 3D surfaces [126].

More recently, symmetry analysis has received attention in computer graphics. Kazhdan et al. [56] constructed a symmetry descriptor and used it for registration and matching. Podolak et al. [92] used a symmetry transform for surface registration, shape matching, mesh segmentation, and viewpoint selection. Mitra et al. [83] described a method to extract a discrete set of significant symmetries and used them for segmentation and editing. Thrun et al. [113] used local symmetries and used them for completion. Gal et al. [34] developed local shape descriptors to look for approximate symmetries in 3D surfaces and used them for visualization and matching. Mills et al. [81] utilized approximate symmetries to guide reverse engineering of CAD structures from range scans. Simari et al. [107] decomposed meshes into a hierarchical tree of symmetric parts to be used for compression and segmentation. Finally, Martinet et al. [77] has detected perfect symmetries in parts of scenes and used them to build instancing hierarchies.

Perhaps closest to our work is the work of Mitra et al. [82], where a method of symmetrizing the geometry of meshes is presented. Their technique is similar to the one we present in Section 2.4. However, we take the output of geometric symmetrization and go further, providing algorithms for symmetric remeshing and a framework for making mesh-processing algorithms symmetry-aware.

2.3 Overview

The goal of our work is to provide tools for symmetry-aware processing of 3D surface meshes. We propose a multi-step processing framework, in which approximate and partial symmetries are detected, preserved, exploited, and sometimes even enhanced as meshes are processed. To support this framework, we provide the following tools, which typically will be used in the sequence of steps shown in Figure 2.1:

1. **Symmetry analysis:** the mesh is analyzed to detect perfect, approximate, and partial symmetries. The output of this step is a set of transformations (e.g., planes of reflection), each

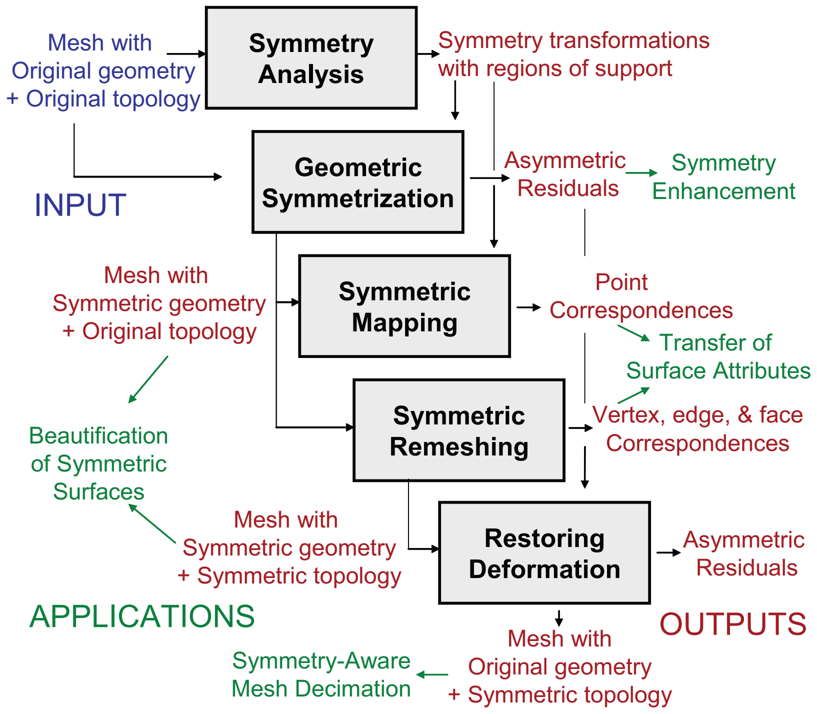


Figure 2.1: Symmetry-aware mesh processing framework.

with a list of vertices indicating the subset of the surface mapped approximately onto itself by the transformation. For this step, which is not a focus of this paper, we use methods previously described in [92] and [83].

2. **Geometric symmetrization:** the surface is warped to make it symmetric with respect to a given set of transformations. The primary output of this step is a “symmetrized” mesh having the same topology as the original, but with geometry that is perfectly symmetric up to the resolution of its tessellation. A secondary output is a set of asymmetric residuals storing the vector difference between the original and symmetrized position of every vertex, which can be used to compute an inverse to the symmetrizing warp.

3. **Symmetric mapping:** correspondences are established between vertices and their images across every symmetry transformation. The output of this step is a dense set of point pairs, where one point is associated with a vertex and the other is associated with a face and its barycentric coordinates. These point pairs provide a mapping between symmetric surface patches.

4. **Symmetric remeshing:** the surface is remeshed so that every vertex, edge, and face has a one-to-one correspondence with another across every symmetry. The output of this step is a

new mesh with perfectly symmetric topology, along with a list of topological correspondences.

5. **Restoring deformation:** the inverse of the symmetrizing warp is applied to the symmetrically remeshed surface to restore the original geometry. The output of this step is a mesh that is topologically symmetric, but geometrically approximates the input mesh.

The motivating idea behind this framework to provide tools that can help mesh processing algorithms to preserve large-scale symmetries present in 3D objects. Our general strategy is to factor a 3D surface into a symmetric mesh and its asymmetric residual and then to perform analysis on the symmetric mesh to gain insight into its symmetric structure. We transfer knowledge about symmetric structure back onto the original geometry so that it can be preserved and exploited as the surface is processed.

In the following sections, we investigate algorithms to support this symmetry-aware mesh processing framework, focusing on the most challenging steps: geometric symmetrization (Section 2.4) and symmetric remeshing (Section 2.5). Thereafter, in Section 2.6, we describe potential applications and present prototype results. Finally, we conclude with a discussion of limitations and topics for future work.

2.4 Geometric Symmetrization

Our first objective is to provide an algorithm that can take a given surface mesh and output a new mesh with similar shape that is symmetric with respect to a given set of transformations. More formally, given a mesh M and a set of symmetry transformations, each having a possibly local region of support on the mesh, our goal is to find the most shape-preserving warp W that produces a new mesh M' with the same topology as M, but where every vertex of M' is mapped onto corresponding points on the surface of M' by all of its symmetry transformations.

This objective is similar to classical problems in non-rigid alignment for morphing of 3D surfaces, medical imaging, surface reconstruction, and several other fields. The challenge is finding both symmetric point correspondences and the warp that aligns them simultaneously. Mitra et al. [82] solve this problem (while also detecting symmetries) using Generalized Hough Transform and clustering algorithms [82]. Since our problem is a bit simpler (symmetry transformations have already been detected), we following a more traditional iterative approach [16], employing an algorithm that greedily minimizes alignment error while allowing increasingly non-rigid deformation [6, 109, 90]. At each iteration, we first propose correspondences from every vertex in the mesh M to its closest compatible

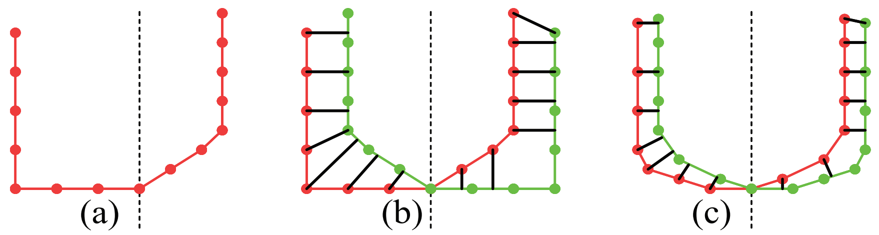


Figure 2.2: Schematic of one iteration in our symmetrization process (in 2D). (a) Given a curve (red) and a symmetry transformation (reflection across the dotted line), (b) we find correspondences between vertices and the closest to point on the reflected curve (green), and (c) solve for new vertex positions that minimize an error function based on those correspondences.

point on the transformed surface for every symmetry. Then, given these correspondences, we solve for new vertex positions that minimize a symmetrizing error function (Figure 2.2). These two steps are iterated until the mesh is fully symmetric (i.e., every vertex transformed by all its symmetries produces a point directly on a face of the mesh).

Our symmetrizing error function balances the primary goal of making the surface more symmetric with the secondary goals of retaining its original shape and position with three error terms:

$$E(M) = \alpha E_{sym}(M) + (1-\alpha)(E_{shape}(M) + \beta E_{disp}(M))$$

The first two error terms are the important ones, as they balance the trade-off between deviations from perfect symmetry and deformations of the surface. The first term, E_{sym}, measures the sum of squared distances between vertices of the mesh and the closest point on the transformed surface. For the second term, E_{shape}, we use the shape preservation function proposed by [90], which minimizes a measure of warp distortion. Any deformation error function would work, but this choice has the advantage of being quadratic in vertex positions. This deformation error is not rotationally invariant, but the symmetrizations performed in our experiments do not have large rotational components. The last term, E_{disp}, measures overall displacement using the sum of squared distances between current and original vertex positions. It is required to penalize global translations because E_{shape} is translationally invariant and because the surface is being warped onto itself (in contrast to traditional alignment problems where either the source or target is fixed in space). We set the weight of this error term very low relative to the others ($\beta = 1/100$), and so it has very little influence on the output surface's shape. Since all three terms of the error function are quadratic in the positions of the vertices, we can solve for the minimal error at each iteration with a least-squares solution to a linear system.

Our implementation contains several simple features that help provide stability and speed as the optimization proceeds. First, it uses multiresolution surface approximations to accelerate convergence and avoid local minima. Prior to the optimization, the input mesh is decimated with *Qslim* [35] to several nested levels of detail. Then, coarser levels are fully symmetrized and used to seed the initial vertex placements for finer levels (new vertices added at each finer level are positioned relative to the current ones using thin-plate splines). Within each level, further stability is gained by slowly shifting emphasis of the error function from shape preservation ($\alpha = 0$) to full symmetrization ($\alpha = 1$). Finally, for each vertex, we use a k-d tree to help find the closest point on the transformed surface, and only retain correspondences to a closest point whose transformed normal does not point in the opposite direction. Overall, compute times on a 3Ghz processor range from 4 seconds for the 1,166 vertex model of the dragon in Figure 2.8 to 10 minutes for a 240,153 vertex face scan.

For instance, consider Figure 2.3, which shows the result of symmetrizing a bust of Max Plank with respect to reflection across a single vertical left-right plane. Note that the original input mesh (Figure 2.3a) is quite asymmetric, as seen by the misalignment of the surface (red) and its reflection (green) in the bottom images of Figure 2.3a – i.e., significant shape features (eyes and ears) do not map onto their symmetric counterparts when reflected across the plane. However, we are able to find a non-rigid warp that aligns those features while producing a symmetric mesh with a small amount of shape distortion. The symmetrized mesh, shown Figure 2.3b, is perfectly symmetric up to the resolution of the mesh, as indicated by the high-frequency interleaving of the original surface (red) with its reflection (green) in the bottom images of Figure 2.3b.

A more complicated example demonstrates symmetrization across partial, approximate planes of symmetry in the Stanford Bunny (Figure 2.4). Using the method of [92], the bunny was automatically segmented into two symmetric parts (the head and the body), each supporting a plane of partial symmetry. Of course, both of these parts are highly asymmetric, as can be seen from top-right image in Figure 2.4, where each part of the bunny (red) is shown along with its reflection (green and blue) over its symmetry plane. Note how poorly the ears and feet align with their reflections. However, our geometric symmetrization algorithm is able to warp both parts into alignment with their reflections simultaneously while retaining significant shape features (e.g., ears, feet) and blending symmetries across the intermediate region of the neck. Note that the crease between the feet is preserved although the symmetry plane does not run through it on the original model, as the surface was warped to align with the plane.

The output of the process is not only a symmetrized mesh, but also a *symmetric map*, a set

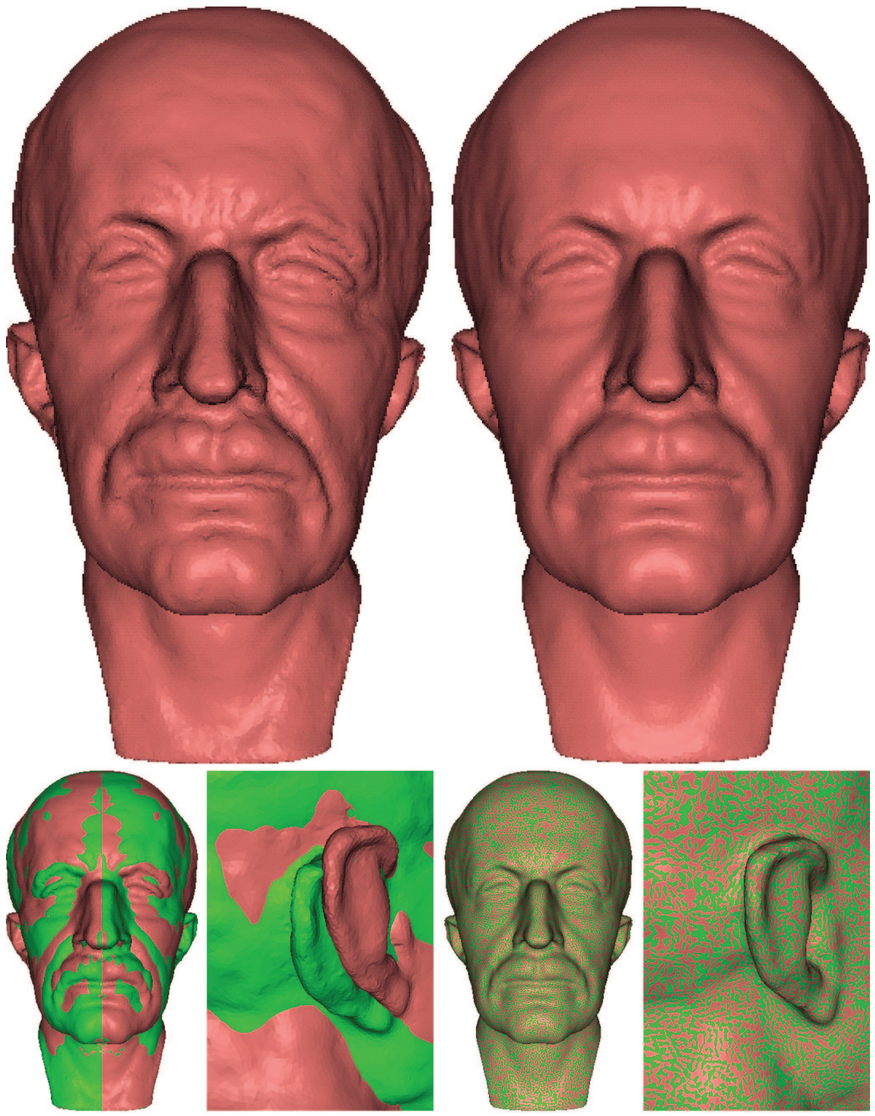


Figure 2.3: Symmetrizing Max Planck. The model of a bust of Max Planck (a) asymetric, as can be seen by overlaying the mesh (red) with its reflection (green) in the bottom images. Our method symmetrizes its geometry (b), while retaining and aligning sharp features like eyes, mouth, and ears.)

.

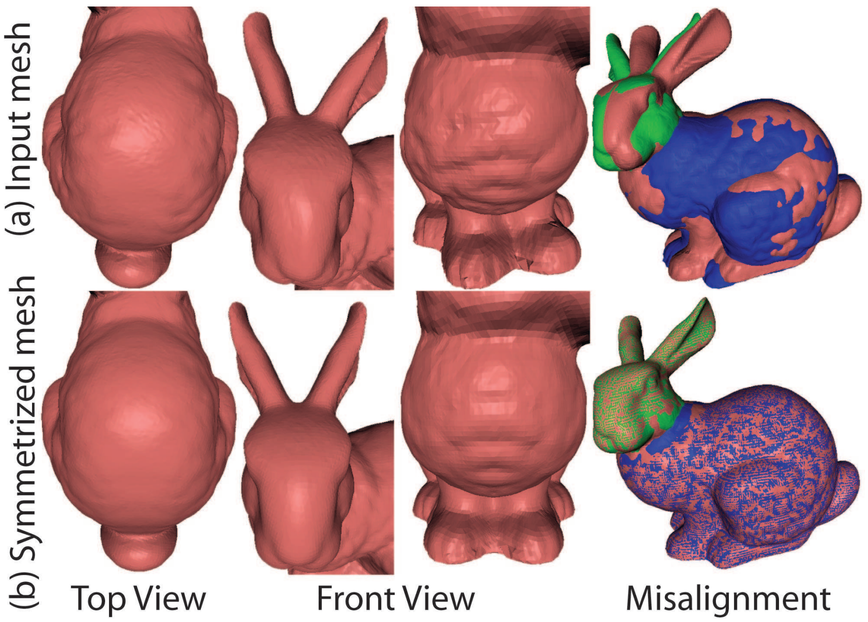


Figure 2.4: Symmetrizing the bunny. The top row shows the original bunny, while the bottom row shows the symmetrized result. The bunny contains partial symmetries with respect to planes through the body and through the head. Our method symmetrizes both parts of the bunny while performing a shape-preserving blend in between, preserving features such as the eyes and feet. The right pair of images show the asymmetry of the original model and the accurate alignment of symmetric parts in our result.

of point correspondences where every vertex is associated with a point on the surface across every symmetry transformation. We store the corresponding points in barycentric coordinates with respect to triangles of the mesh so that they deform with the surface (e.g., when we apply the inverse of the symmetrizing deformation to restore the original geometry). This is a key point, since it allows us to transfer the symmetric map learned from the perfectly symmetric surface back to the asymmetric one.

2.5 Symmetric Remeshing

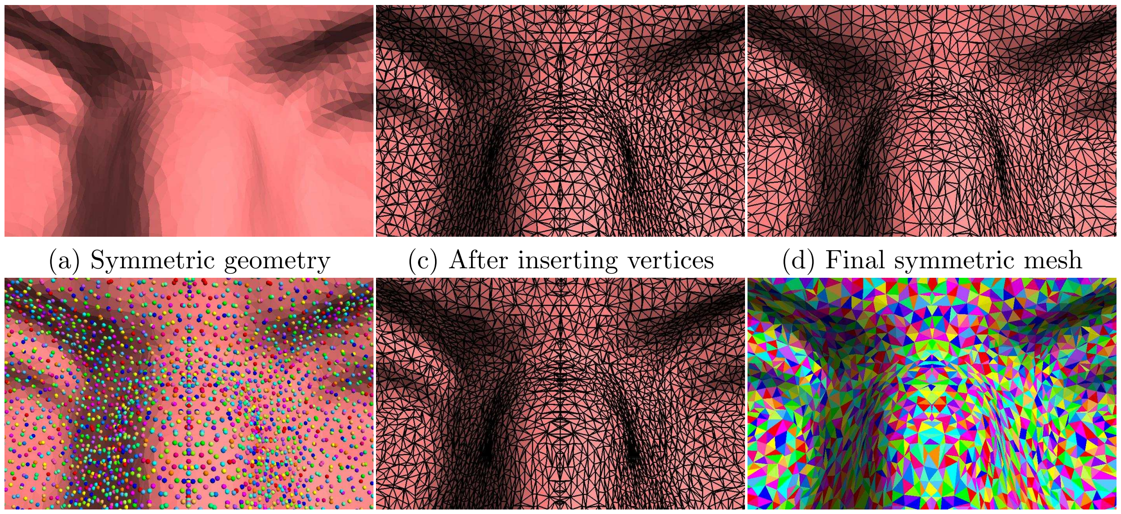
Our second objective is to develop an algorithm that can take a geometrically symmetrized mesh with arbitrary topology and remesh it so that every vertex, edge, and face has a direct correspondence with another with respect to every symmetry transformation. Our motivation is to provide a topology that not only reflects the symmetric structures of the object, but also can provides efficiency in representation and manipulation due to topological redundancies (e.g., compression), cues for preservation of symmetries during topological modifications (e.g., simplification), and symmetric sampling to avoid assymetric artifacts in photorealistic renderings and physical simulations (e.g., boundary element methods).

This problem is a special case of compatible remeshing [3, 59]. Given the symmetric map from every vertex to a point on the surface for every symmetry transformation provided by the geometric symmetrization algorithm, we aim to find the mesh with perfectly symmetric topology that has the least geometric error and/or fewest extra vertices.

A strawman approach that may be appropriate for highly symmetric and/or oversampled meshes is to partition the mesh into "asymmetric units" and then copy the topology from one instance of others in correspondence and then stitch at the boundaries. For a single planar reflection, this would entail cutting the mesh along the plane, throwing away the mesh connectivity on one side (M_t), and then copying the connectivity over from the other side (M_s). While this simple method would provide symmetric topology with the same number of vertices as the original mesh, it would produce an asymmetry in the quality of the geometric approximation (M_s would have the quality of the original surface, but M_t would have blurring where edges oriented appropriately for the geometry of M_s are not appropriate for M_t), and it would produce artifacts where the topology of M_s provides a poor approximation for M_t.

There are many methods in the literature to overcome this problem, most of which introduce a large number of extra vertices to capture the geometric variations of both M_s and M_t. For example, one way is to create an overlay meta-mesh that contains the original vertices of both M_s and M_t along with new vertices at all edge-edge intersections [4, 65]. Another way is to map M_s and M_t to a common base domain (e.g., a sphere [4], or a simplified triangle mesh [65, 94]) and then remesh with semi-regular connectivity until all geometric features are resolved. Alternatively, it is possible to create a meta-mesh M_{st} by inserting all the vertices of M_s into M_t, and vice-versa, and then iteratively swapping edges until a compatible mesh topology is achieved [3]. These methods all produce compatible mesh topology and so could be used for symmetric remeshing. However, the resulting mesh would usually be significantly over-sampled.

We provide a simple method to address the problem: *compatibility-preserving mesh decimation* (or, in our case, *symmetry-preserving mesh decimation*). Our general approach is to use any of the above methods to produce compatible mesh topology with vertices from both M_s and M_t, and then to decimate the resulting meta-mesh with a series of edge collapse operations that operate on corresponding edges in lock-step. Specifically, we build clusters of edges whose vertices are in symmetric correspondence and then follow the same basic approach as the original *Qslim* algorithm [35], however working on clusters rather than individual edges. We load the clusters into a priority queue sorted by the Quadric Error Measure (QEM) of the edge with highest error in each cluster, and then


(a) Symmetric geometry (c) After inserting vertices (d) Final symmetric mesh

(b) Point correspondences (d) After edge flips (e) Final symmetric mesh

Figure 2.5: Symmetric remeshing. The input is shown in the left column (the symmetrized bust of Max Planck, zoomed to the bridge of the nose); intermediate steps are shown in the middle column, and the output is shown in the right column. The final mesh has perfectly symmetric topology (as shown by colored face correspondences in (e)) and still approximates the surface well with the original number of faces.

we iteratively collapse all edges in the cluster with minimal error until a desired number of triangles or a maximum error has been reached. Since all edges in the same cluster are processed atomically, the method is guaranteed to maintain topological symmetries as it decimates the mesh. Yet, it still provides a good approximation of the original surface, as QEMs approximate deviation from the original surface.

This method is similar in goal to the method of [59], which copies the mesh topology of M_s onto M_t and then optimizes the positions and number of vertices to match the geometry of both M_s and M_t with a combination of smoothing and refinement operations. The difference is that we first produce an over-sampled mesh with vertices from both M_s and M_t, and then "optimize" it to minimize the QEM by decimation. Since our process is seeded directly with (a conservatively large set of) compatible vertices and edges from both M_s and M_t, the optimization starts from an initial configuration that encodes features from the entire mesh. So, our challenge is mainly to decide which vertices and edges can be removed, rather than discovering suitable places for new vertices from scratch. As a result, it is easy to produce compatible mesh topologies for any number of surface regions with any number of vertices.

We have experimented with this approach using an algorithm based on the Connectivity Transformation technique of [3] to form an over-sampled mesh with symmetric topology prior to decimation. Given a geometrically symmetrized mesh and a set of vertex-point correspondences (Figure 2.5a-b),

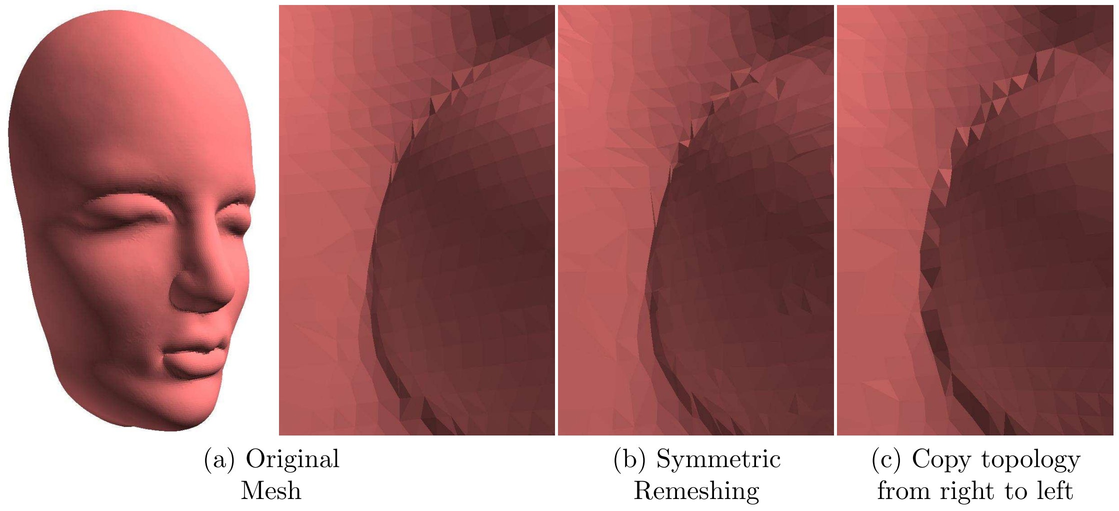

(a) Original Mesh (b) Symmetric Remeshing (c) Copy topology from right to left

Figure 2.6: A zoomed-in comparison the crease along left side of the nose on the surface of the mask shown on the left. Note how our approach (b) does not produce blurring when compared to the original (a), while the simple alternative (c) of copying topology from one side to the other does.

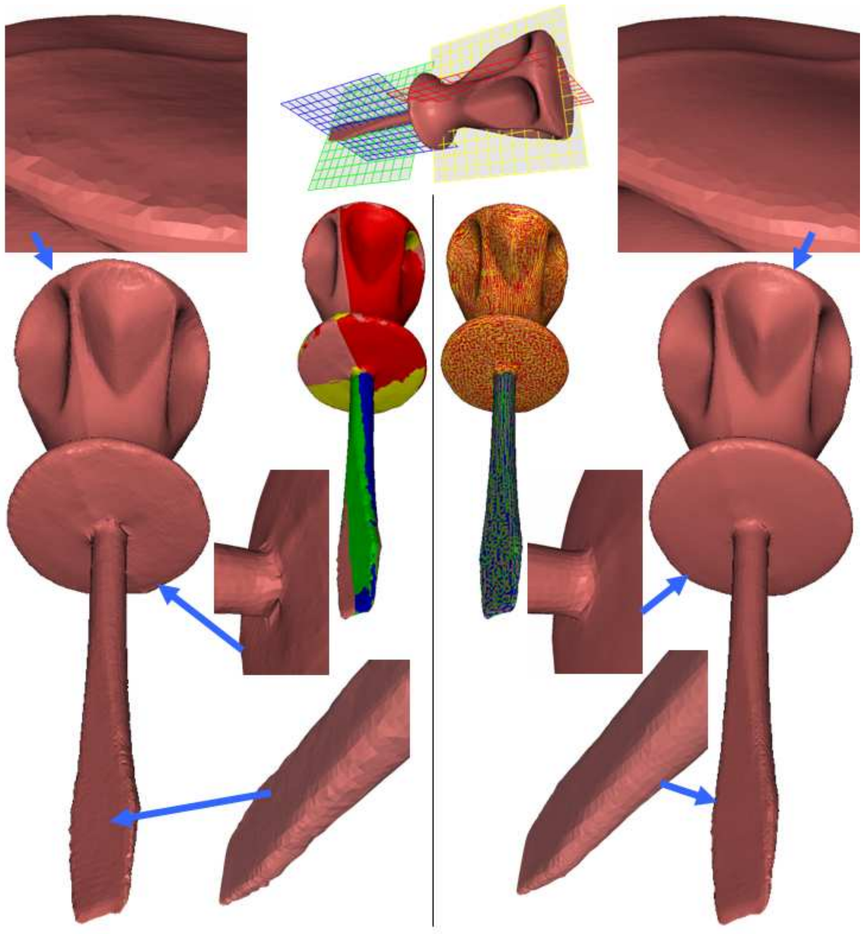
we first produce a meta-mesh M_{st} with symmetric vertex correspondences by inserting all the vertices of M_s into M_t, and vice-versa, splitting faces into three when an inserted vertex maps to the interior of an existing face (Figure 2.5c). We then swap edges in order of an error function that measures the differences in the QEM for edge midpoints before and after the swap, plus a quadratically growing penalty for swaps of an edge multiple times, plus an infinite penalty for any swap that would generate a topological fin in the mesh or break a greater number of symmetric correspondences than it creates. The process terminates when all edges are found to be in symmetric correspondence, or when the minimal error of any cluster exceeds some preset threshold. We have not implemented the edge-crossing constraint and termination criterion of [3], as it only guarantees convergence for meshes on a plane [45]. However, we find that our method finds symmetric correspondences for all but few edges in practice (99.9% in all of our examples). For the remaining edges, we simply copy those edges from one asymmetric unit to the other(s). The net result is a mesh with fully symmetric topology containing approximately twice as many vertices as the original (Figure 2.5d). We give that mesh as input to the symmetry-preserving version of Qslim to produce the final result – a topologically symmetric mesh with a user-specified number of faces or geometric error. Figure 2.5e-f shows the result for decimation to the number of faces in the original mesh (98K). Compute times for the entire symmetric remeshing process range between tens seconds for the dragon and two hours for the armadillo on a 3GHz processor. Of course, this process must be done only once per model.

It is difficult to make comparisons of our symmetric meshing method to others, since our problem is somewhat different from previous ones. However, to validate that our approach provides benefits over the simple strawman approach described earlier in the section (copy the topology of the right side over to the left), we provide a comparison of symmetrically remeshed surfaces of a mask along the left crease of the nose (Figure 2.6). Note how our method (middle) produces a mesh that retains sharp features of the original (left), whereas the simpler approach (right) suffers from blurring due to poorly oriented edges. Besides these differences in surface quality, our method has the additional advantage that it works without modification for partial and multiple symmetries of any type of transformation, and produces symmetric mesh topology at any user-selected face count.

2.6 Applications

The main theme of this paper is that awareness of symmetries can and should be incorporated into mesh processing algorithms. Since objects with perfect and/or approximate symmetries are prevalent in our world, and since symmetries are often critical to an object's function and/or a human's perception of it, we believe that algorithms processing 3D models should understand their symmetries and preserve them. In this section, we investigate how this can be done for several classes of applications.

Roughly speaking, applications can be divided into classes according to what type of mesh data they process, and almost equivalently, what type of symmetry information they can exploit: (1) Some applications are concerned mainly with creating new geometry (e.g., surface scanning, interactive modeling, etc.). For this class, geometric symmetrization provides a useful tool for coercing the geometry of approximate input (e.g., scanned points, sketched surfaces, etc.) to become more, less, or perfectly symmetric to match the intended structure of the object being modeled. (2) Other applications are concerned with manipulating attributes associated with local regions of a surface (e.g., texture mapping, signal processing, etc.). For them, symmetric mapping provides a way to blend and transfer attributes between symmetric regions. (3) Still other applications are concerned with the manipulating the topology of a mesh (e.g., remeshing). For those applications, symmetric remeshing provides an automatic way to coerce the mesh topology to respect the symmetric structure of an object and provides correspondence information that can be used to preserve topological symmetries as the mesh is processed further. Finally, of course, there are applications that can exploit all three types of symmetry information simultaneously (e.g., beautification, compression, etc.). In the following subsections, we show at least one example from each of these classes.



(a) Original mesh (b) Symmetrized mesh

Figure 2.7: Symmetrizing a scanned screwdriver model. The input mesh is shown on the left side, and the output mesh is on the right. Note that the output mesh is perfectly symmetric, has less noise (e.g., on the tip and at the junction with the handle), and retains sharp features (e.g., the ridge on the top of the handle).

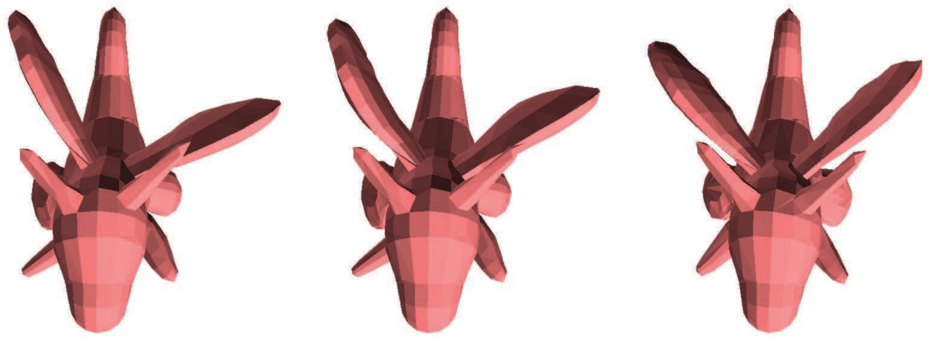
2.6.1 Beautification of Meshes for Symmetric Objects

There are many application domains in which scans are acquired for symmetric real-world objects. For example, in rapid prototyping applications, physical mock-ups are often constructed for a proposed design (e.g., with clay) and then scanned for computer simulation and processing. Likewise, in reverse engineering, objects are scanned when the original design is not available. However, rarely are the scanned models perfectly symmetric, due in part to scanner bias and noise, and due in part to processing tools that introduce asymmetries as a surface mesh is reconstructed. Since so many scanned objects are in fact symmetric, it seems useful to have a tool that takes a scanned mesh as input and produces the most similar symmetric mesh as output.

As an example, consider the scanned screwdriver downloaded from the Cyberware repository of Desktop 3D Scanner Samples (left side of Figure 2.7). In this case, the physical object has two parts (handle and tip), each of which is approximately symmetric with respect to two plane reflections (top-middle of Figure 2.7). Yet, the scanned mesh contains significant asymmetries with respect to all of these planes (e.g., artifacts at the junction of the tip and the handle).

Motivated by the idea of "beautifying" this mesh, we extracted planes of symmetry automatically with the Iterative Symmetric Points algorithm of [92], augmented to ensure that pairs of planes for the same part were perpendicular, and that all four planes aligned on a single axis (note that the planes for the handle are rotated by 20 degrees with respect to those of the tip). Then, we ran our geometric symmetrization algorithm on the entire mesh, with all four planes of symmetry guiding the surface deformation.

The result is shown in the images on the right side of Figure 2.7. Looking closely at the image in the middle right, it can be verified that the surface is symmetric up to the resolution of the mesh (note the high-frequency interleaved pattern of yellow, red, green, and blue overlaid surfaces). It can also be seen that significant shape features are retained during symmetrization (e.g., the ridge in the top of the handle), while noise is reduced (e.g., the tip shown in close-up on the bottom right). In general, shape features that align across multiple symmetries are retained, while those that do not are diminished. Overall, the mesh on the right of Figure 2.7 has the principal symmetries of the physical object and lower levels of noise, and thus is probably preferable for most simulation and visualization applications.



(a) Original (b) Symmetrized Part-way (c) Symmetrized Completely

Figure 2.8: Enhancing the symmetries of a sketched model under interactive control.

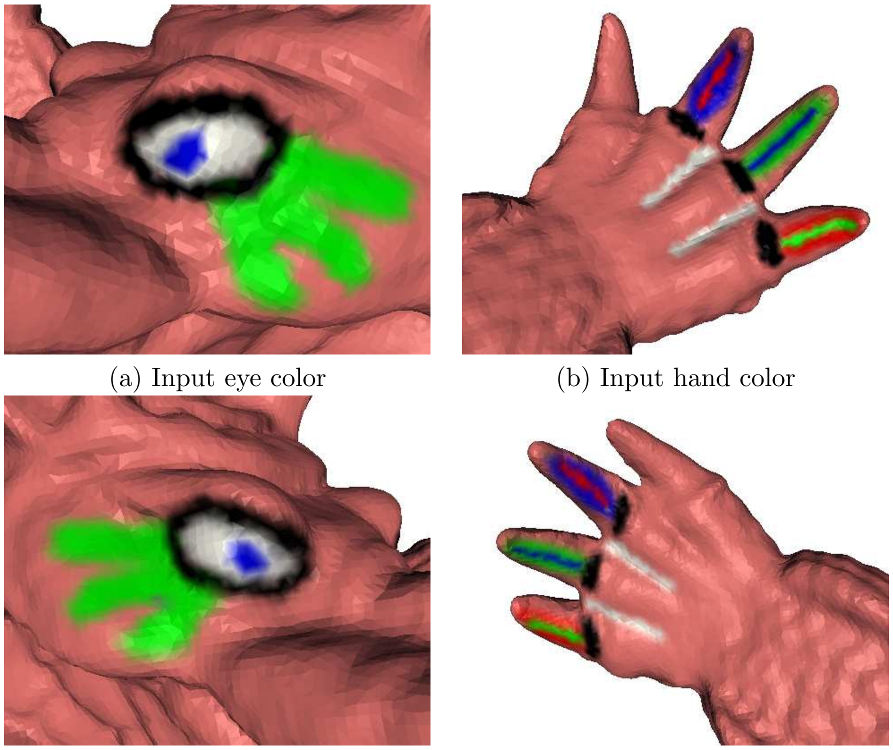
2.6.2 Symmetry Enhancement

In some applications, it may not be desirable to symmetrize a surface completely, but rather to enhance or to diminish symmetries instead. As a concrete example, imagine that a person has drawn the dragon shown in Figure 2.8a using a sketching tool like Teddy [47], but wants to make it more symmetric (note that the wings are quite misaligned with respect to the left-right symmetry plane). While this type of operation is possible with a series of deformations and local surface edits, it would be tedious with current modeling tools.

Instead, we propose an interactive tool that allows a user to control the degree of symmetrization applied to a surface. We provide a slider that the user can manipulate to make a surface more or less symmetric with respect to a selected transformation while the model is updated with real-time visual feedback. As an example, Figure 2.8b-c shows screenshots after the user has interactively symmetrized the dragon part-way (middle) and completely (right). In this simple case, the symmetrizing deformation could be computed in real-time. For more complex models, symmetry enhancement can be performed in real-time following symmetrization as a pre-process (see the video for examples). We believe that such a tool would be a useful addition to the suite of commands for interactive surface design.

2.6.3 Attribute Transfer

There are many applications that require blending or transferring attributes between semantically related surface regions – for example, texture transfer, denoising, and morphing. The challenge is usually to establish correspondences between semantically related parts. In the case of objects with approximate symmetries, symmetric mapping provides a useful way to solve this problem.


(a) Input eye color (b) Input hand color

(c) Transferred eye color (d) Transferred hand color

Figure 2.9: Transferring surface attributes between symmetric parts.

For example, consider the Armadillo model. Although the surface is “semantically symmetric” (e.g., the arm on the left has a functional correspondence with the one on the right), the surface is not symmetric geometrically (e.g., the arms are in significantly different poses). In cases like this, our symmetrization framework provides a natural way to establish correspondences between approximately symmetric parts via symmetric mapping.

This mapping can be used to transfer and blend surface attributes. For example, Figure 2.9 shows a demonstration of transferring per-vertex colors between symmetric regions of the Armadillo model. In the top row, the user has drawn colors on the eyes and hands on one side of the surface with an interactive painting interface. The system then automatically transfers the colors to the other side (bottom row) via an automatically generated symmetric map.

In this example, the main benefit is to save the user the effort of painting details twice. However, in other examples, perhaps it is important that the surface attributes are applied to both sides in exactly the same way, or that surface details are blended very precisely, which would be difficult without guidance from a symmetric map.

2.6.4 Simplification

Simplification algorithms take a mesh and produce an approximation with fewer polygons, usually to increase rendering speed, decrease storage, and/or provide a base domain for parameterization. Generally, however, they do not preserve large-scale symmetries (or other global shape features), in favor of minimizing local geometric errors.

In this section, we investigate whether the symmetry-preserving mesh decimation algorithm described in Section 2.5 can be used effectively for extreme simplification of approximately symmetric surfaces. Following the general approach outlined in Section 2.3, we establish symmetric topology for an asymmetric surface by first symmetrizing it, remeshing with symmetric topology, and then warping the new topology and correspondences back to the original geometry. We then perform symmetry-preserving mesh decimation on the symmetric topology over the asymmetric mesh.

Figure 2.10 shows the results of this method (first two columns) in comparison to the original version of *Qslim* (last column). Note that the topology of the mesh output by our algorithm is perfectly symmetric, even though the geometry of the surface is not. Note also that the geometric approximation achieved with symmetry-aware simplification is similar to the original (according to Metro [89], it has a Hausdorff distance approximately 6% larger). Since the symmetric mesh better reflects the semantic structure of the surface, we believe it may be preferable as a base domain for parameterization, animation, simulation, and other applications.

2.7 Conclusion

In summary, this paper has investigated methods for and applications of symmetrizing 3D surface meshes. The main idea is that symmetry-aware algorithms can be used to preserve, exploit, and enhance structural symmetries of a surface, even if the underlying geometry is only approximately symmetric. This idea is important because the vast majority of objects in the world have some sort of structural symmetries, and current mesh processing algorithms generally do not preserve them.

The main contribution of this paper is the symmetry-aware mesh processing framework, which includes algorithms for geometric symmetrization and symmetric remeshing. We provide demonstration of the framework for mesh beautification, symmetry enhancement, attribute transfer, and simplification.

The initial results seem promising, but our implementation has limitations, which suggest immediate topics for future work. We have demonstrated our algorithms only for symmetries across

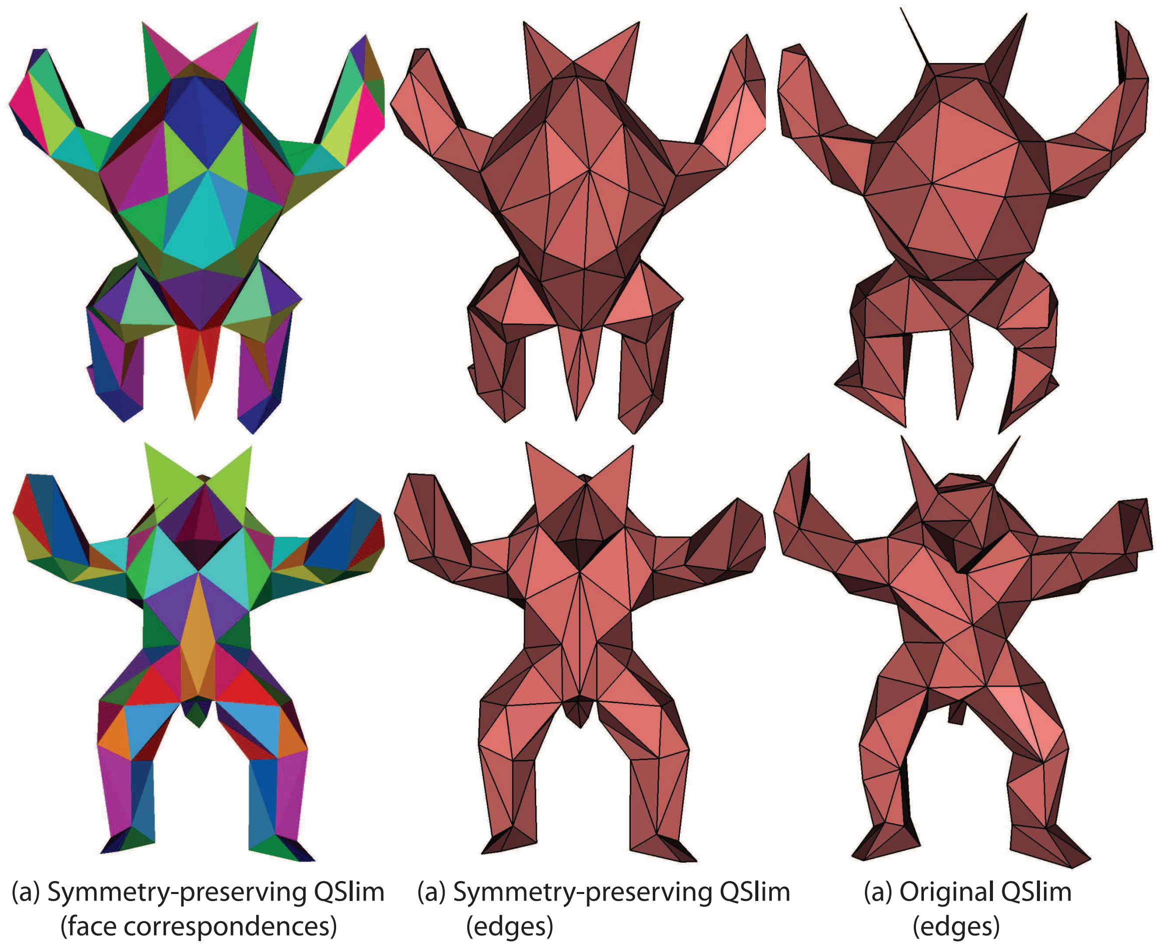


Figure 2.10: Symmetry-preserving QSlim (a-b) produces a surface approximation comparable to the original algorithm (c) of [35], but guaranteed to have symmetric topology, even for an asymmetric surface (the first column (a) shows symmetric face correspondences preserved during the decimation).

planar reflections. Although our code can handle symmetries for arbitrary affine transformations, we have not investigated examples of this type in our study.

Considering steps forward, the most obvious next step is to investigate other applications enabled by symmetry-aware processing. First candidates include compression and denoising. In the former case, it is possible that factoring a mesh into its symmetric part and its asymmetric residual could provide increased compression ratios, since at least half of the symmetric part can be discarded [107]. For denoising, the symmetric map could provide a way to blend noise across symmetric surfaces, as in Smoothing by Example [120]. These are just two examples – considering other applications that exploit symmetries will be a fruitful topic for future work.

The main long-term direction suggested by this work is that digital geometry processing algorithms can and should consider large-scale structural features as well as local surface properties when processing a mesh. So, future work should consider better ways to detect and encode large-scale shape features (such as symmetry) and to preserve and exploit them during surface processing.

Chapter 3

Part Decomposition of Objects

3.1 Introduction

The goal of part decomposition is to take a 3D model (usually represented as a mesh), and decompose it into functional or otherwise meaningful parts. This process is referred to in the computer graphics community as segmentation, and it is a central problem in computer graphics. A part decomposition reveals the global structure of an object. Segmentation is important for shape analysis and useful for a number of applications beyond analysis, including modeling, skeleton extraction, and texture mapping.

A great deal of research into segmentation has been conducted; a survey can be found in [2]. Most segmentation methods proceed by representing the mesh as a graph, and clustering the graph, and can be characterized by their choice of clustering error and minimization procedure. The algorithms described in recent research have produced segmentations of increasing quality. However, part decomposition is a difficult and under-specified problem, so even state-of-the-art segmentation techniques fail to consistently produce desired segmentations. While we describe novel methods of segmentation in this chapter, our focus here is not the general segmentation problem, but instead two more specific directions.

First, we investigate how to segment a set of meshes consistently, so that corresponding parts are grouped together. The motivation for consistent segmentation is two-fold. Many segmentation applications require part correspondences between objects, so it is useful to create such correspondences immediately. Also, segmenting a set of meshes simultaneously, rather then one at a time, leads to better segmentations, since the part decomposition of a particular mesh can be aided by cues from other meshes.

Second, we investigate a “continuous” version of segmentation. This investigation is motivated by the observation that segmentation is a difficult problem in part due to its discrete nature: a decision must be made about whether to cut a mesh at a proposed boundary, which inevitably leads to thresholds and incorrect cuts for some cases. Instead, we propose a continuous structure called a “partition” function, which is more robust because it avoids such discrete decisions. We show how to compute such a structure using randomized cuts, and demonstrate that although a partition function is not quite a part decomposition, it is more robust and can be useful for several applications in shape analysis.

The following section summarizes previous work in mesh segmentation. Section 3.3 describes our method for consistently segmenting a set of meshes. Section 3.4 described our shape analysis framework based on randomized cuts.

3.2 Related Work

In this section, we summarize previous work in general mesh segmentation. In the following sections, we review work more closely related to our investigations within mesh segmentation.

Many mesh segmentation methods exist in the graphics literature that aim to decompose a mesh into functional parts; surveys can be found in [103] and [2], and comparisons between several algorithms appear in [11]. These approaches aim to create segments that are well-formed according to some pre-defined low-level criteria: the segments are convex, boundaries lie along concavities, etc.. They use techniques such as K-means [106], graph cuts [55], hierarchical clustering [36, 38, 48], random walks [62], core extraction [54], tubular primitive extraction [84], spectral clustering [73], critical point analysis [70], convex decomposition [21], watershed analysis [76], region growing [130], mesh simplification [69], primitive fitting [10], Reeb graphs [9], and snakes [67].

As mentioned in the previous section, while the methods proposed above yield segmentations of increasing quality, the results from automatic algorithms remain inadequate. Our main goal, rather than improving on the general segmentation problem, investigating two particular scenarios related to segmentation: consistent segmentation of a set of models, and randomized cuts for a continuous version of segmentation. Our methods can be synergistic with many previous methods of segmentation in that: (a) many of these methods can be adapted to improve quality in the case of segmenting a set of meshes simultaneously by creating a similar algorithm to one described in Section 3.3, and (b) most of these methods can be randomized to create a continuous version of

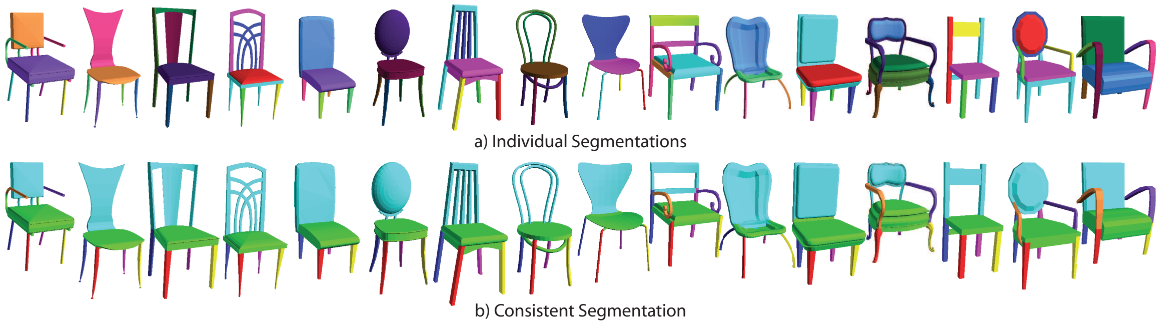


Figure 3.1: Individual segmentations of a set of chairs are shown in (a). Instead, our method creates a consistent segmentation of a set of meshes (b), that allows outlier segments, such as armrest segments in those chairs that have armrests. Consistent segmentations not only bring similar object parts into correspondence, but create better part decompositions for each mesh than individual segmentations do. Note that in many individual segmentation, the backs of chairs are either broken into several components, or aggregated with the rear legs.

segmentation using the procedure described in Section 3.4.

3.3 Consistent Segmentation

3.3.1 Motivation

The goal of this section is to develop a method that can produce a consistent segmentation of a set of meshes (Figure 3.1b). Such a segmentation is useful for a number of applications. Parts can be labeled and put into a knowledge ontology [97], they can be interchanged as part of a modeling tool [58], and they can be put into a searchable database [102]. In addition, we can expand lexical databases such as Wordnet [29] from having part-of relationships ("a seat is a part of a chair") to having possibly probabilistic spatial relationships ("some chairs have armrests, which are oblong, in front of backs, and above seats"). All these applications have at their heart the problem of this paper: consistently decomposing a set of 3D models into parts.

Many methods have been proposed to segment an individual mesh into parts. While these methods have produced segmentations of increasing quality, consistently segmenting a set of meshes remains challenging. Some mesh segmentation methods use heuristics designed to remain consistent between meshes (such as the Shape Diameter Function [104]). However, segmenting meshes individually ignores important cues available from processing a whole class of objects simultaneously. While several methods [102, 58] have been proposed to segment multiple objects, compared to our technique they have shortcomings, such as assuming that each mesh can be segmented individually, or that each mesh has the same number of segments.

Our approach extends the idea of single-mesh segmentations to a segmentation of multiple meshes: we simultaneously segment models and create correspondences between segments. Specifically, we first build a graph whose nodes represent faces of all the models in the set, and whose edges represent links between adjacent faces within a mesh, and between corresponding faces of different meshes. We then cluster the graph, creating a segmentation in which adjacent faces of the same model and corresponding faces between different models are encouraged to belong to the same segment.

Our approach has several advantages. First, segmenting a set of objects in a class simultaneously can produce not only more consistent results across the class, but also better individual segmentations than segmenting each object separately (as demonstrated in Figure 3.1). This is because a set of objects helps to identify salient segments that are shared across the set, and because those models that have more obvious segmentation cues help to segment more difficult models. Second, modeling tools are often used to create an object via a hierarchy that is then saved in VRML and other formats, and this hierarchy can be used as "prior" segmentation to give cues to the desired segments. Our approach combines these prior segmentations with other traditional segmentation cues, such as connected mesh components, concavities, and short boundaries between components. Third, by posing the problem as that of clustering a graph representing all the meshes, our method allows for outlier segments in the resulting segmentation, such as detection of armrests only on those chairs that have them. Finally, our method handles models with disconnected components, which is not the case for many other segmentation algorithms.

We demonstrate the effectiveness of our method on several object classes. We then suggest two new applications of our method: 1) creation of a symmetry-respecting segmentation of a single model, and 2) transfer of segmentations between a set of models of the same class.

3.3.2 Related Work

We review related work in two categories: segmentation of sets of objects, and semantic labeling of meshes.

Segmentation of Sets of Objects Several computer graphics papers have been written that involve segmentation of sets of objects. Part Analogies [102], for example, segments each model into parts, and then creates a distance measure between parts that takes into account both local shape signatures, and the context of the parts within a hierarchical decomposition. The main output of this is a catalog of parts with inter-part distances, which can then be used to create a consistent segmentation. We differ from this approach in that instead of first creating independent segmentations of the models and then finding correspondences between these segments, we create segments and correspondences between them simultaneously. Segmenting models and finding segment correspondences simultaneously improves segmentation quality because (i) a set of objects helps to identify salient segments that are shared across the set, and (ii) those models that have more obvious segmentation cues help to segment more difficult models.

Another related work is Shuffler [58], a modeling tool that allows the user to swap a segment in one mesh for a segment in another. Shuffler creates a mutually consistent segmentation between a pair of meshes, in which the segments are in one-to-one correspondence. We differ from their approach in two ways. First, while Shuffler's method could in principle be extended to segmenting a set of objects, their work demonstrates only pairwise segmentations. Second, we do not assume that all models contain all parts but instead allow outlier segments, making the problem more difficult, but allowing richer decompositions.

Some papers in the computer vision literature aim to obtain a segmentation of the same object or similar objects in different images: [68] creates an implicit shape model for segmenting a class of objects from examples, [98] uses a generative graphical model to improve segmentation by using two images rather than one, and [114] uses spectral clustering to find correspondences and matching segments. The last method constructs a Joint-Image Graph that encodes intra-image similarities as well as inter-image correspondence, which is similar to the graph we construct. Our method extends this idea to the 3D model domain, where the challenges lie not in disconnecting foreground objects from a cluttered background, but in using geometry to create precise segmentations.

Semantic Labeling Several papers address the problem of assigning semantic labels to meshes. ShapeAnnotator [97] describes an effort to put 3D models into a knowledge base, and consistently annotate their parts. They do not provide an automatic algorithm to do so, instead offering a tool that presents the output of several automatic segmentation algorithms such as those described above, and allows the user to select sections from each output to form segments and annotate them. Such efforts would benefit from a tool such as ours that creates a consistent segmentation of meshes automatically.

3.3.3 Method

Our algorithm takes as input a set of meshes, and produces as output a *consistent segmentation*, which assigns every face of every model to a segment, so that, for example, all mesh faces on seats

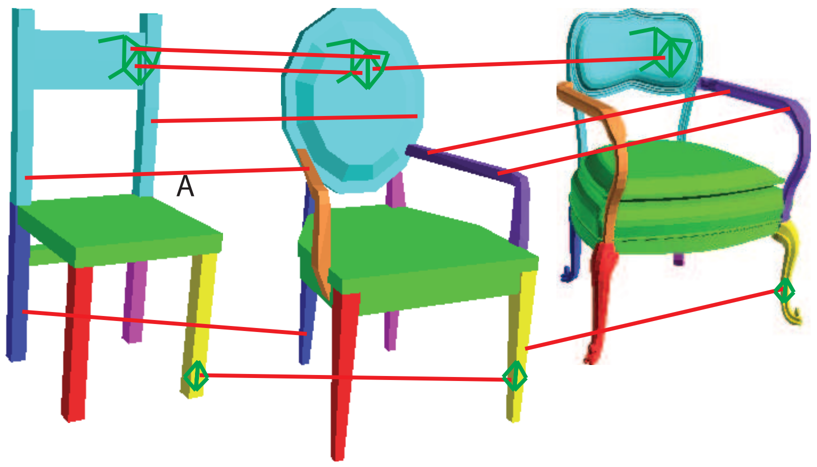


Figure 3.2: Schematic of our method. We create a graph whose nodes are mesh faces, and whose edges connect adjacent faces within a mesh (green lines) and corresponding faces in different meshes (red line). Some of the correspondences are incorrect (such as the one labeled 'A'). We find a clustering of this graph (with clusters represented by the colored rendering) that represents a consistent segmentation, allowing for outliers such as armrests.

of chairs belong to the same segment. This algorithm proceeds in two main steps, as follows.

First, we create a graph that contains as nodes the faces of the models. To this graph, we add two types of edges. Adjacency edges connect neighboring faces within a mesh, and are responsible for intra-mesh segmentation. Their weights use cues such as disconnected components and concavities to estimate whether they cross a segment boundary. The adjacency edges are similar to edges used in other segmentation algorithms and encourage short segmentation boundaries along concave edges (and, in our case, between disconnected components).

The other edges in our graph are correspondence edges. These edges are created between closest point pairs of globally aligned models. Ideally, correspondence edges link surfaces of corresponding parts of different models. These edges are responsible for inter-mesh segmentation; that is, they encourage corresponding parts of different meshes to aggregate into the same segment.

The resulting graph is sketched in Figure 3.2, with adjacency edges shown in green and correspondences edges in red. Note that the correspondence edges are not always correct: the correspondence edge labeled 'A' associates the armrest of one chair with the back of another.

Given this graph, we create a consistent segmentation by clustering its nodes into disjoint sets. This is done with a greedy hierarchical algorithm that seeks to minimize an error function similar to the normalized cut cost. The resulting clusters aggregate mesh parts such that each part (i) is weakly connected to the rest of the mesh and (ii) has more inter-mesh correspondences to the other parts in the cluster than to parts outside of the cluster. In Figure 3.2, these clusters are represented by mesh face colors.

Our approach has the following advantages. It balances the two smoothness constraints desirable in a consistent segmentation: adjacent faces should belong to the same segment, and corresponding faces from different meshes should belong to the same segment. Because these are not hard constraints, inevitable wrong correspondences can be corrected by strong adjacency cues, and poor adjacency cues can be corrected by consistent correspondences. For example, some of the faces of a chair's armrest may be initially erroneously found to correspond to faces of another chair's back (correspondence labeled 'A' in Figure 3.2). However, strong adjacency cues would prevent a small part of an armrest from being cut from the rest of the armrest and being associated with the "back" segment. Conversely, a chair may have no adjacency cues to suggest that its back should be separated from its rear legs (such as the first chair of Figure 3.2). But, consistent correspondences between the faces of this chair's and other chairs' rear legs would suggest that they belong in a separate segment. Finally, posing the consistent segmentation problem as a clustering problem allows for outlying segments: it permits identifying parts such as armrests only in those chairs that have them.

3.3.4 Implementation

The following two subsections describe in greater detail how we build the graph for our method, and how we segment the graph.

Graph Construction

The first step is to construct a graph, sketched in Figure 3.2, whose nodes represent mesh faces, and whose edges encode two kinds of weighted constraints: adjacency constraints, that suggest that adjacent faces belong in the same segment, and correspondence constraints, that suggest that faces from different meshes that are in correspondence belong in the same part.

Adjacency Edges Adjacency edges represent traditional segmentation cues of single-mesh segmentations: segment boundaries should be small, should lie along concavities, and should separate disconnected components. In addition, if the mesh contains a part hierarchy (as meshes created with modeling tools often do), the output should aim to respect this hierarchy.

Adjacency edges are formed between every pair of adjacent faces. Weights are chosen such that the cost of a graph cut corresponds to the perimeter of the enclosed segment, weighed by the concavity of the edges. Namely, if θ is the exterior dihedral angle of a mesh edge and l is its length, the weight of the corresponding graph edge is $\min((\theta/\pi)^\alpha l, l)$, with α chosen to be 10. Finding segments of small cut cost in this graph encourages segment boundaries to be small, and to lie along concavities.

These edges are sufficient to form a graph within connected components of the mesh, but 3D models are often composed of many disconnected components (in the examples in this paper, the number of disconnected components ranges from 1 to 70). More adjacency edges need to be added to encourage segment boundaries along disconnected components, while allowing disconnected components to join into the same segment when necessary. The details of these additional edges are described below for completeness, but the key idea is illustrated in Figure 3.3, where disconnected components are represented by rectangles, and the new adjacency edges by green lines. These edges are created such that those connected components that are closer to each other (Figure 3.3a), and that are relatively close along a larger surface area (Figure 3.3b), are more strongly connected in the graph.

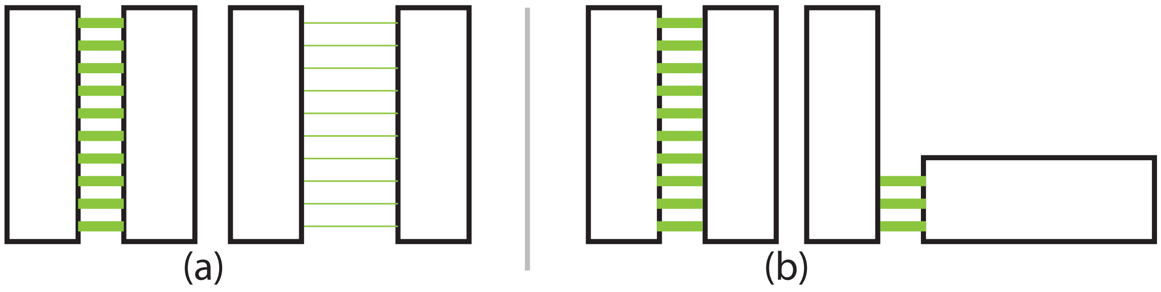


Figure 3.3: Disconnected components are drawn as rectangles, which we connect with (green) edges. We want components that are closer (a), and have more surface area in proximity (b) to be more strongly connected.

In greater detail, if D is the diameter of the bounding box of the mesh, we consider all pairs of disconnected components whose closest distance d is less than $.1D$. We would like to add edges to the graph (corresponding to the green lines in the schematic of Figure 3.3) whose lengths are close to d. We let $d_{eps} = .005D$ be a measure of a "small" length, and we consider $d' = 1.2(d + d_{eps})$. We sample points on both components, and for each point whose closest point on the other component is closer than d', we add an edge between the faces containing the points. The weight of that edge is a decreasing function of the distance times a constant; namely, we set it to $k_{cc}\overline{C}/(1 + d'')$, where k_{cc} is a constant that controls how important aggregating faces within a connected component is versus aggregating connected components, $\overline{C}$ is the average adjacency edge cost of the mesh, and d'' is the distance between the pair of points relative to a small mesh distance (that is, divided by $.005D$). For setting $k_c c$, no heuristic is foolproof, but we find it helpful to set k_{cc} low enough so that each connected component is fully aggregated before disconnected components are combined; after a few experiments, we set it to $k_{cc} = .01$.

Finally, we incorporate cues from a pre-existing part hierarchy, if one is available. We do this by considering the leaf nodes of the hierarchy as a “prior” segmentation, and encouraging segment boundaries along the boundaries of this “prior” segmentation. Specifically, we lower the cost of edges that cut across prior segmentation boundaries by a factor k_{prior} (we use $k_{prior} = 10$).

Correspondence Edges Correspondence edges link corresponding faces from different meshes using a set of weighted point correspondence pairs. While our algorithm accepts any scheme that creates weighted correspondences between surface points of different models, to avoid obscuring our main algorithm, we only use the simplest such scheme: closest points in the alignment that minimizes the Root Mean Square Distance between a pair of meshes within the space of similarity transforms. In particular, we align the pair of models with PCA, and starting with the 24 possible axis orientations, run the ICP algorithm, returning the alignment with the lowest RMSD. We find that this method rarely fails to find the best similarity transform relating a pair of models, and failure to find high quality correspondences instead usually indicates the need to use non-uniform scale or a non-linear alignment.

To form correspondence edges, we first align each mesh to every other mesh. We then sample the surface of the “from” mesh, and for each point, find the closest compatible point on the “to” mesh. Compatibility is determined by whether the dot product of the normals is greater than a threshold (we use .3). If this closest compatible point is sufficiently close (within 20% of the diameter of the bounding box of the model), we form a correspondence edge between the face node containing the “from” point, and the face node containing the “to” point. The result is that the cut cost of the correspondence edges in a clustering represents the surface area of each mesh of those points on the mesh that are in different segments from their corresponding points on other meshes.

Clustering

Given the graph constructed above, we need to define an error function and a procedure to minimize it that results in a segmentation of the graph, grouping together adjacent and corresponding faces of the input meshes. A number of segmentation schemes can be adapted to this problem; we use as our basis hierarchical clustering, in which each face starts in a separate segment, and segments are greedily merged in the order that minimizes a measure similar to a normalized cut cost of the graph.

To measure adjacency error E_{adj}, we use an area-weighted normalized cut cost: the sum of the adjacency edge cut cost of each segment normalized by the area of the segment. Because adjacency edges represent the length of a segment boundary modulated by concavity weights, this error encourages segments with small, concave boundaries with roughly equal areas. We normalize cut costs by areas, rather than total cost of edges in segment as in traditional Normalized Cuts [105], to avoid dependency on mesh tessellation.

For correspondence error E_{corr}, we use the normalized cut error of the correspondence edges, which is the sum of the correspondence edge cut costs of each segment normalized by the total cost of the correspondence edges attached to nodes in the segment. These errors represent quantities of different dimensionalities, so we form our segmentation error as the weighted product of the two:

$$E_{seg} = (E_{adj})^{\alpha} (E_{corr})^{1-\alpha}$$

where α controls the trade-off between merging segments within a mesh (towards $\alpha = 1$) and merging segments in close correspondence between different meshes (towards $\alpha = 0$).

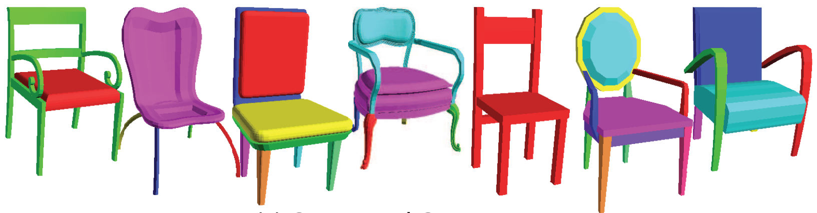
Our method of (approximately) minimizing the above error is hierarchical clustering: we start with each face in its own segment, and greedily merge segments in the order that yields the least error. However, there are two difficulties with this approach directly applied to the above error. The first is that the adjacency error encodes several cues (concavity, disconnected components, pre-existing segmentation) that vary widely from mesh to mesh and are difficult to normalize among meshes. This may result, for example, in one mesh having the same error with two segments as another mesh may have with twenty. The second problem is that merging in order of E_{seg} is relatively slow, since the contribution of each segment to the error is complicated, whereas merging in order of E_{adj} can be done relatively quickly with a priority queue since each potential segment merge affects the error as only a function of the two segments involved.

Due to these two reasons, we split our method into two steps. First, we oversegment each mesh independently to some preset number of segments, which is assumed to be an oversegmentation of the final result, by merging to minimize only E_{adj}. This initial oversegmentation is similar to the idea of superpixels [96] in computer vision. Note that while this stage is executed independently for every model, we do not try to segment the model to its final decomposition into natural parts; we merely aim for an oversegmentation, which is a much easier problem. Following this stage, we perform a slower segmentation of the entire graph representing all the models in the class by, at each iteration, considering all potential segment merges and computing the full E_{seg} for each (we use $\alpha = .01$ in this step). Note that while inter-mesh segmentation only begins at the second stage, intra-mesh segmentation continues.

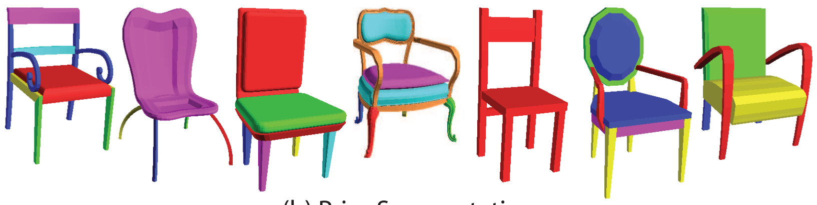
Finally, to compensate somewhat for the greedy nature of the algorithm and to allow the correction of a poor choice of segments to merge, we perform border refinement every k_{br} steps during the second segmentation stage (we use $k_{br} = 12$). In this step, we check over all graph nodes adjacent to each segment boundary, and expand the segment to contain an adjacent node if that expansion decreases the error. In all, two parameters are manually chosen for each consistent segmentation: the number of parts to oversegment each model to in the first phase, and the final number of segments in the consistent segmentation.

Running Time

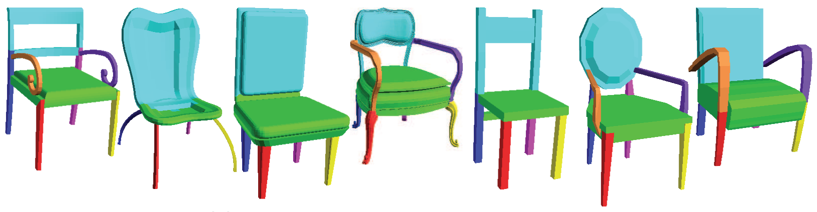
Our algorithm has three main time-intensive stages. First, the graph is constructed; in particular, adjacency edges between disconnected components and correspondence edges are created. If there are m meshes, with n vertices, and d disconnected components per mesh, with the aid of a kd-tree the time complexity is $O(md^2 n \log n)$ and $O(m^2 n \log n)$ for disconnected component and correspondence edges respectively. This stage took about 20 seconds for the chair example shown in Figure 3.1 on a 2GHz PC (the chairs class has 16 chairs, with a number of faces ranging from 168 to 9272, with an average 2296 faces per chair, and a number of connected components ranging from 1 to 16, with an average of 8). Second, each mesh is independently oversegmented; the running time for this oversegmentation is $O(mn \log n)$, and it took about a further 5 seconds for the chair example. For classes with many models, the most time-consuming stage is the final stage, in which we consider potential merges between segments of the oversegmented meshes at every iteration. The complexity of this stage is independent of mesh resolution. If the oversegmentation creates k segments per mesh, there are $O(mk)$ merges to perform, and a potential $O(m^2 k)$ merges to check at every iteration, so the complexity is $O(m^3 k^2)$. This stage takes about 3 minutes for the chair example (which was oversegmented to $k = 13$ segments). In all, the algorithm takes several minutes for the most complicated experiments in this paper. Many optimizations can be made to improve this. In particular, it is likely that not all pairs of meshes need to be considered for correspondence, and a constant number of corresponding meshes may be sufficient for each mesh.



(a) Connected Components



(b) Prior Segmentation



(c) Output Consistent Segmentation

Figure 3.4: For the right-most chairs of Figure 3.1, we show the connected components (a) and prior segmentation (b) that serve as cues for segmentation. These cues are helpful, but inconsistent between different models. We replicate the consistent segmentation results for these models in (c).

3.3.5 Results

In this section, we discuss some results of our method, and suggest several applications. We discuss three scenarios: the standard use of our algorithm to make consistent segmentations, the creation of a symmetric segmentation of a single mesh, and segmentation transfer between meshes of the same class.

Consistent Segmentations

The basic application of our method is to take as input a set of models and create a consistent segmentation. These models may be pulled automatically from the Internet, categorized automatically or semi-automatically, and then processed with our method to create a database of segments which may then be used in modeling or animation. We demonstrate the utility of our method for this application with models from the Viewpoint database, a commercial database containing household objects and furniture.

Figure 3.1b shows the results of segmenting a set of chairs. To give an example of the adjacency cues for this set, the connected components and prior segmentations are shown for several of the chairs in Figure 3.4. Note that while these are helpful cues, they are inconsistent between different meshes. There are instances where final part boundaries did not lie along boundaries in connected components or prior segmentation, and most connected components and prior segments do not correspond to the final parts. These cues had to be combined to form the segmentations: some chairs relied more heavily on connected components, others on prior segmentations, and yet others used cues from the underlying geometry, such as concavity (the back and seat of several chairs would have been one segment if only prior segmentation or connected components were used). Despite these challenges, our method was able to create a consistent segmentation, and also find the outliers: armrests were found in those chairs that had them, enriching the resulting segmentations. Finally, Figure 3.1a shows the chairs segmented individually to the same number of segments achieved in the consistent segmentation. This illustrates how segmenting the chairs in a group enhances individual segmentations: many of the cues to segment back legs from the backs of chairs come from correspondences to other chairs.

We show similar results for several other object classes in Figure 3.5. To the left of the dotted line, we show the consistent segmentations of several sets of objects: outlets, brooms, tables, and swivel chairs. To save space, we do not show the connected component and prior segmentation cues, but, similarly to the example in Figure 3.1, these cues are inconsistent between models, and the results used a combination of connected component, prior segmentation, and geometry cues. Most of these classes show examples where the possibility of outlier segments is helpful. The plugs and middle screw of outlets, the cylindrical elements around table tops and the legs of the four-legged table, and the armrests of swivel chairs would not have been identified if every object were required to have every part. Finally, to the right of the dotted lines are those models found immediately to the left of the dotted line segmented individually (to the same number of segments). These examples demonstrate how segmentations of sets of objects enhance the individual segmentations. For tables, for example, comparison to other tables makes clearer that the leg stands are a more important segment than a frequently disconnected component joining the stand to the top.

Symmetric Segmentations

It is desirable that the segmentations of symmetric or nearly symmetric objects be symmetric. However, most segmentation methods do not not produce symmetric results when run on nearly symmetric models. Figure 3.6a, for example, shows the results of the hierarchical segmentation algorithm we use in our clustering phase on two nearly symmetric models. Such segmentations of nearly symmetric objects are often not symmetric for two reasons. First, segments are treated

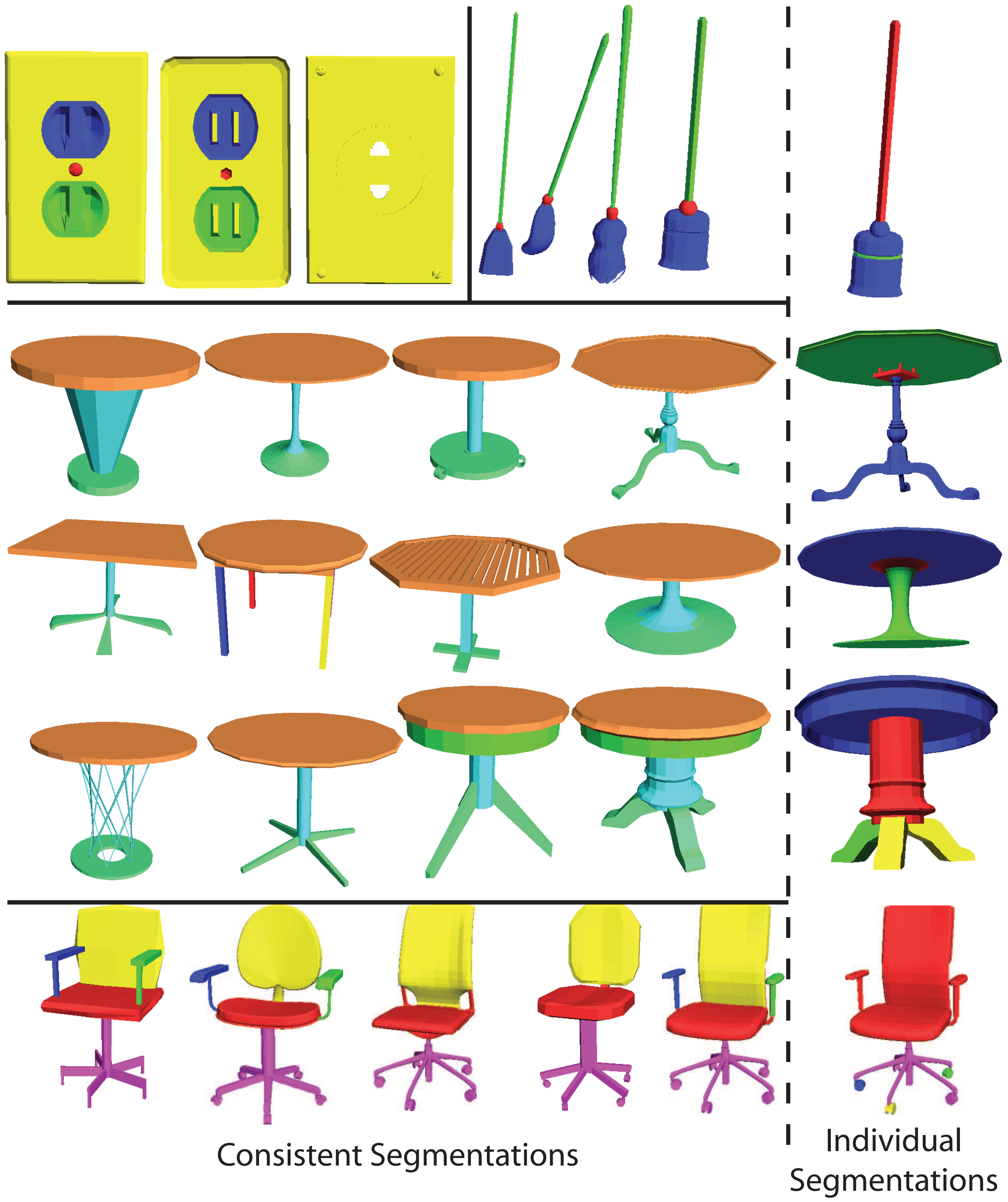


Figure 3.5: Results of consistent segmentations of several classes are shown to the left of the dotted line. Note that in many cases, the ability of our method to find outlier segments is helpful. To the right of the dotted lines are individual segmentations of the last models in the row. These individual segmentations fail to identify important segments that can be identified by comparing to other models in the class.

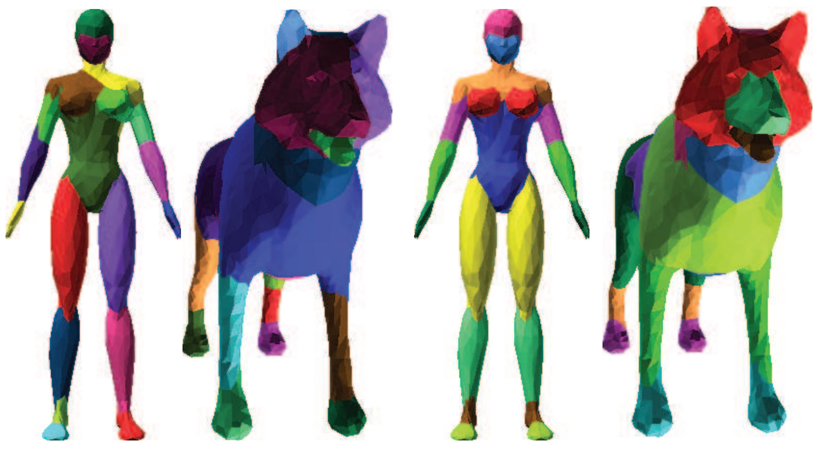


Figure 3.6: Using intra-model constraints to create a symmetric segmentation (b) of models that would have otherwise been segmented asymmetrically (a).

one at a time in many algorithms, and an asymmetry forms when, for example, one human arm is processed before the other–the segmentation errors become different after the merge and the algorithm may proceed differently when it gets to the other arm. Second, slight asymmetries in triangulation and geometry result in asymmetries in energies. To overcome such deficiencies, we take a "symmetry-aware" processing approach as proposed in Chapter 2.

Our method places intra-model correspondence constraints between symmetrically corresponding points instead of between points on different meshes. To get these constraints, the method in Chapter 2, which finds surface mappings between corresponding parts of nearly symmetric objects. For the two examples shown here, the symmetry is so close to perfect that it is sufficient to create closest-point correspondences to the reflection of the model. In Figure 3.6b, we show segmentations of the models created with our method, with constraints added from a global reflective symmetry. Note that not only does this result in correspondences between symmetric segments, but also produces better segmentations: the asymmetric cuts above the chest of the human model, and across the face of the dog are eliminated. While we demonstrate results for the simple case of one strong global symmetry, since our method takes as input any weighted point correspondences, it can also be used for multiple, local, and partial symmetries.

Segmentation Transfer

In many scenarios, it may be helpful to start with example segmentations of objects and transfer the segmentations to previously unsegmented meshes. These example segmentations may come in the form of labels from a user interface, previously existing examples, or the result of other algorithms

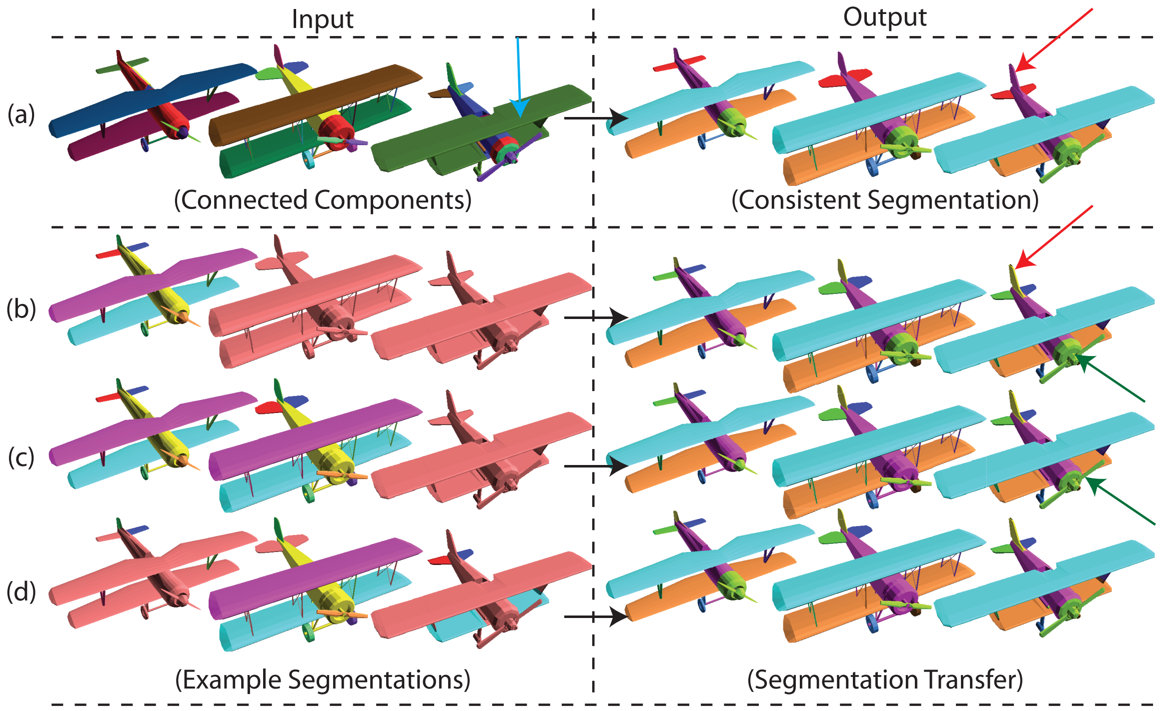


Figure 3.7: Our method can transfer segmentations from example meshes. Consider the three biplanes, whose connected components are shown on the left side of (a). We can run our method, as usual, to create a consistent segmentation (right column of (a)). This segmentation may not match the desired one; for example, it may be desirable to separate the three stabilizers in the back of the plane (red arrow). We can use the first plane as an example to segment the others (b). This segments the stabilizers correctly, but may miss other desired details. For example, the propeller segment may be too large (green arrow). In (c), we use the example segmentations of the first two planes to segment the third. Compared to (b), this creates more precise restrictions, and creates a smaller propeller segment. Our method can also handle constraints in which segments are partially given, and the remainder is required to be filled in. In (c), we randomly choose half of the segments to be known, and use these as examples to generate the segmentation in the right column.

that may be tuned to find specific object parts. Here, we explore three such scenarios: transferring segmentations from a single example to a set, using a set of segmented examples to segment a new model, and taking advantage of partially segmented data to complete the segmentations. Our algorithm can handle any mixture of these constraints.

Consider the three biplanes in Figure 3.7. They have no prior segmentation, and their connected components are shown in Figure 3.7a on the left. Note that here, connected components mostly represent an oversegmentation; however, for example, in the third plane, the top and bottom wings as well as their vertical supports are all one connected component, where we would like to create four segments (blue arrow). We can run our method, as usual, to create a consistent segmentation (right column of (a)). This segmentation may not match the desired one; for example, it may be better for the horizontal tail stabilizers to be represented as two segments, and to separate the vertical tail stabilizer from the body (red arrow). Below, we describe three scenarios in which additional constraints (in the form of "known" segments) may be used to improve the situation and to transfer segmentations from examples. The only change needed to our method to reflect these constraints is to start with the faces merged into those segments that are "known", and to prevent two "known" segments from merging.

In the first scenario, we can use a segmented example mesh to label a set of other meshes in the class. This segmented example may be generated in real time with an interface, or it may come from an existing repository or other external sources. In our example, such a scenario is represented in Figure 3.7b on the left. The first plane is segmented, and used as a template for the other two planes. The result is shown on the right of Figure 3.7b. Compared to the baseline consistent segmentation result, the horizontal tail stabilizers are recognized as two segments, and the vertical stabilizer is separated from the body.

In the second scenario, there exists a fully segmented set of models belonging to some class. We would like to augment this set with a new, unsegmented model. This scenario is represented in Figure 3.7c on the left: the first two planes are segmented, and the third is not. Using these two models, we segment the third plane; the segmentation is shown on the right of Figure 3.7c. Compared to the baseline consistent segmentation case, the stabilizers are segmented correctly following the examples. Compared to using one segmented example, since more precise correspondences are available, the propeller segment is smaller and better resembles the example propeller segments, although it also includes a small part of the plane that is not the propeller (green arrow).

In the third scenario, the input is a partially segmented set of meshes. Such a partially labeled set can come from several sources. One possibility is to use the labels created by users during modeling, and attached to object parts. While these labels are not consistent, it is conceive able that parsing them and using a lexical database such as Wordnet [29] will create proper associations. Alternatively, one may use automatic detectors that are trained to find specific object parts. Note that in this scenario, these partial segmentations offer much more information than the "prior" segmentation shown, for example, in Figure 3.4b: for this application, we assume that those segment boundaries that are provided are correct, and that segments of the same type provided in different meshes are associated with each other. We simulate this situation by manually segmenting the biplanes of Figure 3.7, and randomly removing half of the segments. The input is shown on the left in Figure 3.7d, and the output shown on the right. Note that, again, compared to the unconstrained segmentation, the rear stabilizers are found as desired.

3.3.6 Conclusion

We present a method that takes as input a set of meshes, and produces a consistent segmentation from several inconsistent cues. The method is demonstrated on several examples, and two more specific applications are suggested: symmetric segmentations of a single mesh and segmentation transfer between meshes belonging to the same class.

As this is a prototype system, there are several limitations. First, our method uses parameters to balance the relative importance of various segmentation cues. These parameters could, in principle, have been learned from example segmentations. Instead, we set them experimentally to favor aggregating adjacent, convex faces, followed by adjacent, concave faces, then close disconnected segments, and then distant disconnected segments, with faces within "prior" segments aggregating before faces across prior segmentation boundaries. We did not fine-tune these parameter values - most of them are our initial choices, so the values are reasonable, but by no means optimal.

Second, as implemented, our method is limited to those cases in which satisfactory correspondences may be created through a global similarity alignment. These correspondences do not need to be perfect (in none of our examples were the correspondences perfect), since segmentation cues can compensate for errors in correspondences. However, better alignments produce more consistent segmentations. Figure 3.8 shows an example of how a poor alignment can hamper the resulting segmentation. Note that because of the poor alignment, the heads of the animals are in separate segments, some of the hoofs are outlying segments, and the back of the horse is merged with the tail of the giraffe. To improve the results of such cases, it may help to consider non-rigid alignments, or part-based correspondence methods.

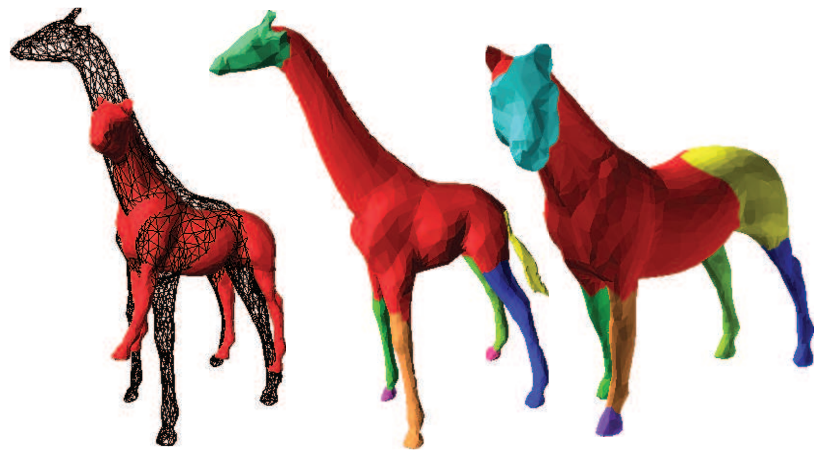


Figure 3.8: An example of how a poor alignment (a) hampers the resulting consistent segmentation (b). Note that due to the poor alignment, the heads of the animals are not found to correspond, the hoofs are sometimes labeled as outlying segments, and the back of the horse is found to correspond to the tail of the giraffe.

Third, we consider only low-level cues: adjacency and point correspondences. Better segmentations may be achieved by expanding the alignment error function to include shape signatures and contextual cues. The heuristics used in Part Analogies [102] and Shuffler [58], for example, include such terms.

Finally, our method takes as manual input the number of desired segments. Ideally, this number would be determined automatically: perhaps studying the behavior of our error function as segments are merged will suggest heuristics to automate the final number of clusters (e.g., as in [55]).

3.4 Randomized Cuts

3.4.1 Motivation

Shape analysis of 3D surfaces is a classical problem in computer graphics. The main goals are to compute geometric properties of surfaces and to produce new representations from which important features can be inferred. In this section, we investigate a new shape analysis method based on *randomized cuts* of 3D surface meshes. The basic idea is to characterize how and where a surface mesh is most likely to be cut by a segmentation into parts.

Our approach is to compute multiple randomized cuts, each of which partitions faces of the mesh into functional parts (Figure 3.9b). From a set of random cuts, we derive: 1) a *partition function* that indicates how likely each edge is to lie on a random cut (Figure 3.9c) and 2) a set of the most

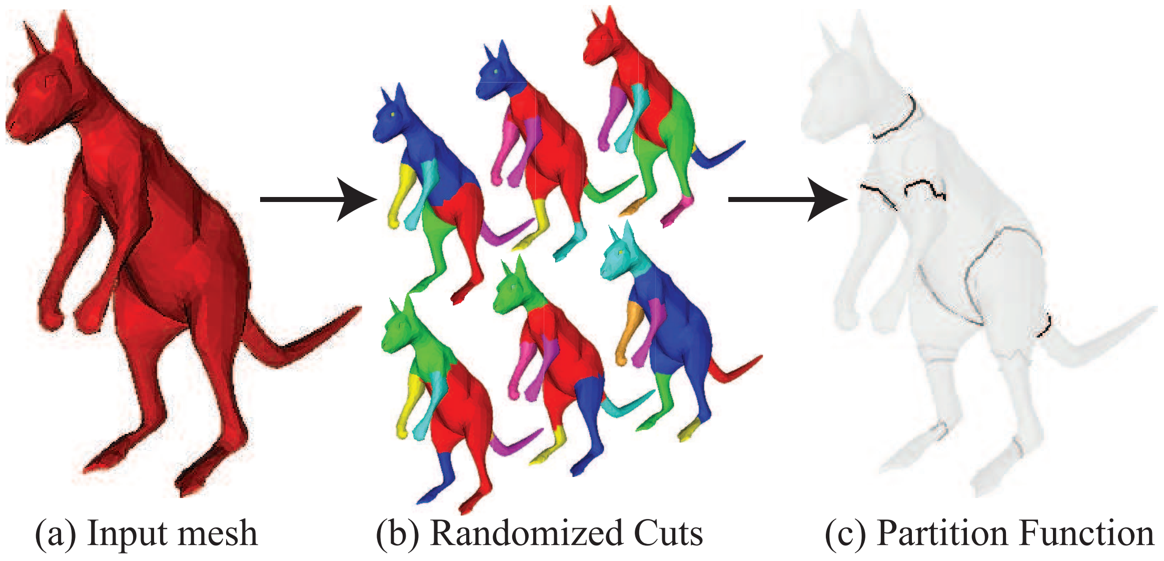


Figure 3.9: Shape analysis with randomized cuts. Given a 3D mesh as input(a), we sample segmentations from a distribution (b) to generate a *partition function* that estimates the likelihood that an edge lies on a segmentation boundary. This function provides global shape information that is useful for mesh visualization, analysis, and processing applications.

consistent cuts.

These structures provide global shape information useful for visualization, analysis, and processing of 3D surface meshes. For example, the partition function provides a continuous function for visualization of "chokepoints" (Section 3.4.7), deformation at joints (Section 3.4.7), and selection of stable features for surface registration (Section 3.4.7). The most consistent cuts are useful for finding part boundaries in surface segmentation (Section 3.4.7).

In this section, we make the following research contributions. First, we introduce two new structures for characterization of 3D surface meshes: the *partition function* and the *consistent cuts*. Second, we describe three alternative methods for computing them, based on randomized K-means, hierarchical clustering, and minimum cut algorithms. Third, we provide an empirical analysis of the properties of the partition function, focusing on sensitivity to surface noise, tessellation, pose, and shape variations within an object class. Finally, we demonstrate applications of randomized cuts in mesh visualization, registration, deformation, and segmentation.

3.4.2 Related Work

Here we summarize related work in two categories closely related to our investigation: random cuts, and shape analysis.

Random Cuts Our approach follows a vast amount of prior work on randomized algorithms for graph partitioning in theory and other subfields of computer science. Perhaps the most relevant of this work is the algorithm by Karger and Stein [52], which utilizes a randomized strategy to find the minimum cut of a graph. The main idea is to generate a large set of randomized cuts using iterative edge contractions, and then to find the minimum cost cut amongst the generated set. Our paper differs in that we use randomized cuts to produce a partition function and to generate a set of scored cuts (rather than finding a single minimum cut).

Other authors have used randomized algorithms for graph analysis. For instance, [37] generated multiple cuts in a graph representing an image with randomized iterations of hierarchical clustering to produce a similarity function indicating the probability that two nodes reside in the same cluster. The probabilities were used to segment the image with "typical" cuts, by constructing connected components of the graph formed by leaving edges of probability greater than 0.5. The result is both a pairwise similarity function for pixels and an image segmentation. Our work takes a similar approach in that we use randomized clustering to explore the likelihood that nodes lie in the same cluster (more precisely, that edges lie on the boundaries between clusters). However, we investigate a several randomized clustering strategies (finding that the hierarchical algorithm of [37] does not perform best in our domain), produce a partition function and a set of consistent cuts for 3D meshes, and investigate applications for 3D mesh visualization, analysis, and processing.

Shape Analysis Our goals are motivated by recent work in computer graphics on defining geometric properties for analysis, visualization, and processing of 3D surface meshes. Much of this research has been summarized in Section 1.2; some of it is repeated here to place it in the context of randomized cuts.

Some classic examples of local shape properties include curvature, slippage [38], accessibility [80], saliency [66], multiscale analysis [85], and electrical charge distribution [118]. Recent examples of global shape properties include shape diameter [104], ambient occlusion [129], spectral analysis [128], and average geodesic distance [46]. These properties have been used for visualization [129], segmentation [104], skeletonization [55], matching [46], and several other applications. Our partition function adds to this list: it is a global shape property that measures the probability that a surface point lies on a segmentation boundary.

3.4.3 Overview of Approach

We investigate the use of random cuts for shape analysis of 3D meshes. The general strategy is to randomize mesh segmentation algorithms to produce a function that captures the probability that an edge lies on a segmentation boundary (a cut) and to produce a ranked set of the most consistent cuts based on how much cuts overlap with others in a randomized set.

This strategy is motivated by three factors. First, no single algorithm is able to partition every mesh into meaningful parts every time [11]. Second, existing segmentation algorithms often have parameters, and different settings for those parameters can significantly alter the set of cuts produced. Third, segmentations produced by different algorithms and parameter settings often cut many of the same edges [12]. Thus, it could be useful to combine the information provided by multiple discrete segmentations in a probabilistic framework, where the output is not a single discrete segmentation, but rather a continuous function that captures the likelihood characteristics of many possible segmentations.

For instance, consider the example shown in Figure 3.9. The images in Figure 3.9b show discrete segmentations of a kangaroo produced by a randomized hierarchical clustering algorithm (as described in Section 3.4.5). Although every segmentation is different, some boundaries are found consistently (e.g., along the neck, cutting off the legs, arms, tail, etc.). These consistencies are revealed in the partition function in Figure 3.9c, which shows for each edge the probability that it lies on a segment boundary. We believe that this continuous function reveals more information about the part structure of the kangaroo than does any one of the discrete segmentations.

More formally, we define the *partition function* $P(e, S)$ for edge e of a 3D mesh with respect to a distribution of segmentations S to be the probability that e lies on the boundary of a random segmentation drawn from S. From this partition function, we define the *consistency*, $P(S_i)$, of a cut, S_i, within a distribution S as the length-weighted average of the partition function values of its edges, and we define the *most consistent cuts* as the set of cuts with highest consistency.

This formulation is similar to the one used to find typical cuts for image segmentation [37], but generalizes it in five main ways. First, the construction combines multiple segmentation algorithms (not just hierarchical clustering) and it allows randomization of any variable guiding those algorithms (not just which edge to contract). Second, it produces a partition function on edges (rather than a similarity function on pairs of nodes). Third, it produces a continuous score for every random cut (not just a discrete set of cuts whose edges all have probability greater than 0.5). Fourth, it provides a natural way to measure the consistency of cuts. Finally, it suggests a number of applications that utilize continuous functions on meshes (not just segmentation). Our main contributions are investigating the space of randomization strategies and mesh processing applications within this general formulation.

3.4.4 Process

The input to our processing pipeline is a 3D surface mesh, M (Figure 3.9a), and the outputs are: 1) a randomized sampling of segmentations, $\hat{S}$ (Figure 3.9b), 2) a partition function estimate, $P(e, \hat{S})$, for every edge e of the mesh with respect to $\hat{S}$ (Figure 3.9c), 3) a consistency score for every segmentation indicating how consistent its cuts are with others in $\hat{S}$, and 4) a ranked set of most consistent cuts. The processing pipeline proceeds in four main steps:

1. Graph construction: The first step produces an edge-weighted graph G representing the input mesh. We follow a standard approach to this problem, building the dual graph of the input mesh, where nodes of the graph correspond to faces of the mesh, and arcs of the graph correspond to edges between adjacent faces in the mesh. Because some algorithms seek to minimize traversal distances across a graph, while others minimize graph cuts, we associate two weights to each graph arc: a traversal cost, and a cut cost. The weights are created such that low-cost cuts in the graph correspond to favorable segmentation boundaries in the mesh, and low-cost traversal paths in the graph occur between points on the mesh likely to be in the same functional parts. Similarly to [33], we assign a concavity weight to each mesh edge as follows: if θ is the exterior dihedral angle across an edge, we define a concave weight $w(\theta) = \min((\theta/\pi)^\alpha, 1)$, which is low for concave edges, and 1 for convex ones (we use $\alpha = 10$ in all experiments and examples in this paper). We form cut costs by multiplying the mesh edge length by $w(\theta)$, and traversal costs by dividing the distance between face centroids by $w(\theta)$. The cut cost of a mesh segment then is its perimeter, weighted to encourage concave boundaries, and the traversal costs represent geodesic distances on a mesh, weighted to make travel through concave edges longer.

2. Random cuts: The second step generates a randomized set of K-way segmentations, where each segmentation partitions faces of the mesh into disjoint parts. For this step, we investigated several algorithms (including K-means, Hierarchical clustering, and Mincuts), and we randomized them with several different strategies. This step is the main focus of our study, and thus we defer details to Section 3.4.5. For the purpose of this section, it suffices to say that the output of this stage is a scored set of segmentations, where each segmentation S_i provides a set of edges C that are "cut" by the segmentation (lie on the boundary between segments), and the score indicates the "quality" of the segmentation (e.g., the normalized cut value).

3. Partition function: The third step estimates the partition function for each edge. Given a set of K-way segmentations, $\hat{S}$, randomly sampled from S, this can be done trivially for each edge e by summing the weights of all segmentations in $\hat{S}$ that contain a boundary on e and dividing the sum by the total weight of all segmentations in $\hat{S}$. If all segmentations have equal weight, then the result is an estimate of the probability that an edge is cut by a segmentation in S.

4. Cut consistency: The final step provides a consistency score for every segmentation S_i in $\hat{S}$. It is computed by taking a weighted average of the partition function values along edges on the boundaries of segments in S_i (where averaging weights are proportional to edge lengths). The M segmentations with highest scores (possibly all of them) form a set that we call the *most consistent cuts*.

The main focus of our project is on methods for generating random segmentations, analyzing properties of the resulting partition functions, and demonstrating applications in computer graphics. The following sections address these three issues, respectively.

3.4.5 Methods

The key step of our process is to define a randomized process that creates a distribution of mesh segmentations and to sample that process to compute properties of the distribution (the partition function). Several strategies are possible. First, random variables could determine which algorithmic strategy should be used to generate segmentations (e.g., K-means vs. hierarchical clustering). Second, they could determine parameters for specific algorithms (e.g., the number of segments in K-means). Third, they could guide choices made within a segmentation algorithm (e.g., the order of hierarchical edge contractions [52]). Finally, they could provide randomized variation to the input graph (e.g., randomized edge weights). Of course, combinations are possible as well. Each combination of random variables produces a distribution of cuts, and therefore a different partition function and set of most consistent cuts. In this section, we investigate a range of strategies for randomizing cuts and describe details of our implementations.

K-Means

K-means is a popular algorithm for clustering, and its variants have been used for graph partitioning and mesh segmentation. For example, [106] used K-means to decompose faces of a mesh into parts as follows. First, K seed faces were chosen to represent segment centers with a deterministic greedy

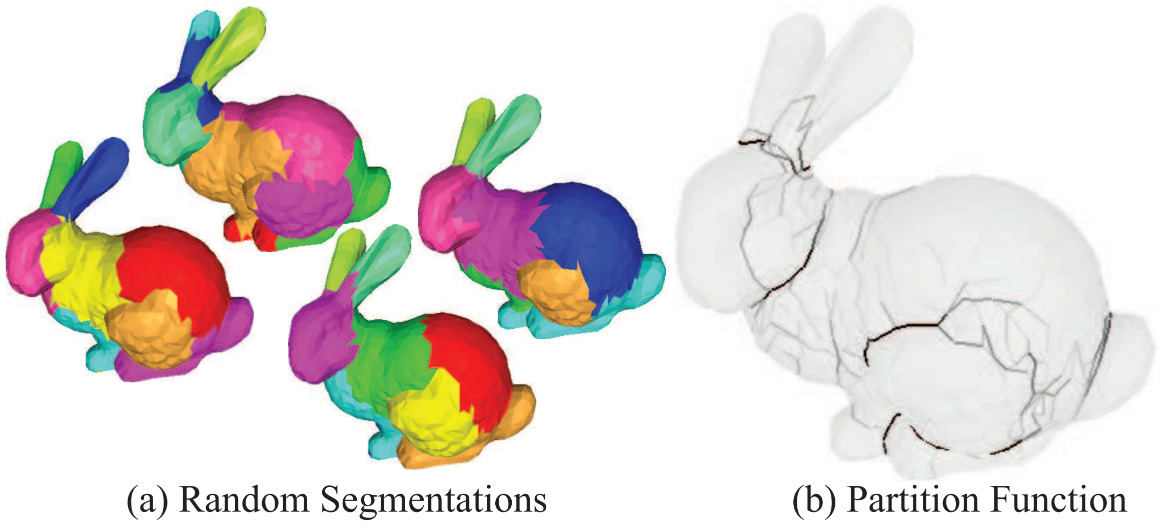


Figure 3.10: Sample randomized segmentations produced with K-means (a) and the resulting partition function (b).

process that maximizes their pairwise distances. Then, the algorithm alternated between assigning faces of the mesh to the segment represented by the closest seed face (according to traversal costs) and recomputing the seed faces to minimize the sum of distances to faces in their segments.[1] This process was iterated until convergence.

We have experimented with this approach within our framework by randomizing the set of seed faces chosen during the first step of the algorithm. That is, for each new segmentation sampled from the random distribution, we simply select K faces randomly from the mesh and then iterate K-means to convergence. After many randomized segmentations, we generate the partition function value for each edge by computing the fraction of segmentations for which that edge lies on a segment boundary.

Figure 3.10 shows an example partition function produced for the Stanford Bunny with this randomized K-means algorithm. The left image shows 4 of 1200 segmentations generated with K-means initialized with random sets of 10 seeds. The right image shows the resulting partition function. Note the dark lines indicating strong consistency in the cuts along the neck and front of the legs, while the cuts at the base of the ears and back of the legs do not always agree, and therefore appear lighter.

Since the value of K given as input to the K-means algorithm dictates the number (and therefore size) of parts generated in each segmentation, there is a family of possible partition functions that could be produced by this algorithm. For simplicity, we provide results for only one setting, $K =$

[1]In our implementation, seed faces are updated to be the ones furthest from their segment boundaries, a slight change from [106] that we found gave better results for our graphs.

10, as it provides a good trade-off between parts that are too big and ones that are too small. Alternatively, it would be possible to randomize K for different segmentations. However, we expect that approach would provide worse results (it would add poor segmentations to the distribution).

Hierarchical Clustering

Hierarchical clustering is another popular segmentation algorithm. In the context of surface meshes, the process starts with every face in a separate segment, and then merges segments iteratively in order of a cost function until a target number of segments (K) has been reached or some other termination criterion is met. Typical cost functions include the minimum cut cost of any edge between two segments (single-link clustering), the total cost of the cut between two segments [37, 52], and the Normalized Cut cost [105].

In our work, we begin with the hierarchical segmentation framework introduced in Section 3.3.3, in which segments are iteratively merged to minimize an area-normalized cut cost. To randomize this deterministic algorithm, rather than merging the pair of segments that minimizes the cost at each iteration, we set the probability of selecting a segment pair to merge to be a function of the cost. Specifically, to select a pair of segments to merge at each iteration, we map the differences in area-normalized cut associated with contracting each graph arc to [0, 1], raise each value to the power of $(1/r)$ where r is a randomization parameter, and choose an arc to contract with that probability. This way, as r approaches 0, this algorithm approaches its deterministic version. We found that values on the order of $r = .02$ produce a reasonable tradeoff between quality and sufficient randomness in the resulting segmentations. As in K-means, the desired number of segments is an input and controls the size and scale of the parts.

Figure 3.11 shows the partition function produced by running the randomized hierarchical clustering algorithm while greedily minimizing the area-weighted Normalized Cut cost to produce $K = 10$ segments. As in Figure 3.10, the left image shows four example segmentations, while the right image shows the resulting partition function. Note that boundaries at the top of the legs and front of the tail are sharper than they are for K-means (in Figure 3.10). Since the hierarchical algorithm based on area-weighted normalized cuts explicitly favors segmentations with boundaries on concave edges, it is more likely to produce consistent cuts in those areas. Note also that that K does not have to match the "natural" number of segments to produce a quality partition function: the randomized nature of the algorithm alternates between cuts with similar costs, such as cutting off one leg versus another, and therefore both are reflected in the partition function.

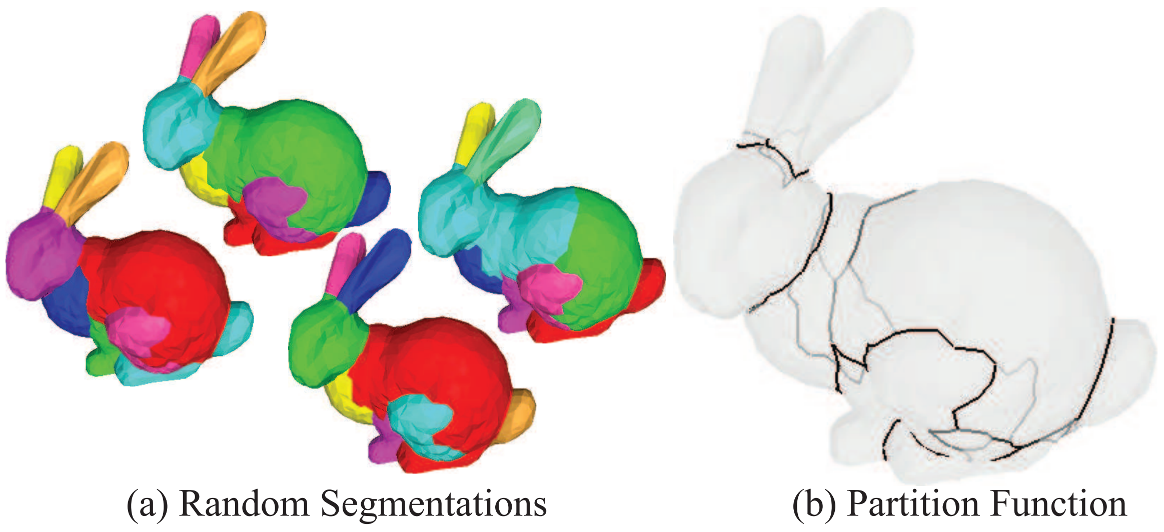


Figure 3.11: Sample randomized segmentations produced with Hierarchical Clustering (a) and the resulting partition function (b).

Min Cuts

Finding minimum cuts is a third possible segmentation strategy. The general approach is to select a set of K seed nodes and then to find the minimum cost cut (mincut) that partitions the seeds. This is a classical network flow problem with well-known polynomial time solutions for $K = 2$, and approximation algorithms for $K > 2$. However, in order to avoid trivial cuts that partition one face from the rest of the mesh and similar small-area segments, the mincut problem must be modified with constraints and/or penalties to favor nearly equal-area segments. Towards this end, [55] constrained cuts to lie within a "fuzzy" area, where the relative geodesic distance, $d = min(dist(e, A), dist(e, B))/(dist(e, A) + (dist(e, B))$ between an edge e and seeds, A and B, was greater than $0.5 - \epsilon$. While this approach avoids the main problem of making trivially small cuts, it can produce cuts that lie on the boundaries of the fuzzy area (e.g., both examples in Figure 3 of [55]), which may do not lie on natural seams of the mesh.

To address this problem, we make a small modification to the minimum cut algorithm of [55]. For every set of seeds, X, we adjust the weights on edges of the graph with a penalty function, $W(e, X)$, that gradually falls off with distance from the closest seed. In our current implementation, $W(e, X)$ is a piecewise linear function that has a constant high value of W_{high} for $d <= x_1$, decreases linearly to 1 at $d = x_2$, and then remains 1 for $d >= x_1$. This approach encourages cuts at larger distances from seeds, but does not enforce hard boundaries on the fuzzy area (within our framework, the method of [55] is equivalent to setting $x_1 = x_2 = .5 - \epsilon$). For simplicity sake, we control the coefficients and cutoffs for the distance penalty function with a single parameter, s ($W_{high} = 400s + 1$, $x_1 = s$, and

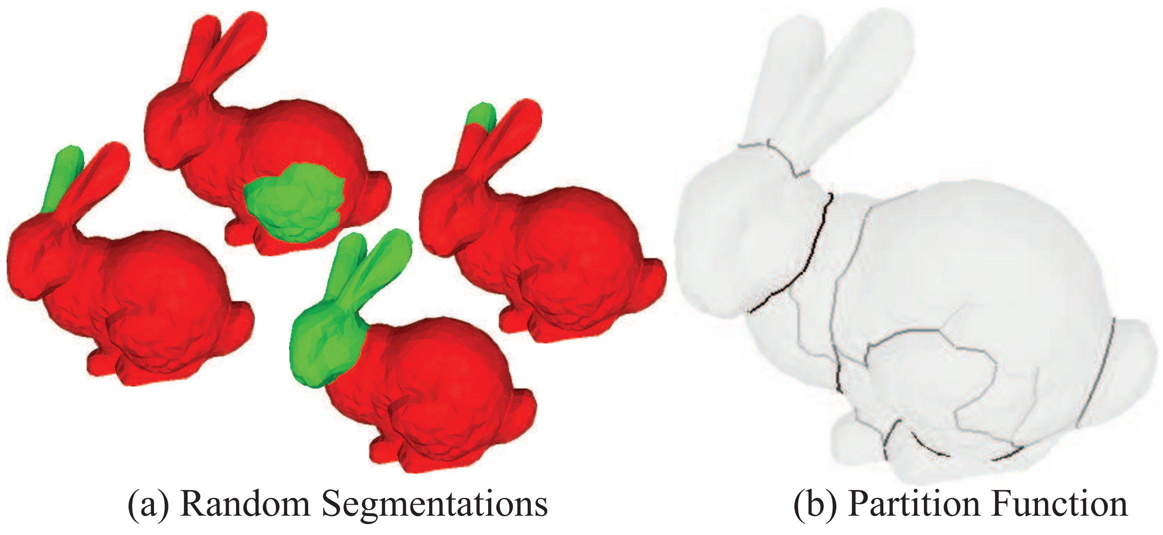


Figure 3.12: Sample randomized segmentations produced with the MinCut algorithm (a) and the resulting partition function (b).

$x_2 = 3s$), which roughly controls the “size” of parts to be generated by the segmentation.

We use this MinCut algorithm to generate a set of cuts by randomizing the selection of seed faces. In our current implementation, we consider only cuts between two seeds at a time in order to leverage fast polynomial algorithms for finding $s - t$ mincuts. In each randomized iteration, our algorithm proceeds by randomly selecting two faces, s and t, modulating the original weights on edges of the graph by $W(e, s, t)$, and then finding the minimum cut separating s and t in the modified graph using the Edmonds-Karp algorithm for finding maxflows [28]. This process is iterated to generate a sampled set of cuts from which the partition function and most consistent cuts structures are derived.

Figure 3.12 shows the partition function produced by this process for 400 iterations of the MinCut algorithm with $s = 0.05$. As in Figures 3.10 and 3.11, the left image shows four random segmentations produced by the MinCut algorithm when randomly seeded with different sink and seed faces (for example, the bottom-right segmentation came from a source on the face and a sink on the body, while the bottom-left came from a source on the ear and a sink on the face). Even though the randomized segmentations produced only two segments each, and not all of the 2-way segmentations find meaningful parts (e.g., the top-right segmentation cuts off only part of the ear), they cumulatively reveal the decomposition of the bunny into its natural set of parts, as shown in the partition function shown on the right. As you can see, the cuts produced with the MinCut algorithm are even more consistent (darker and sharper lines in the partition function) than the ones generated with Normalized Cuts.

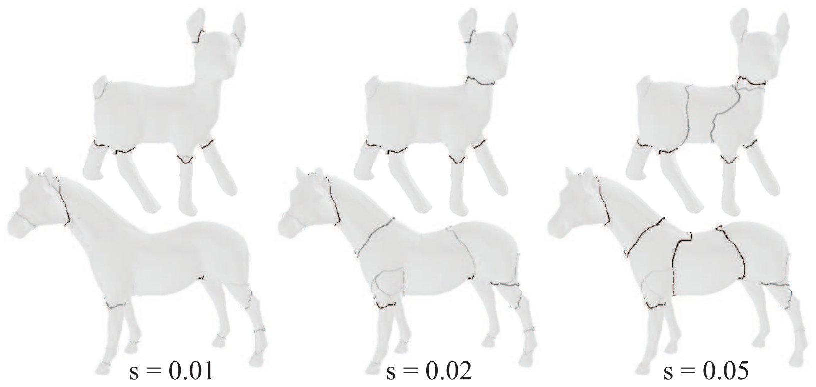


Figure 3.13: Comparison of partition functions generated with the MinCut algorithm using different settings of the s parameter, which loosely controls the "sizes" of parts cut off in random segmentations.

Figure 3.13 shows how the partition function reveals features of different size as s is varied: the ears of the dog and mouth of the horse are revealed for $s = 0.01$, and large-scale features such as the cuts through the bodies are revealed at $s = 0.05$. For some applications, it may be useful to created a collection of partition functions for a range of values of s, and treat them as a feature vector for each edge. For others, it may make sense to randomize s within a range. For the experiments in this paper, however, we simply use $s = 0.05$.

3.4.6 Experimental Results

We have implemented the algorithms of the previous section and incorporated them into the pipeline for computing the partition function and consistent cuts described in Section 3.4.4. In this section, we show examples for many types of meshes, analyze compute times, and investigate sensitivities to noise, tessellation, articulated pose, and intra-class variation. Our goal is to characterize the speed, stability, and practicality of the partition function as a shape analysis tool.

Due to space limitations, it is not possible to do a detailed analysis for all the possible randomized cut strategies described in the previous section. So, in this section, and for the remainder of the paper, we will only discuss results generated with the MinCut algorithm using $s = 0.05$. These results are representative of what is possible with randomized cuts, but possibly not as good as what could be achieved by tuning the algorithm for specific applications.

Compute Time

Our first experiment investigates the compute time and convergence rates of our randomized process for computing partition functions. We performed a timing test on a Macintosh with a 2.2GHz CPU

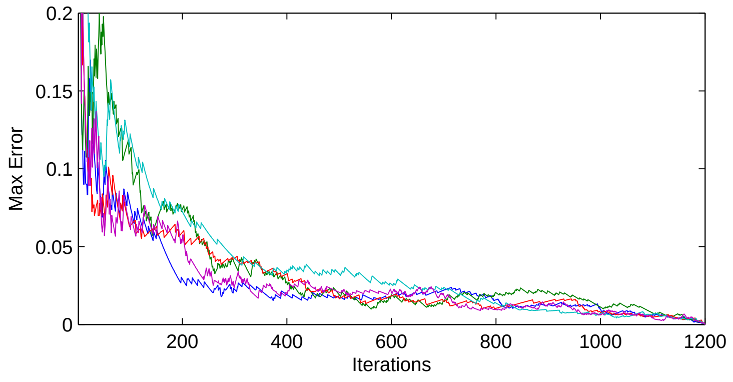


Figure 3.14: Maximum error in partition function versus iterations for five example models.

and 2GB of memory using 5 meshes from Figure 3.15 (screwdriver, mask, human, pronghorn, and bull). For each mesh, we first decimated it to 4K triangles with QSlim [35], and then generated random segmentations using the MinCut algorithm until the partition function stabilized. We measure convergence rates by reporting the *maximum* error for each iteration with respect to the final partition function value (i.e., the maximum error for any edge – RMSD values would be orders of magnitude smaller).

A plot of error versus iteration appears in Figure 3.14. We can see that after about 400 iterations, the maximum error for any edge in any of the 5 models is below 5% (this is the number of iterations used to make most images in this paper). The algorithm took .6 seconds to run per iteration, and thus an average of 4 minutes were taken to create a stable partition function for each of these models. Both the timings and convergence rates do not vary much between models of similar triangle counts, but they do increase with $O(VE^2)$ for models with N faces and E edges, as we execute Edmonds-Karp to find a mincut for every iteration.

While the computational complexity of the MinCut algorithm grows superlinearly with triangle count, this is not a practical problem for our system, since we aim to reveal large-scale part structures of a mesh, which are apparent at low triangle counts. We strike a conservative balance between mesh resolution and compute time by decimating all meshes to at most 4K triangles, which contain more than enough resolution to describe parts, but allow reasonable compute times.

Examples

Our second experiment investigates the qualitative properties of the randomized cuts generated with our methods. We do this by computing and visualizing the partition function for a wide variety of examples.

Figure 3.15 shows a visualization of the partition function computed for meshes representing animals, humans, faces, organs, and tools. From these examples, we see that the partition function generally favors edges that: 1) lie on cuts that partition the mesh with few/short/concave edges, 2) lie on cuts that partition the mesh into nearly equal size parts, and 3) do not lie nearby other more favorable cuts. The first property is a direct result of using the MinCut algorithm to produce segmentations (the cut cost favors few/short/concave edges). The second comes from both penalizing edges close to the source and sink with $W(e, X)$ and random sampling of mincut sinks and sources (edges near the center of the mesh are more likely to lie "between" randomly chosen sources and sinks). The third is a combined result of mincuts and random sampling – an edge will only get votes if it has the least cost among cuts that separate the source and sink. So, a very favorable cut will "dominate" nearby cuts whose cost is not as low.

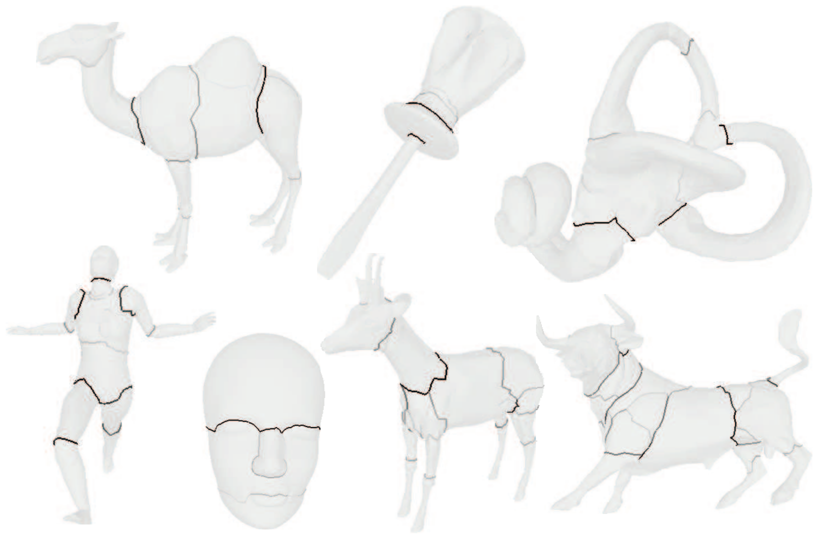


Figure 3.15: Partition function examples.

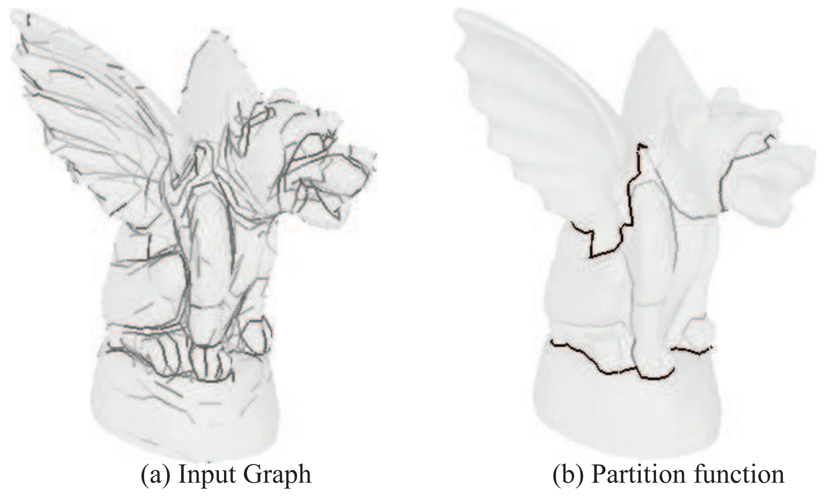


Figure 3.16: Comparison of graph edge concavity weights (a) vs. partition function (b). Note that the edge weights (left) capture only local concavity features, while the partition function captures global shape properties.

Figure 3.16 provides a comparison of the partition function (right) with the original edge concavity weights computed from dihedral angles (left) for a Gargoyle. The main difference is that the partition function captures *global* properties of the mesh. As a result, not all concave edges have large partition function values, and vice-versa. For example, for the Gargoyle, the partition function value is highest on edges that lie along the junctions of the wings-to-body, head-to-body, and feet-to-stand. In some cases, convex edges have high partition function values because they lie on a favorable seam determined by other concave edges (e.g., on the top of the shoulder), while some concave edges have low partition function values because they do not lie on a global seam (e.g., folds of feathers in the wings), or because they do not cut off a large part (junctions of ear-to-head), or because they lie near other global seams (e.g., top of the foot). The global nature of the partition function makes it more useful for applications that require stable, large-scale shape features.

Sensitivity to Pose and Intra-Class Variation

Our next experiments investigate the sensitivities of the partition function to articulated pose and variations of shapes within the same class of objects. Since edge weights on the dual graph used to generate random segmentations are determined directly from dihedral angles (which vary with changes to pose and between instances of the same class), one might wonder how robust our methods are to variations of this type.

To test sensitivity to pose, we computed the partition function and consistent cuts for seven poses of a horse provided by [104] and twenty poses of the Armadillo provided by [39]. As can be seen in Figure 3.17, the gross characteristics of the partition functions are fairly stable across different poses – the strongest cuts appear consistently along the neck, tail, face of the horse and at the key joints of the Armadillo (knees, thighs, shoulders, etc.). Even though the dihedral angles of individual edges vary with pose, the global properties of the most consistent cuts generally remain stable.

However, there are exceptions. Figure 3.17 clearly shows variations in the cuts across the torso of the Armadillo. In this case, the torso is nearly symmetric, and diagonal cuts across it have nearly equal costs. Thus, small perturbations to the edge weights can cause a different global cut to be chosen consistently. We believe that this effect can be ameliorated somewhat by randomization of graph edge weights, but experimentation with this strategy is future work.

To investigate variation across objects within the same class, we computed and compared the partition functions for all 400 meshes in all 20 classes of the Watertight Data Set of the SHREC benchmark [39]. We find that examples within most classes in this data set have similar partition functions. In particular, the most consistent cuts amongst instances of the airplane, ant, armadillo, bird, chair, cups, glasses, hand, octopus, plier, and teddy classes are very similar, while those of the bust, human, mech, spring, and table have greater variation. Representative results are shown for the human class in bottom row of Figure 3.17. While many of the prominent cuts remain consistent despite considerable variability in body shape and pose (e.g., neck, shoulders, thighs, knees, etc.), there are certainly cases where a large concave seam appears in some objects, but not others (e.g., there is a strong concave waist in three out of the six examples shown). These results suggest that many, but not all, consistent cuts are stable across instances within the same class.

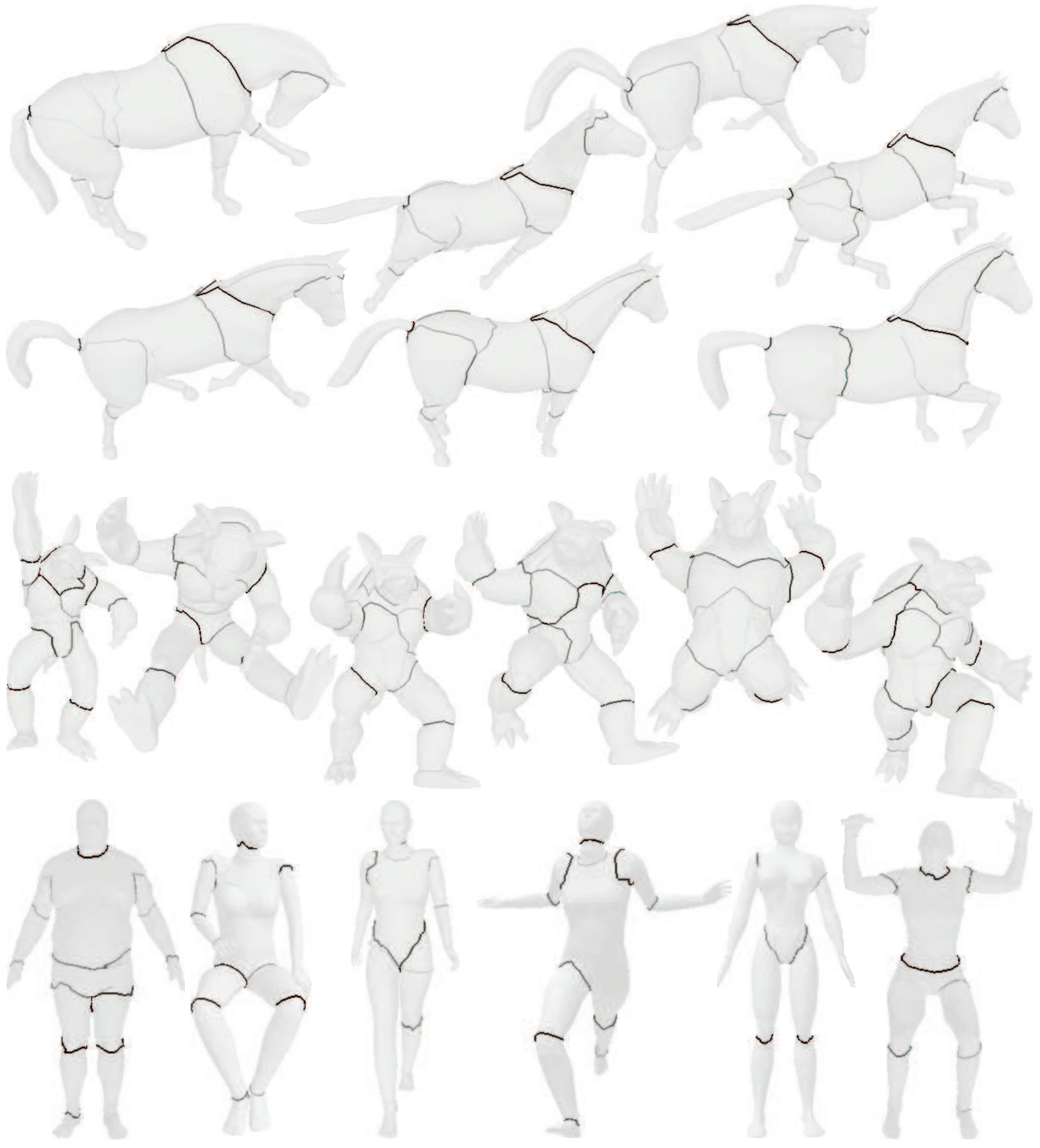


Figure 3.17: Partition function for different poses (horse and armadillo) and instances within the same class (humans). Note that many of the prominent cuts remain consistent despite considerable variability in body shape.

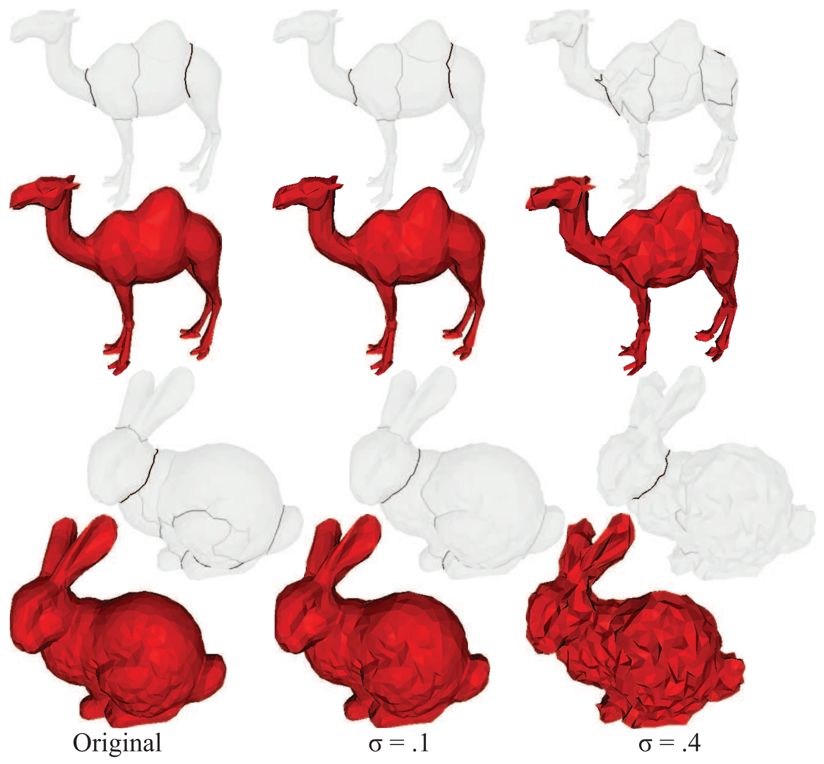


Figure 3.18: Partition function on noisy meshes: random vertex displacement of standard deviation σ times the average edge length has been applied. Note that the partition function degrades only with high noise, and even then the main cuts remain.

Sensitivity to Noise and Tessellation

Our fourth experiment studies whether the partition function is sensitive to noise and tessellation of the mesh.

To study sensitivity to noise, we calculated the partition function for a series of increasingly noisy meshes (vertices were moved in a random direction by a Gaussian with the standard deviation of σ times the average edge length). Three meshes with increasing noise (red) and the resulting partition functions (gray) are shown from left-to-right for two objects in Figure 3.18. These images show that the strong consistent cuts are largely insensitive to noise (e.g., neck), but that weaker cuts at the extremities are diminished (e.g., legs). The reason is as follows. Strong consistent cuts mainly appear at the boundaries between large, nearly-convex parts. Since they are the results of votes by many source-sink pairs selected from opposite sides of a part boundaries, they remain stable as noise is added to the mesh – i.e., there is still a lowest-cost cut that separates the large parts, and that cut is found consistently even in the presence of noise. On the other hand, for boundaries between smaller parts near the extremities, where the partition function is weaker, fewer votes are cast for the cut, and thus new low-cost cuts along concave seams created by gross vertex displacements receive

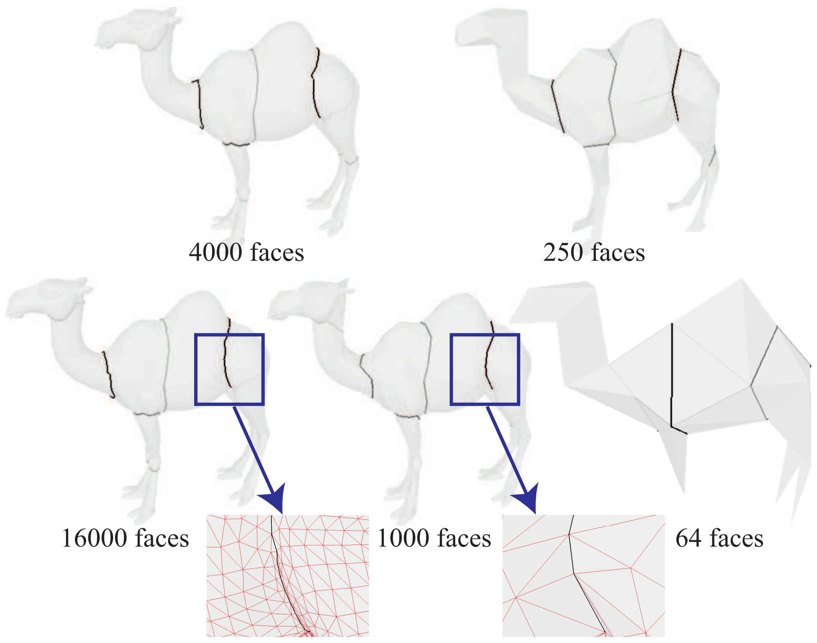


Figure 3.19: Partition function on a mesh decimated to varying resolutions. Although decimation alters the meshing considerably, the gross structure of the partition function remains largely unchanged until decimation becomes extreme. The two zoomed views show that the partition function remains consistent at the smoother segment boundaries of the 16K mesh (compared to the 1K mesh).

as many votes as they do, and their relative strength is diminished. Overall, we find that the gross structure of partition functions is stable within a moderate range of noise (e.g., less than 10% of average edge length).

To study sensitivity to tessellation, we computed partition functions for a camel model decimated to several resolutions with QSlim [35]. Figure 3.19 shows the results. Although decimation changes the graph representation of the surface considerably (increasing dihedral angles), the most consistent cuts remain largely unchanged (they can even be found after decimation to 250 faces). The exact placement of consistent cuts can vary slightly (e.g., on the neck), and extra cuts sometimes appear as the mesh is decimated and concavities become more exaggerated (e.g., around the hump). However, the original structure of cuts persists for moderate decimations. One concern may be that sharp features are less prominent at higher resolutions. As the zoomed images in Figure 3.19 show, the cuts are still found consistently as model resolution is increased. Of course, our algorithm assumes that segment boundaries are located along mesh edges, so an adversarial triangulation (such as one where edges cut across a natural boundary) or simplification so extreme that segments are broken (such as in the last image of Figure 3.19) would pose challenges to our method (and to other graph-based mesh analysis methods).

Comparisons to Alternative Methods

Our final analysis compares the proposed partition function to other functions previously used for revealing natural decompositions of graphs. The most obvious alternative is the work of [37], which computes Typical Cuts. Their method is a randomized hierarchical clustering, where the probability of merging two segments is proportional to cost of the cut between them. We have implemented this approach and find that it tends to create only very large and very small segments for our graphs (single faces whose cut costs are low get isolated). So, in our comparison, we stop contracting segments when 500 are still remaining in each randomized iteration (which we find gives the best results). A partition function calculated this way is shown in Figure 3.20b.

Another alternative is spectral embedding. Spectral segmentation methods embed the input graph in a lower-dimensional subspace, and segment in that subspace. Distances in these spectral embeddings can be used for a partition function. For example, the method of [122] creates a segmentation into n segments by first embedding the input graph into an n dimensional space, and then projecting it onto a unit sphere. We can use distances between graph nodes on that sphere to create a function similar in concept to a partition function (shown in Figure 3.20c).

Comparing results, the partition function more clearly highlights the cuts between natural parts (Figure 3.20a). Although the level of darkness is different in the three images, the second two have been tuned to reveal as much of the part structure as possible, and adjusting visualization parameters does not reveal the part structure as well as the partition function does.

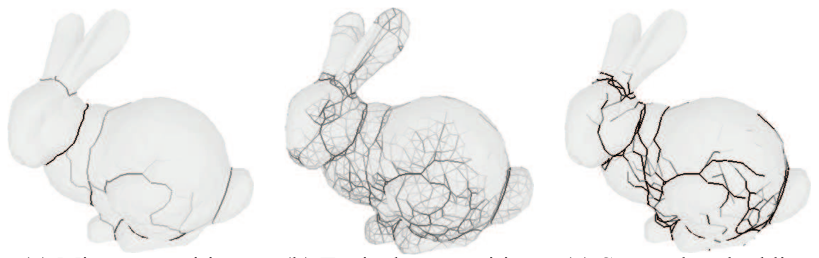


Figure 3.20: The partition function created by our method (a), compared with Typical Cuts (b), and spectral embedding (c).

3.4.7 Applications

Randomized cuts are a basic shape analysis tool that may be useful in a wide range of application domains. In this section, we investigate four applications in computer graphics.

Visualization

An obvious application of the partition function is visualization. In the figures shown in previous sections of this paper, meshes are drawn with diffuse shading in light gray, with lines superimposed over edges based on the partition function (darker lines represent higher partition function values). In most cases, this visualization provides a sense for the whole shape (due to the light gray shading), while highlighting the seams between large parts (due to the dark lines), and thus perhaps it conveys the large-scale parts structure of the surface more clearly than diffuse shading alone. Of course, it can also be useful to display a sequence of the most consistent cuts, and/or provide the user with an interactive tool to explore a ranked set of consistent cuts to better understand the structure of a surface. While we have experimented with these methods for our own understanding of partition functions, we have not investigated the full potential of visualization of surfaces based on partition functions and consistent cuts, which is a topic for future work.

Segmentation

Segmentation is an important pre-processing step to many algorithms. Many methods exist to segment models; some of these are described in Section 3.2, and some of their randomized variations are described in Section 3.4.5. The partition function and ranked cuts represent a consensus of which edges and cuts are more likely to participate in a segmentation, and are therefore useful to segment a model.

There are several ways the partition function can be used to segment models. The idea suggested in Typical Cuts [37] is to define as segments those parts of the mesh that are connected with partition function edges less than some threshold (they use .5). While this has the benefit of simplicity and a natural stopping criterion, it has two major drawbacks when applied to mesh segmentation. First, the desirable cut to make within a segment should depend on the surrounding segments. For example, there are frequently cuts of similar probabilities that are close to one another, often overlapping, and rather than segment the narrow slivers between these close cuts, it would be better to recalibrate the partition function within the chosen segment. Second, it is impractical to compute the partition function globally for all levels of detail, since most of the computation of calculating a partition function on the entire mesh is spent on finding large, global cuts.

This suggests a hierarchical algorithm: compute the partition function on the current segment (starting with the entire mesh), split the segment into child segments with the most consistent cut, and recurse. Re-computing the partition function in each segment discourages cuts adjacent to

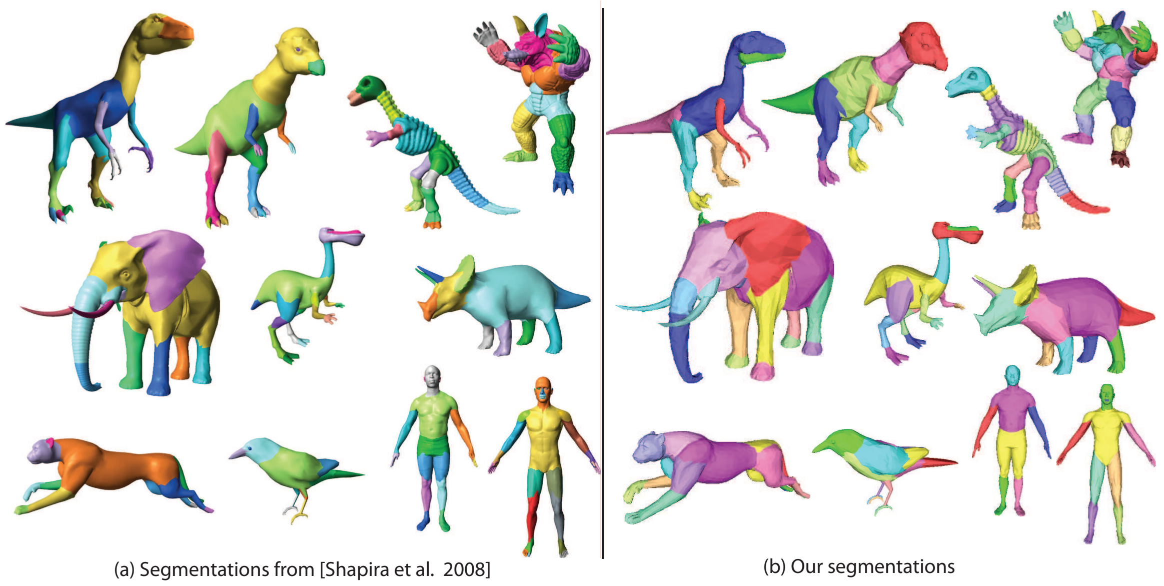


Figure 3.21: Comparison with [Shapira et al. 2008]. (a) Figure 11 from [Shapira et al. 2008] (reproduced with permission). (b) Our results. None of the segmentations are perfect. Some of their segmentations are better, and some of ours are better; overall, the quality is similar between the two methods.

existing segment boundaries, and focuses computation on cuts within the segment. However, the resulting probabilities of cuts within child segments are not normalized with respect to one another, and so we use the area-normalized cut cost (see Section 3.4.5) to choose the order of the recursion. The full algorithm is: cut the current segment with the most consistent cut, compute the partition function on the child segments and propose splits of each child segment, and put these proposed splits into a global priority queue ordered by the area-normalized cut of the resulting segmentation. The algorithm takes as input the desired number of segments, and uses that as its termination criterion.

Figure 3.21 has our segmentation results for a variety of meshes (b), and compares them to the results of [104] (a). The partition function finds segment boundaries on these meshes through a combination of their being partially concave, and/or being closer to the "middle" of previous segments. Some of the segmentations of [104] are better, and some of ours are better; overall, the quality is similar between the two methods.

Figure 3.22 shows segmentations of more shapes. The models in the bottom two rows were chosen to provide a direct comparison to the segmentation survey of [11] (to save space, we do not reprint their results, but direct the reader to their Figures 7, 8, 10, and 11). Again, we find that the quality of our segmentations appears similar to the best of prior work. However, to be fair, we have not included an automatic criterion for stopping the hierarchical segmentation, and instead

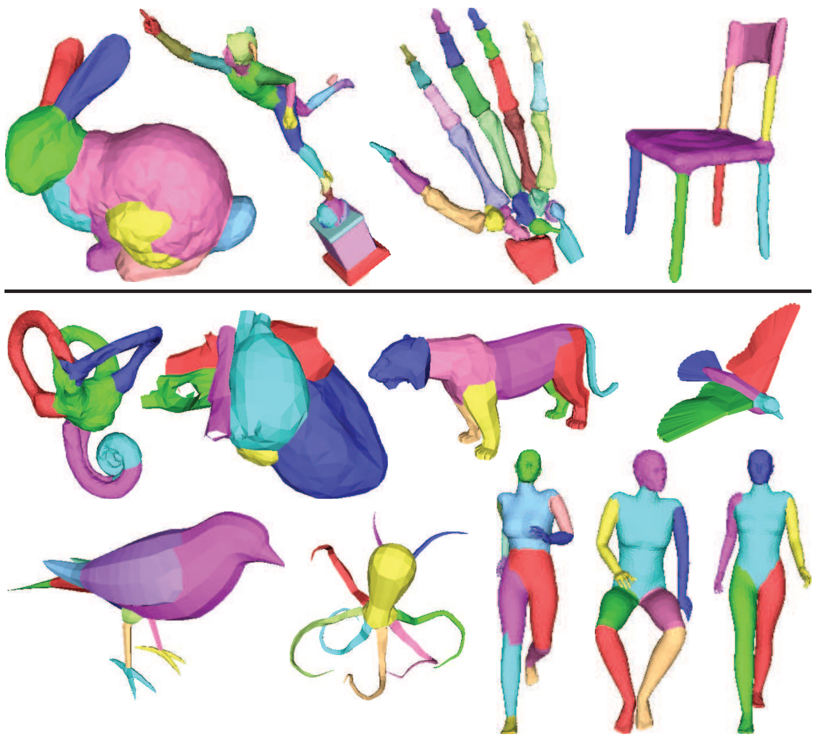


Figure 3.22: More segmentation examples. The models in the bottom two rows appeared in the segmentation comparison study of [Attene et al. 2006a] (in their Figures 7, 8, 10, and 11). The quality of our segmentations is similar to that of the best methods in their survey.

simply select a number of segments to match results of prior work (this aspect of the segmentation algorithm is orthogonal from our work on the partition function, and thus we have not addressed it in our system).

A quantitative evaluation of several segmentation algorithms on a database consisting of 180 models was performed by [24]. In their study, humans were asked to manually segment the meshes, and the results were compared to several automatic segmentation methods. We refer the reader to the paper for a thorough discussion of their methodology, and reproduce here the results of using one of the error metrics they suggest: the Rand Index, which measures, for two segmentations, how likely they are to agree that a pair of faces belongs to the same segment.

Figure 3.23 shows the Rand Index error of several segmentation algorithms (relative to ground truth segmentations). The first column shows, for reference, the error of manual segmentations relative to one another. The next columns show the errors for the following algorithms: Randomized Cuts (presented in this section), Shape Diameter Function [104], Hierarchical Segmentation (the segmentation algorithm presented in Section 3.3.3), Core Extraction [54], Random Walks [61], Fitting Primitives [10], and K-Means (the version described in [24]). Then, for baseline reference, the errors

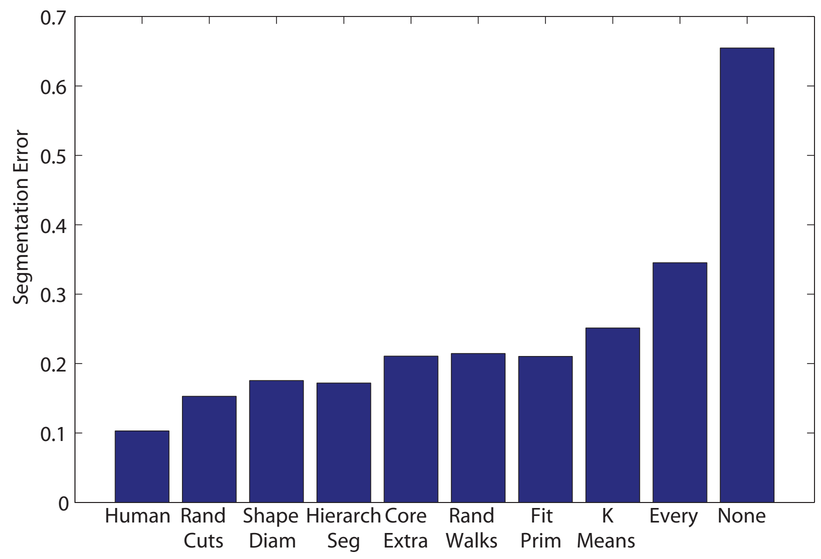


Figure 3.23: Quantitative evaluation of several segmentation algorithms (reproduced from Figure 5 in [24]) according to the Rand Index error metric. Higher bars correspond to greater error. First, the average error of a human segmentation (relative to other manual segmentations) is shown. Then, the error of the following algorithms is shown: Randomized Cuts (this section), Shape Diameter Function [104], Hierarchical Segmentation (Section 3.3.3), Core Extraction [54], Random Walks [61], Fitting Primitives [10], and K-Means [24]. Finally, for reference, the error of segmenting every face separately and then every face into one segment is shown. Note that Randomized Cuts performs better than Hierarchical Segmentation, and that both are competitive with other algorithms.

of segmenting every face separately and all faces in one segment are shown. Note that Shape Diameter and Core Extraction use mechanisms to automatically determine the number of segments, whereas the others require this number as an input parameter; the source paper contains greater detail about how these comparisons are made.

While the ranking of these algorithms fluctuates slightly depending on the error metric and object category (see the paper for greater details), Figure 3.23 illustrates three trends. First, of the two segmentation algorithms introduced in this thesis: hierarchical segmentation and randomized cuts, the latter consistently performs better. Because randomized cuts aggregates the results of many deterministically performed segmentations and is therefore more computationally intensive, this is not surprising. Second, both segmentation algorithms introduced in this thesis are competitive with state-of-the-art methods, with randomized cuts in particular performing well. Third, a large gap exists between the error induced by inconsistency of human-made segmentations, and that of the best automatic algorithms, which suggests room for considerable improvement.

Surface Correspondence

Finding correspondences between points on two different surfaces is a fundamental problem in computer graphics – it is an underlying problem in surface matching, morphing, completion, symmetrization, and modeling.

While there are many algorithms for finding a dense inter-surface mapping when given an initial coarse set of point correspondences (e.g. [101]), it is still very difficult to establish the initial correspondences automatically, even for rigid-body transformations. The problem is to derive an alignment error function that heavily penalizes mismatches of semantic features. Typical methods based on RMSD and local shape descriptors are not always effective [14], and thus most systems rely upon human-provided markers (e.g., [6]).

In this section, we propose that the partition function can be used to help establish a coarse set of correspondences automatically between two surfaces. Our intuition is based on the results shown in Sections 3.4.6 – the partition function is high in concave seams where large parts connect. Since those seams are often a consistent feature across a set of deformations and/or within a class of objects (Figure 3.17), we hypothesize that they provide a useful cue for establishing point correspondences.

To investigate this hypothesis, we have implemented a simple adaptation of the RMSD error measure for surface alignment. Rather than measuring the sum of squared distances between closest points for all points on two surfaces, we computed a weighted sum of the squared distances between closest points on edges with high partition function values. That is, given two surfaces, we compute their partition functions, sample points from edges with values above a threshold in proportion to their lengths and partition function, and then use ICP with multiple restarts to find the similarity transformation that minimizes the RMSD of the sampled point sets. This procedure effectively sets weights on the original RMSD to strongly favor correspondences between points on the most consistent cuts.

Figure 3.24 shows an example of how this simple scheme for aligning cuts can help find a coarse alignment for one model of a horse (red) with six others in different poses (green). The top row shows alignments produced with our cut-weighted alignment scheme, while the bottom row shows the results produced without it (i.e., sampling points on all edges uniformly). Note that head and tail are flipped for three of the examples in the bottom row (they are marked with a black 'X'), while all examples in the top row find good correspondences. In this case, the strong cuts along the neck, across the face, and at the base of the tail provide the main cues that produce semantically correct coarse alignments.

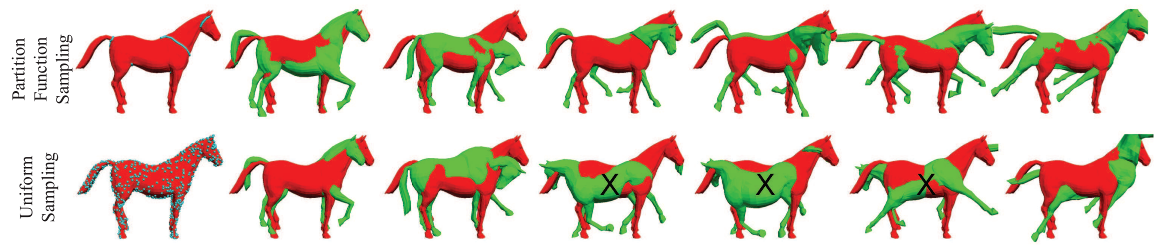


Figure 3.24: Using the partition function for registration. Alignments based on RMSD between points sampled (left image) according to the partition function (top row) provide better point correspondences than ones based on points sampled uniformly (bottom row).

Deformation

Methods for skeleton-free deformation of a mesh have recently been developed (for example, [72, 17] among others). They often allow a user to select a region of influence and a handle, and then move or re-orient the handle while the rest of the mesh deforms in a way that locally preserves surface shape while meeting the user-specified handle constraints. Such methods generally spread deformation error uniformly over a surface, which gives the surface a "rubbery" feel, sometimes creating undesirable deformations (see Figures 3.25a and 3.25c).

To address this issue, [93] described a "material-aware" deformation framework, where the stiffness of deformation is determined by material properties of the mesh. However, a user had to provide stiffness properties manually (or they had to be learned from example deformations). Here, we use our partition function to provide stiffness properties automatically. Our intuition is that edges of the mesh with high partition function values appear along seams of large parts, and thus should be very flexible, while edges having low partition function values should remain stiff.

Our implementation is based on the deformation framework of [72]. Their framework reconstructs a mesh from local coordinates in two steps: first, the Frenet frames of vertices are solved for to maintain the changes in frames between adjacent vertices, and second, the positions of the vertices are found. We modify the first step so that edges neighboring *high* partition function edges are *less* obligated to preserve frame changes from the original model. To extend the partition function to neighboring edges, if $p(i)$ is the partition function on edge i, we form $p'(i)$ as $p'(i) = \max_{j \in N(i)} p(j)$ (where $N(i)$ is the set of edges adjacent to edge i). We then invert the partition function as: $w(i) = \frac{\alpha}{\alpha + p'(i)}$. These $w(i)$ are the weights we use in the linear system that reconstructs frames to determine how closely the change in frames across edge i must be preserved from the original model. We choose $\alpha = .5$, so that an edge bordering a definite partition has a weight of 1/3, and an edge bordering no partition has a weight of 1.

Figure 3.25 compares deformations created with (3.25b and 3.25d) and without (3.25a and 3.25c) weights determined by the partition function. Note that deformations with weights from the partition function preserve the rigid shape of the head and leg of the cheetah, spreading the deformation into the likely segment junctions favored by the partition function. In contrast, the deformations with uniform weights warp the head and leg of the cheetah in an unnatural-looking way.

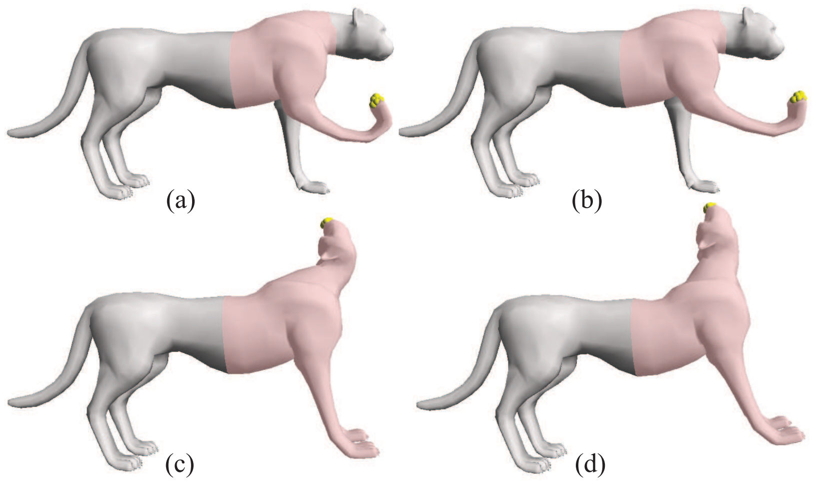


Figure 3.25: Spreading deformation error uniformly can lead to unnatural deformations (a), (c), whereas allowing more deformation near edges of high partition value leads to more natural, segment-like deformations (b), (d) The deformations were made by setting a region of influence (pink) and adjusting the orientation of handles (yellow spheres).

3.4.8 Conclusion

The main contribution of this paper is the idea that randomized cuts can be used for 3D shape analysis. This idea is an instance of a broader class of techniques where randomization of discrete processes produce continuous functions. In our case, the combination of multiple randomized mesh segmentations produces a continuous partition function that provides global shape information about where part boundaries are likely to be on a mesh. This information is stable across several common mesh perturbations, and thus we find it useful in mesh visualization and processing applications.

While the initial results are promising, there are many limitations, which suggest topics for further study. First, our MinCut algorithm produces unstable results for symmetric objects (such as the chest of the Armadillo in Figure 3.17), as it favors one of multiple cuts of similar costs. Perhaps this problem could be ameliorated by randomizing graph edge weights, but further investigation is required. Second, our study considers a limited set of methods for randomizing segmentation algorithms. For example, it might be useful to randomize algorithm selection, numbers of segments, scales, and other parameters in ways that we did not investigate. Third, we have considered only scalar representations of the partition function. Perhaps it could be extended to multiple dimensions by encoding for each edge the value of the partition function for different scales, numbers of parts, and other parameters that affect segmentations. This could provide a feature vector for every edge that could be useful for visualization or shape matching. Fourth, our study of applications is a small first step. For example, we believe that the partition function could be leveraged more effectively for segmentation, and perhaps it could be used for chokepoint analysis, saliency analysis, feature-preserving smoothing, skeleton embedding, grasp planning, feature detection, and other applications in computer graphics.

Chapter 4

Analysis of Scenes

4.1 Introduction

Detailed models of cities with semantically tagged objects (e.g., cars, street lights, stop signs, etc.) are useful for numerous applications: city planning, emergency response preparation, virtual tourism, multimedia entertainment, cultural heritage documentation, and others. Yet, it is very difficult to acquire such models. Current object recognition algorithms are not robust enough to label all objects in a city automatically from images, and interactive semantic tagging tools require tremendous manual effort.

However, new types of data are now available to assist with urban modeling. There has been a recent explosion in worldwide efforts to acquire 3D scanner data for real-world urban environments. For example, both Google and Microsoft have been driving cars with LIDAR sensors throughout most major cities in North America and Europe with the eventual goal of acquiring a high-resolution 3D model of the entire world. This new data opens unprecedented opportunities for object labeling and city modeling. Traditionally, range scan processing algorithms have focused either on small objects in isolation or on large objects in scenes. Never before has it been possible to reason about all small objects in an entire city. In this paper, we take a step in this direction by developing a set of algorithms to locate, segment, represent, and classify small objects in scanned point clouds of a city.

Data representing geometry of this scale is relatively new, and not many algorithms exist to try to identify objects from 3D data in real-world cluttered city environments. Algorithms have been proposed for modeling specific object types (e.g., buildings [5, 27] and trees [119, 123]), for extracting

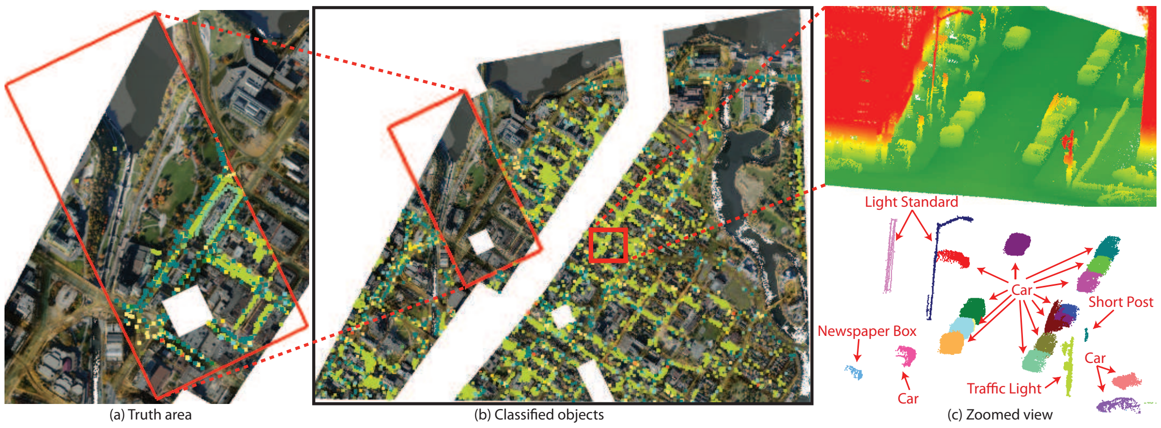


Figure 4.1: Our method recognizes objects in 3D scans of cities. In this example, it uses about 1000 manually labeled objects in the truth area area (a) to predict about 6000 objects elsewhere in the scan (b). (Objects are depicted as colored points, with colors representing labels.) A zoomed view, with the height-encoded input points on top, and classified and segmented objects on bottom, is shown in (c).(Automatically generated classifications are shown in red text, and colors denote object instances.)

geometric primitives (e.g., [95, 115]), and for identifying objects in cluttered scenes [7, 63, 31]. However, they have been demonstrated only for synthetic scenarios and/or for small scenes with relatively few object categories.

In this paper, we describe a system for automatically labeling small objects in 3D scans of urban environments. Our goal is to characterize the types of algorithms that are most effective to address the main challenges: location, segmentation, representation, and classification of objects. For each component, we provide several alternative approaches and perform an empirical investigation of which approaches provide the best results on a truthed data set (Figure 4.1a) encompassing a large region of Ottawa, Canada [86] and containing about 100 million points and 1000 objects of interest. Our results indicate that it is possible to label 65% of small objects with a pipeline of algorithms that includes hierarchical clustering, foreground-background separation with minimum cuts, geometry and contextual object description, and classification with support vector machines. We then use this truthed data set as training to recognize objects in a larger scan of Ottawa (Figure 4.1b), which contains about a billion points. An example input scene is shown at the top of Figure 4.1c, and the output labeled, segmented objects are shown at the bottom of Figure 4.1c.

4.2 Related Work

We describe related work in detection of objects in point clouds, labeling of individual points in point clouds, and shape descriptors.

Detection of Objects in Point Clouds Much of prior analysis of urban point clouds concentrates on reconstructing buildings. Fitting parametric models is often used for low-resolution aerial scans [60, 19, 87, 121] and partial scans of buildings [23]. Frueh et al. [20] developed a method for reconstruction of densely sampled building facades and filling occluded geometry and texture. A lower quality but faster reconstruction was presented by Carlberg et al. [75].

Point cloud data has also been used to find roads, trees, and linear structures. Jaakkola et al. [49] developed a method for identifying road markings and reconstructing road surface as a triangular irregular network. Among smaller objects, trees drew attention of a good number of researchers. Wang et al. [123] developed a method for detecting and estimating 3D models of trees in a forest from a LIDAR point cloud. Xu et al. [119] created visually appealing reconstructions from a densely sampled point cloud of a tree. Lalonde et al. [64] classify natural terrain into “scatter”, “linear”, and “surface”.

These methods are synergistic with ours, as reconstructed models of buildings and small objects can be combined to form a more complete model of a city.

Point Labeling Several papers use statistical models to label points in scenes based on examples. In [63], points are labeled with a bayesian classifier based on local properties. Several papers have adapted the machinery of Markov Random Fields to the problem of labeling 3D points [7, 117, 110]. In this approach, the label of a point is assumed to depend on its local shape descriptor (to assign similar labels to similar shapes), and on its neighbors (to assign smooth labels). These methods have only been demonstrated on synthetic or small scenes, with relatively few object categories. The primary difference between these methods and our approach is that we assign labels at the level of object instances, rather than individual points.

Shape Descriptors There has been considerable work on constructing local and global shape descriptors [31, 51, 15]. The work focuses on making shape descriptors more discriminative for object classification, and on either determining canonical frames or adding invariances to the descriptors. In particular, [31] and [51] propose methods to find 3D models in cluttered scenes by starting with proposed correspondences from scene points to query model points that match shape descriptors. Shape descriptors based on spin images were used by [79] and [78] to categorize objects such as vehicle types in 3D point clouds. We combine spin images with other shape and contextual features to recognize a wide variety of object types throughout an entire city.

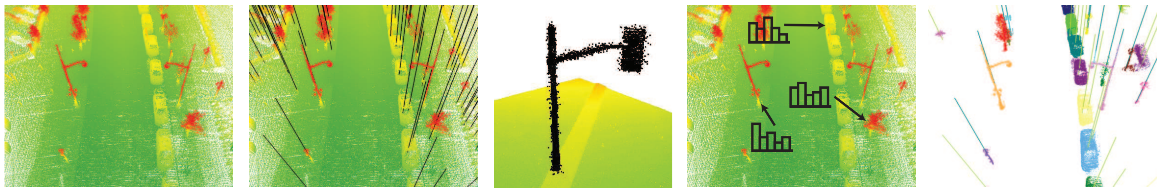


Figure 4.2: Overview of our system. The input is a point cloud representing the city (a). First, locations for potential objects are identified (b). Second, they are segmented (c). Third, features are constructed describing the objects' shape and context (d). Finally, these features are used to classify the objects (e).

4.3 Method

4.3.1 Overview

Our system takes as input a point cloud representing a city and a set of training objects (2D labeled locations), and creates as output a segmentation and labeling, where every point in the city is associated with a segment, and every segment has a semantic label (possibly "Background"). The system proceeds in four steps, as outlined in Figure 4.2. First, given the input point cloud (Figure 4.2a), we generate a list of locations for potential objects of interest – e.g., where point densities are highest (Figure 4.2b). Second, we predict for each of these potential locations which of the nearby points are part of the object and which are background clutter (Figure 4.2c). Then, we extract a set of features describing the shape and spatial context of the object (Figure 4.2d), and use them to classify the object according to labeled examples in the training set. The end result is a set of labeled objects, each associated with a set of points (Figure 4.2e).

The following four sections describe these steps in detail. For each step, we discuss the challenges, alternatives, algorithms, and design decisions made in our system. We present results of experiments aimed at quantitatively evaluating the performance of each stage of our system in comparison to alternatives in Sections 4.4.1- 4.4.3. The results of the system run on an entire city are described in Section 4.4.4.

4.3.2 Localization

The first step for our system is to start with a point cloud and find candidate object locations. This step needs to find at least one location per object (ideally close to the center of the object), while minimizing false alarms. Although multiple locations per object and false locations are undesirable, they can be merged and eliminated in future processing steps, and so our main goal in this step is to not miss true object locations.

Our first step is to remove points that clearly are not part of small objects. We filter out points close to the ground, which is estimated at uniformly spaced positions with iterative plane fitting. We then remove isolated points. Finally, we filter out points likely to belong to buildings by removing very large connected components. Once these filters have been run, we proceed with one of four approaches to find potential object locations.

Since objects of interest are likely to rise above their local surroundings, a simple approach to finding potential object locations is to generate a 2D scalar image representing the "height" of the point cloud, and performing image processing operations to find local maxima. We experimented with several variations of this approach. The most successful variant is to generate an image using the maximum height of the points in each pixel, run a difference of Gaussian filters to find high frequencies, and then extract connected components with area under a threshold to find small objects. This method is effective at finding isolated poles, but performs worse for cars and objects amongst clutter.

Another approach stems from the observation that objects are often found at local maxima of point density. A reasonable way of finding such maxima is to adapt a Mean Shift [32] approach to start with evenly spaced potential locations and iteratively move each location to the center of its support (we use all points a horizontal distance of 2m as the support). This method has two problems: first, in order to find sufficiently small objects, the initial location spacing needs to be small, leading to unnecessary computation, and, second, the support size is difficult to set to be effective for small and large objects.

The third method postulates that objects locations are likely to be in the center of connected components. The algorithm then extracts from the scene connected components at some distance threshold, and creates an object location at the center of each connected component. To reduce the number of false locations, we reject clusters that are too small or whose lowest points are too high (since we are interested in ground-based objects). This approach has trouble with objects that are sampled at different rates and with objects that are near other objects or background.

Finally, the fourth method refines the connected components approach by creating a better clustering of the point cloud and placing object locations at cluster centers. The algorithm proceeds by building a nearest neighbors graph and using a clustering algorithm similar to normalized cuts [105] to extract clusters of points. Specifically, we create a K-nearest neighbors graph (with $K = 4$), connect nearby disconnected components, and weigh edges as a Gaussian on their length with a standard deviation of the typical point spacing (.1m for our data). Because the k-nearest neighbors

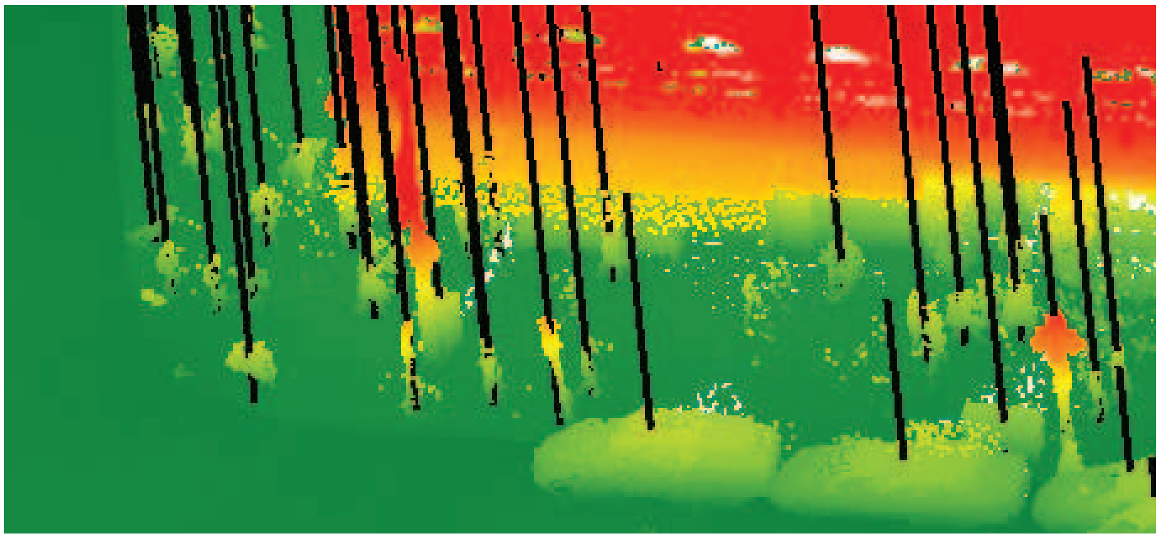


Figure 4.3: Sample results for the location finding algorithm, using normalized-cut clustering.

graph often results in several disconnected components within each foreground object, we connect the closest point pairs of disconnected components. Then, we cluster by starting with each point in its own cluster and greedily merging to minimize sum of the ratio of each segment's cut cost to its number of points. Note that this clustering method is similar to the hierarchical mesh segmentation introduced in Section 3.3.3. The differences are that (i) the graph reflects the structure of 3D point cloud rather than a mesh, and (ii) the error is normalized by number of points rather than area. We stop the merging when the reductions of this error fall below a pre-set threshold. As with the connected components algorithm, we reject clusters that are too small and too high.

Example results of the normalized cut localizing method are shown in Figure 4.3, with the resulting locations depicted as black vertical lines. Note that larger objects such as cars are sometimes assigned two locations, and that building exteriors and interiors are sometimes erroneously assigned locations.

4.3.3 Segmentation

Once potential object locations are found, the objects need to be segmented from the background. This segmentation stage has two purposes: first, it will identify the object shape so that shape descriptors can be applied in the next stage, and, second, it will identify the segmentations and assign points to objects once the potential objects have been classified. We explore three approaches to segmentation.

One approach may be to use all above-ground points within a preset horizontal radius, and under

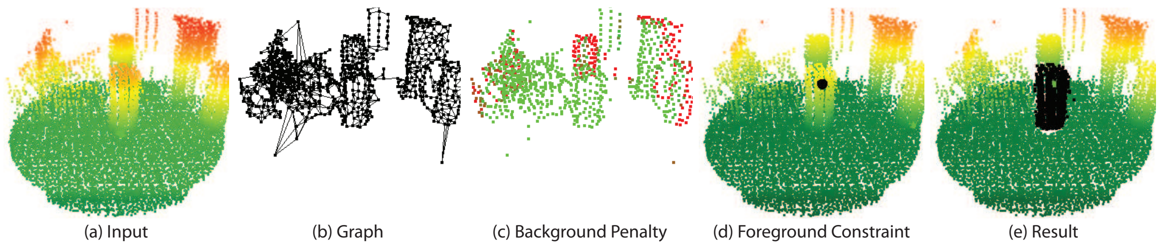


Figure 4.4: Overview of our segmentation system. (a) The system takes as input a point cloud near an object location (in this case, a short post). (b) A k-nearest neighbors graph is constructed. (c) Each node has a background penalty function, increasing from the input location to the background radius (visualized with color turning from green to red as the value increases). (d) In the automatic version of our algorithm, a foreground point is chosen as a hard constraint (in the interactive mode, the user chooses hard foreground and background constraints). The resulting segmentation is created via a min-cut (e).

a pre-set height. While this method has high recall, since it consistently includes almost all of the points inside small objects, it has a low precision as it does not try to exclude the background.

A simple way to extract the foreground from background is to start with the closest point to the predicted object location at a pre-defined height and define the foreground object as all connected points, where points are connected if they lie within some distance threshold. In many cases, objects are isolated from their surroundings, and this method works well. However, due to noise, sparse sampling, and proximity to other objects, there are cases in which no distance threshold exists that separates the foreground from background without also partitioning the foreground).

The intuition behind our final algorithm (based on min-cut) is that a good foreground segmentation consists of points that are well-connected to each other, but poorly connected to the background. The overview of our method is shown in Figure 4.4. Given an input scene (Figure 4.4a), we use the same nearest-neighbor graph from the previous section (Figure 4.4b) to encourage neighboring points to be assigned the same label. Then, given as input an expected horizontal distance to the background, we create a background penalty (Figure 4.4c) that encourages more distant points to be in the background. In an automatic or interactive manner, we add hard constraints for foreground and, optionally, background points (Figure 4.4d). Our algorithm returns the segmentation generated by the min-cut, which (i) minimizes the cut cost of the nearest neighbor graph, (ii) minimizes the background penalty, and (iii) adheres to the hard foreground or background constraints (Figure 4.4e). This approach is similar to that of [18] for image segmentation. We now describe it in greater detail.

Basic Setup

The input to the segmentation algorithm is (i) a 2d location and (ii) a suspected background radius (horizontal radius at which we assume the background begins). The nearest-neighbors graph ensures smoothness of segmentation, and a soft background penalty, which encourages points far from the object location to be labeled as background.

Note that the cut cost of a potential segmentation of this graph takes into account both distances between points in a foreground/background cut, and the density of points on the boundary via the number of broken link edges. This makes the min-cut algorithm more robust to spurious connections between foreground and background. Note also that the construction of the graph makes it adaptive to the point cloud resolution, without requiring a pre-defined threshold. When the min-cut is computed, this graph ensures that the segmentation is smooth (neighboring points are more likely to be assigned to the same segment) and that a larger separation between foreground and background is encouraged.

Given a background radius, we create a soft background penalty whose goal is two-fold: to strongly encourage points at the background radius to be labeled as background, and to encourage components only loosely connected to the object location to be in the background. We create a point-wise background penalty $B(p)$ that is added to the total cost of the cut for every point p chosen to be in the foreground. This is done by introducing a virtual background node that connects to every real node with edges of cost $B(p)$.

This background penalty $B(p)$ can be set to reflect any background prior. In our system, we make $B(p)$ a function of the horizontal distance r to the object location (relative to the background radius). The penalty consists of two linear components (Figure 4.5). The first component begins at a small distance, and increases relatively slowly (Figure 4.5a). This ensures that components that are disconnected or weakly connected to the foreground are encouraged to be in the background. The second component begins at a large distance, closer to the background radius, and increases rapidly (Figure 4.5b), ensuring that points near the background radius are labeled as background.

Performing Segmentation

The previous section described the how the graph is set up to encourage two properties with soft constraints: a smoothness error that encourages nearby points to have the same label, and a background penalty that encourages points close to the background radius to be in the background. It remains to specify constraints that encourage points to be in the foreground. Our algorithm can be

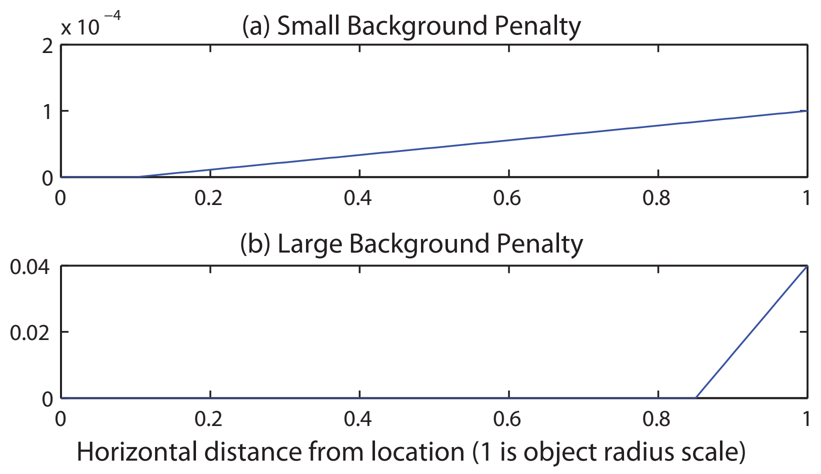


Figure 4.5: Background penalty, as a function of horizontal radius from the input location (normalized so that 1 is the input background radius). The penalty consists of two components: (a) a penalty starting at a small radius but rising slowly that encourages points disconnected from foreground to be in the background, and (b) a steep penalty close to the background radius that mandates that points outside of the background radius be in the background.

run in two regimes: automatically, and interactively, both adding hard constraints. In both cases, the final segmentation is found with a min-cut. Below, we describe the automatic regime (including automatically choosing the background radius), the interactive regime, and some accelerations.

Automatic Regime In the automatic version of our algorithm, some assumption for the foreground needs to be made. In our system, we include as a hard constraint the point closest to the (horizontal) object location at a predefined height (we use 1.2m) and its M closest neighbors in the foreground (we use $M = 3$).

The algorithm given so far produces an automatic segmentation of objects with a radial scale given as input. Some applications (such as extracting features from multiple segmentations) benefit from this parameter, but for other applications, a fully automatic selection of this radial scale is useful. To select the background radius (which ranges from 1m to 5m for our objects of interest), we run the min-cut algorithm for several iterations to automatically determine the best background radius for the segmentation. Starting from the smallest radius in the range, we run the above algorithm, and increase the radius to the maximum of the range (by 1m increments) until (i) the number of foreground points exceeds a threshold (we use 35) segmentation and (ii) the resulting cut is below a threshold (we use .4).

Two examples of choosing the background radius automatically are shown in in Figure 4.6. For these objects (a car and a sign), segmentations are shown for $R = 2m$ and $R = 4m$, and the chosen

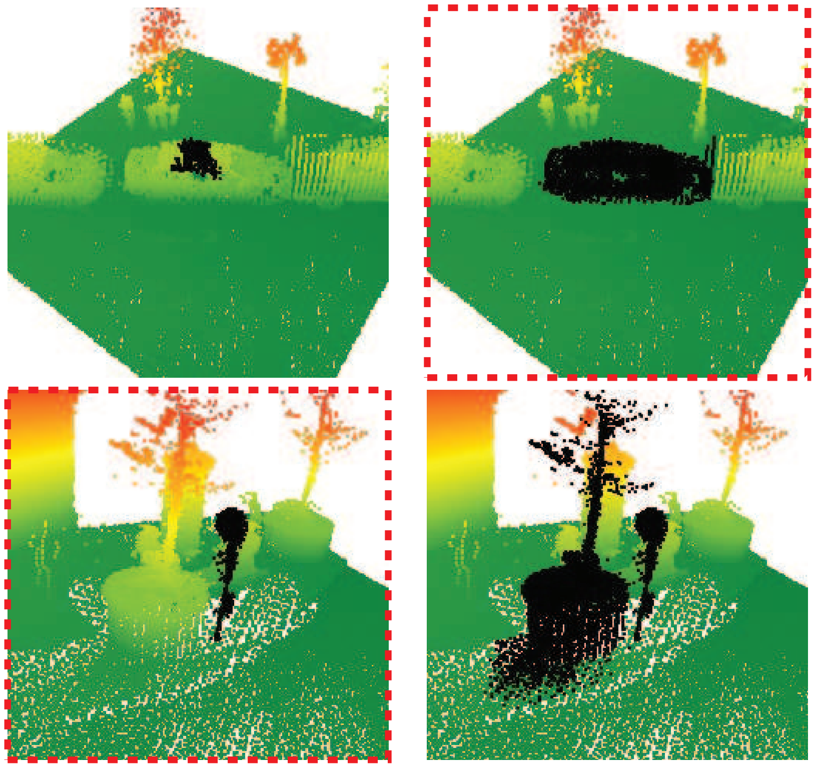


Figure 4.6: A constant choice of background radius cannot segment both examples above (R = 2m and R = 4m are shown). To automatically choose the background radius, we run the min-cut algorithm for several choices, and select the smallest radius that results in a segmentation with a small enough cut cost and large enough foreground size, choosing correctly the segmentations outlined by dotted squares in the above examples.

radius is outlined. These examples illustrate that it is difficult to choose a static background radius that works for a range of objects, but that it is possible to automatically determine (even without additional prior information) the appropriate radius for a particular object.

Interactive Regime While no automatic algorithm will be completely successful, in some scenarios it may be practical to use an interactive segmentation tool. Such a tool should follow the user-constraints at interactive rates, while automatically making a reasonable guess in unconstrained regions, and allowing any segmentation to be reached with sufficiently many constraints. Similar to the ideas of [18] our min-cut algorithm is easily set up for such a tool.

The interactive algorithm starts with the graph and background weights given in previous section. Instead of assuming a foreground constraint, as in the automatic algorithm, we allow the user to iteratively add (and remove) points as hard background or foreground constraints. The segmentation is re-calculated as the min-cut under these constraints.

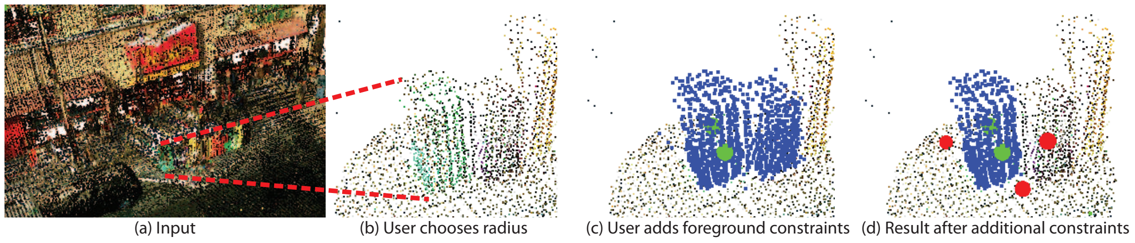

(a) Input
(b) User chooses radius
(c) User adds foreground constraints
(d) Result after additional constraints

Figure 4.7: The user surveys the input scene (a), and chooses a radius that includes the object to segment (b). The user creates several foreground constraints (green circles), and a segmentation is interactively performed, with foreground point shown in blue (c). If necessary, the user adds additional constraints (background constraints in red), until the segmentation is satisfactory (d). The user has the option of toggling between views of all points (as shown here), or only background or foreground points, to make sure the object is not over or under segmented.

The interactive tool is shown in Figure 4.7. To create a segmentation, the user looks at the input scene (Figure 4.7a), and selects a radius that includes the object to segment (Figure 4.7b). Note that for an object such as the shown newspaper box, automatic segmentation is very difficult since the box is connected to adjacent newspaper boxes. To perform manual segmentation, the user adds foreground and background constraints as necessary, responding to the interactively generated segmentation until the result is satisfactory (Figure 4.7c, d). Note that the nearest neighbors graph ensures that each successive segmentation is a smooth extrapolation of the constraints.

Accelerations Segmentation code is likely to be used at least once for each object of interest. A scan of even a moderately sized city or town will have tens of thousands (if not more) of objects of interest, so it is important for the algorithm to run quickly. A typical object will have from 10 to 200 thousand points in its radial support, so the basic algorithm described above will run slowly (the $O(n^3)$ cost of the min-cut algorithm is the initial bottleneck). We describe several accelerations to the basic algorithm that reduce the running time from about 10s to about .1s per object.

To reduce the number of nodes on which the min-cut is performed, we contract the graph by hierarchically merging nodes. A number of clustering errors may be used; we order the nodes to merge by the distance between their centroids, and merge nodes until the graph reaches a pre-set size (we use 1000 nodes). Note that arc weights are added to form outgoing arcs from a newly merged node, so the min-cut is altered only if two nodes on different sides of the correct min-cut are merged during this stage.

After introducing the above contraction, finding K-nearest neighbors (using a KD-tree) and the contraction itself become the bottleneck. To reduce the number of nodes that serve as input to finding nearest neighbors and contraction, we add a pre-process that creates an axis-aligned grid and creates a graph node at each occupied grid cell, at the location of the centroid of points in that cell. We use a grid spacing of .2m. Note that this stage adjusts the cut costs of those grid cells that have multiple nodes.

4.3.4 Feature Extraction

The previous two stages yield segments representing potential objects. In this stage, features are extracted describing the shape of the objects as well as their context. Since this step takes as input automatically generated potential object locations, which include spurious noise as well as background objects, the features generated here must distinguish object types from one another as well as from the background. We investigate both shape and contextual features.

Shape Features. We begin with features that describe the shape of the object in an orientation-invariant way. We first compute several quantities that describe the segmented point set: the number of points, estimated volume, average height, standard deviation in height, and the standard deviations in the two principle horizontal directions. Next we append a spin image descriptor [51] of the shape centered at the predicted object location with a radius of 2m and central axis perpendicular to the ground.

Multiple Segmentations. Different segmentation methods provide different information about the geometry of the object. A segmentation that takes all points within a radius, for example, consistently retrieves the entire object, but fails to remove the background. Min-cut based methods, on the other hand, usually remove the background but are less consistent to include all of the object. To take advantage of the different segmentations, we append together the above shape features computed on several segmentations. In particular, we use all-above ground points at 2m, min-cut at 4m, and min-cut with automatic radius.

Contextual Features. The position of an object relative to its environment is a useful cue about its type. Cars, for example, are found on streets, often in a line, whereas lampposts are found on sidewalks, sometimes in a pattern. We extract features that describe such cues.

Because digital maps exist that are freely available for most cities, we incorporated one (OpenStreetMap [1]) into our automatic algorithm. The first contextual feature we extract is the distance to the nearest street.

Then, we create a feature that indicates where objects are likely to be with respect to other objects. Specifically, we locally orient each training truth object with respect to its closest street. Then, we create a histogram on a 2-d grid of the locations of other objects of that class in this local orientation. We aggregate these grids for objects of each class, creating a locally orientated "autocorrelation"-like grid for each object type. This tells us, for example, that the presence of a car predicts another car further down the street. Then, for each object type, for both training and testing, we create a "prediction" grid by adding to a globally oriented 2d grid the autocorrelation grids locally oriented at each object location. This feature is able to provide, for each 2d location, n features (if there are n object types), with the each feature indicating the likelihood of an object with the corresponding label at that location. To create these globally-oriented prediction grids, for training, we use the correct object locations and labels, and for testing, we use the automatically generated locations, classified by the previous features.

4.3.5 Classification

In the final stage, we classify the feature vector for each candidate object with respect to a training set of manually labeled object locations. The training set does not include examples of background objects, and thus we augment it with automatically generated locations that are not close to truth objects, and label them as "Background". Then, during the testing stage, any query location that is classified as "Background" is assumed to be part of the background, and is disregarded.

We experimented with several classifiers using the Weka [116] toolkit, including: a k-nearest neighbors (NN) classifier with $k = 1$ and $k = 5$, random forests, and support vector machines (SVM) with complexity constant $C = 2.5$ and 5th order polynomial kernels. A comparison of their performance can be found in Section 4.4.3.

4.4 Results

We tested our prototype system on a LIDAR scan covering 6 square kilometers of Ottawa, Canada [86]. The data was collected by Neptec with one airborne scanner and four car-mounted TITAN scanners, facing left, right, forward-up, and forward-down. Scans were merged at the time of collection and provided to us only as a single point cloud covering containing 954 million points, each with a position, intensity, and color (Figure 4.1b). The reported error in alignments between airborne and car-mounted scans is 0.05 meters, and the reported vertical accuracy is 0.04 meters. Since the colors collected with car-mounted scanners were not very accurate, we focused on using geometry for classification in this study.

Ground truthing was performed in conjunction with BAE Systems within an area of the city covering about 300,000 square meters and containing about 100 million points (Figure 4.1a). Within

	Precision			Recall	
Method	Predicted	Correct	(%)	Found	(%)
Image Filters	3267	423	(13)	510	(48)
Mean Shift	17402	573	(3)	680	(64)
CC Clustering	9379	1287	(14)	962	(90)
NC Clustering	10567	1236	(12)	976	(92)

Table 4.1: Performance of localization algorithms. The first column has the number of predicted locations, and how many were in range of an object (precision). The second column shows the number of objects located and their percentage, out of the 1063 in our dataset (recall).

this area, all objects of the types listed in Table 4.5 were manually located and classified. The object types were chosen to describe man-made objects found in outdoor urban scenes whose sizes range from fire hydrants to vans. This "truth" data provides the basis for our quantitative evaluation experiments (we split it into training and testing regions for the recognition experiments) and the training data for labeling all objects throughout rest of the city.

4.4.1 Localization

To quantitatively evaluate the localization methods, we ran each algorithm on the truth area, and recorded the number of locations produced by each algorithm, how many of these locations were in range of an object (within 1m), and how many of the truth objects were in range of a location (Table 4.1).

The 3D clustering algorithms performed best, with the normalized cut clustering locating more truth objects than the connected components. Class-specific location results are shown in Columns 3-4 of Table 4.5. Note that some cars and fire hydrants are difficult to locate because the former are very sparsely sampled and the latter are both very small and sparsely sampled, making it difficult to distinguish from isolated, small blobs of noise. In addition, cars show worse performance because of the choice of a constant 1m distance threshold for evaluation of all classes; truth car locations are at the centers of cars, so a predicted location at the hood of a car, for example, is often evaluated as incorrect.

4.4.2 Segmentation

We first present some example segmentation results, and then perform a quantitative evaluation.

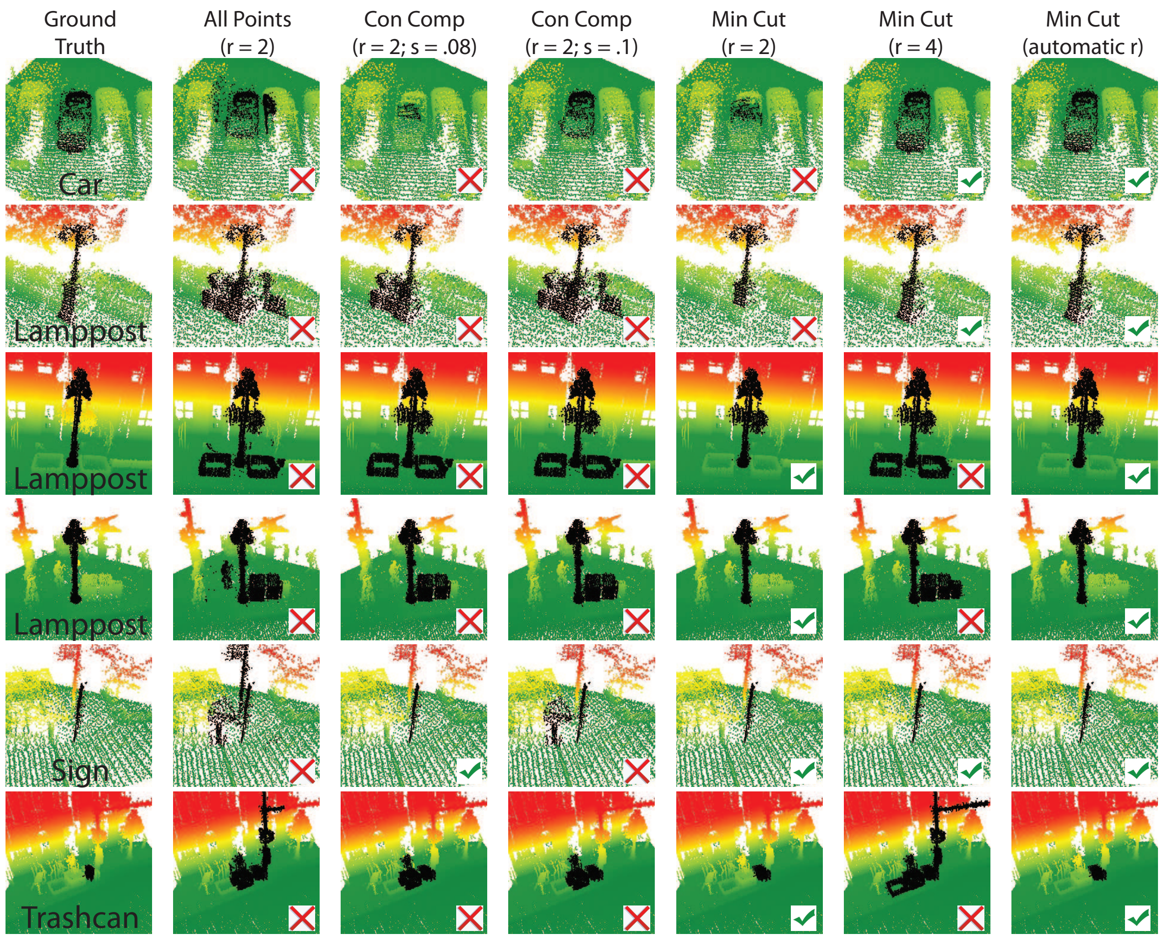


Figure 4.8: Example segmentations. Each row has an object with ground truth segmentation, followed by an all-points segmentation, two connected component segmentations with different spacings, two min-cut segmentations with different static background radii, and a min-cut segmentation with automatically chosen background radius. While connected components and min-cut with static background radii are sometimes successful, the min-cut segmentation with automatic background radius is more robust to clutter and varying object sizes.

Examples

In this section, we show several example segmentations created with our method as well as with alternatives. While a more complete, quantitative comparison is performed in the next section, these examples provide useful intuition.

Figure 4.8 contains segmentations of several objects, including a car, several lamp posts, a sign, and a trash can. The first column has the ground truth segmentation created with our interactive tool. The next column has the all points segmentation, with a radius $r = 2$. The next two columns have connected components segmentations, with radius $r = 2$ and spacing $s = .08$ and $.1$. The next two columns have our min-cut segmentation, with a static background radius $r = 2$ and $r = 4$. The last column has results of our method with an automatic background radius.

Because none of the example objects are isolated, the all points segmentation returns many background points. The connected components segmentation is more successful: the result for the car example is close to correct with $s = .1$, and the sign example is correct for $s = .08$. However, the spacing is difficult to adjust: both connected component segmentations under-segment the car while over-segmenting most of the other examples.

The min-cut segmentations with a static background have better results. For the sign example, both radii return the correct segmentation, and the first lamp post example is close to correct with both radii. For each of the example objects, one of the two settings of r returns the correct segmentation with the min-cut. But, for most examples, the wrong choice of r returns a drastic over- or under-segmentation. The min-cut method with automatic radius, on the other hand, is able to choose the correct background radius for these examples, returning correct segmentations.

Of course, no automatic segmentation algorithm is perfect. Some example failure cases of our automatic algorithm are shown in Figure 4.9. Our algorithm can fail in several scenarios. In (a), many noise points lie between a control box (on the right) and a light standard (on the left). The resulting high cut cost of the correct segmentation makes it difficult for our algorithm to automatically choose the correct (smaller) background radius, so both objects are returned. In (b), a car has a very weakly connected component, which is returned. The small cost of the cut of this component (relative to the correct segmentation of the entire car, which has noisy connections to the adjacent car), again makes it difficult to choose the correct background radius. In (c), a lamp post is very strongly connected to an adjacent roof. While an appropriate choice of background may return a better segmentation, more useful cues for this case include normal continuity and concavity.

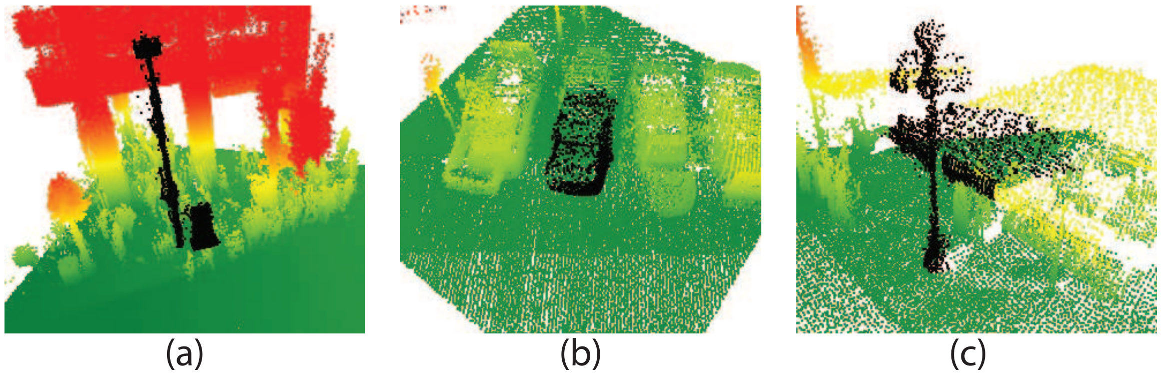


Figure 4.9: Example failures of the automatic segmentation algorithm. In (a) the control box (on the right) cannot be separated from a close light standard (on the left). In (b), only a part of a car is returned. In (c), a lamp post is not separated from a close roof.

Evaluation

Using the ground truth segmentations we created with our interactive segmentation tool, we are able to quantitatively evaluate the performance of our segmentation algorithm, and compare to the alternatives.

We gather statistics as follows. For each object of interest, we run a segmentation algorithm, and record its precision (ratio of correctly predicted foreground points to the total number of predicted foreground points) and recall (ratio of correctly predicted foreground points to the number of ground truth foreground points). A high precision indicates that most of the predicted foreground points are in the object, and a high recall indicates that most of the object points have been predicted to be in the foreground.

Table 4.2 contains the results, averaged first by object class and then overall, for the segmentation algorithms shown in Figure 4.8: all points, two settings of connected components, two settings of min-cut with static radius, and min-cut with an automatically chosen background radius. Some objects are easier to segment: parking meters are often isolated, so both connected component and min-cut algorithms perform well. Other objects, such as trash cans, are often close to background clutter, so the precision is lower. Min-cut algorithms are able to raise both precision and recall for trash cans relative to connected components. Likewise, min-cut algorithms improve performance significantly for cars and signs. Other objects, such as newspaper boxes (an example of one is shown in Figure 4.7) are very close to each other, so while our algorithm improves the precision, it remains low.

Overall, as expected, the all points algorithm has a relatively high recall at the cost of low precision. The two connected component algorithms have a higher precision, and the min-cut algorithms

		All Points		Con Comp		Con Comp		Min-Cut		Min-Cut		Min-Cut	
	# in	r = 2		r = 2		r = 2		r = 2		r = 4		auto r	
	Truth			s = .08		s = .1							
Class	Area	Pr	Re	Pr	Re	Pr	Re	Pr	Re	Pr	Re	Pr	Re
Short Post	338	13	99	89	99	86	99	93	98	82	99	92	99
Car	238	77	75	93	47	91	59	93	20	92	82	92	77
Lamp Post	146	60	99	82	95	79	97	89	96	86	99	89	98
Sign	96	36	100	73	74	68	97	84	98	73	100	83	100
Light Standard	58	68	93	84	92	83	92	92	86	91	92	91	92
Traffic Light	42	58	75	75	75	72	75	92	72	84	87	84	86
Newspaper Box	37	13	100	15	96	14	100	40	86	21	100	38	93
Tall Post	34	35	100	42	89	42	96	79	84	46	100	58	96
Fire Hydrant	20	36	100	81	89	81	95	89	100	82	100	88	100
Trash Can	19	17	100	48	93	43	94	57	100	54	100	60	100
Parking Meters	10	14	100	100	98	100	99	100	100	100	100	100	100
Traffic Control Box	7	19	100	82	96	79	99	79	100	68	100	80	100
Recycle Bins	7	46	100	71	94	64	99	92	99	80	100	92	100
Advertising Cylinder	6	70	100	79	83	79	83	97	100	89	100	96	100
Mailing Box	3	48	100	86	100	86	100	98	100	98	100	98	100
"A" - frame	2	59	100	70	50	69	100	87	100	69	100	86	100
All	1063	43	93	82	84	79	88	89	78	81	95	86	93

Table 4.2: Per-class precision/recall results of the segmentation algorithms (units for algorithm parameters r and s in meters).

improve on this performance. The min-cut algorithm with automatic radius has better performance that the two shown settings of the static radius version. This last point is more apparent in Figure 4.10, which shows the precision-recall curves resulting from running the above segmentation algorithms at several settings. Specifically, it shows the all-points algorithm at varying radii (blue), connected components at varying spacing (red), min-cut with varying static background (green), and min-cut with automatically chosen radius at varying thresholds (purple). Comparing the last two curves shows the improvement in performance made by automatically choosing the background radius.

4.4.3 Recognition

For the recognition experiment, we designate the north quarter of the truth area to be the training area, and the rest to be the test area. In this section, we present three experiments that evaluate the effect of some of our design choices on the final recognition results: first, we present the effect of different features, then of different classifiers, and, finally, we present the per-class results of all the stages of our algorithm.

To test how much each of the features add to the performance, we evaluate the recognition rates as more features are added. To evaluate the recognition, similarly to the localization evaluation, we

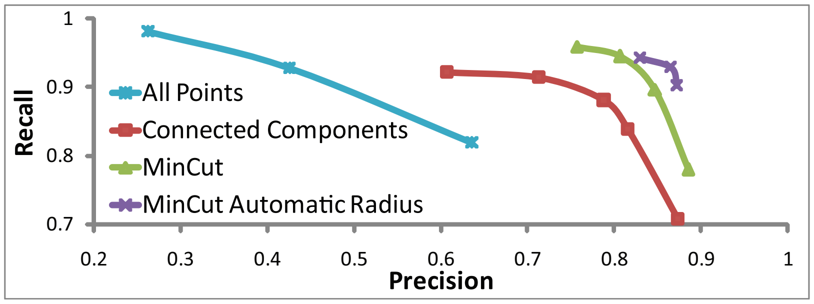


Figure 4.10: Precision-recall plots of our algorithm compared to several alternatives. The all points algorithm is shown in blue at varying radii. The connected components algorithm is shown in red with varying spacing. The min-cut algorithm is shown in green with varying statically chosen background radii. Finally, the min-cut algorithm with automatically chosen background radius is shown in purple with varying cut cost thresholds. The performance improves in the order of the algorithms presented.

	Precision			Recall	
Feature	# Predicted	Correct	(%)	Correct	(%)
Shape Features	568	313	(58)	298	(55)
+ Multiple Segs	591	336	(59)	314	(59)
+ Context	586	360	(64)	327	(61)

Table 4.3: Effect of features on recognition rates.

consider an automatically labeled location to be correct if there exists a truth object of the same type in range (1m), and we consider a truth object to have been found if our algorithm produces a location of the same label in range. Therefore, each classification experiment yields a precision/recall pair of values.

Because different classifiers are able to make more use of some features than others, to get a more robust measure of how much each feature set improves performance, we average the recognition results for the classifiers listed in Section 4.3.5 (and discussed in more detail below). For each classifier, we start with the shape features (computed on an all-point within 2m segmentation), then add shape features computed on multiple segmentations, and finally add context features. The results, shown in Table 4.3 describe, for each choice of features, how many non-background objects the algorithm predicted, how many of those and what percentage was correct, and how many and what percentage of the truth objects were found.

Shape features identify the easiest objects, with an average precision of 54% and recall of 55%. Adding multiple segmentations enhances the performance, and adding contextual features raises average the precision and recall rates to 64% and 61%, respectively. Note that the location algorithm is able to find 92% of the objects, which places a limit on the number of objects this stage can recognize.

	Precision			Recall	
Classifier	# Predicted	Correct	(%)	Correct	(%)
NN1	707	379	(54)	344	(64)
NN5	582	374	(64)	342	(63)
Random Forest	368	288	(78)	270	(50)
SVM	687	400	(58)	351	(65)

Table 4.4: Effect of classifiers on recognition rates.

Next, in Table 4.4 we present the differences in performance due to the choice of classifiers described in Section 4.3.5: NN1, NN5, Random Forest, and SVM. SVM performs comparably to the NN5, which outperforms NN1, and the Random Forest classifier has considerably higher precision rates at the cost of lower recall. Because of its higher recall rate, we use SVM for the subsequent experiments.

Finally, we present per-class results for all stages of our algorithm in Table 4.5. For each class (ordered by number of instances), we present: the number of objects in the truth area, the number and percent of truth objects found by the localization algorithm, precision and recall values of the segmentation algorithm (initialized at truth locations), the number of objects in the test area, the number of predictions made by our recognition algorithm, and the precision and recall rates of this algorithm. Note that the labels in the table do not include the special "Background" category; of the 6514 locations predicted in the test area, 5827 are classified as background, 96% of them correctly (in the sense that they do not have a true object in range).

From these results, recognition rates are clearly highest for object types with more examples. Looking closely at the classes with few training examples (lower rows of the table), we note that the the location and segmentation algorithms perform very well for these classes, and thus we conclude that the main bottlenecks in recognition performance are the feature extraction and/or classification stages. These results suggest that investigation of better shape descriptors, contextual cues, and/or classifiers that explicitly adapt to few training examples are suitable topics for follow-up work.

4.4.4 Large-scale recognition

In the final experiment, the 1000 or so truth objects (Figure 4.1a) were used to find and recognize 6698 objects in the remainder of the city (Figure 4.1b). The entire process took 46 hours on a 3GHz PC: 15 hours to pre-process the points (estimate ground and buildings); 6 hours to produce about 95,000 locations; 15 hours to produce segmentations; 6 hours to extract features; and 4 hours to classify using SVMs. While this experiment was run on a single PC, most of these steps

		Location		Segmentation		Recognition			
Class	# in Truth Area	# Found	(%)	Pr	Re	# in Test Area	# Predicted	Pr	Re
Short Post	338	328	(97)	92	99	116	131	79	91
Car	238	179	(75)	92	77	112	218	50	62
Lamp Post	146	146	(100)	89	98	98	132	70	86
Sign	96	96	(100)	83	100	60	71	58	65
Light Standard	58	57	(98)	91	92	37	51	45	62
Traffic Light	42	39	(93)	84	86	36	33	52	47
Newspaper Box	37	34	(92)	38	93	29	14	0	0
Tall Post	34	33	(97)	58	96	10	6	67	40
Fire Hydrant	20	17	(85)	88	100	14	10	30	21
Trash Can	19	18	(95)	60	100	15	14	57	40
Parking Meters	10	9	(90)	100	100	0	4	0	0
Traffic Control Box	7	7	(100)	80	100	5	0	0	0
Recycle Bins	7	7	(100)	92	100	3	1	0	0
Advertising Cylinder	6	6	(100)	96	100	3	0	0	0
Mailing Box	3	3	(100)	98	100	1	2	0	0
"A" - frame	2	2	(100)	86	100	0	0	0	0
All	1063	976	(92)	86	93	539	687	58	65

Table 4.5: Per-class results of the stages of our algorithm. Shown for each class are: the number of objects in the truth area, number and percent of truth objects found by the localization algorithm, precision and recall of segmentation, number of objects in the test area, the number of predictions made by the recognition algorithm, and the precision and recall rates of recognition.

are parallelizable. An example scene is shown in Figure 4.1c. Although we cannot quantitatively evaluate these results, visual confirmation suggests that they have similar recognition rates to those recorded in the truth area.

4.5 Conclusion

We described a system that recognizes small objects in city scans by locating, segmenting, describing, and classifying them. We described several potential approaches for each stage, and quantitatively evaluated their performance. Our system is able to recognize 65% of the objects in our test area.

One of our major design decisions was to perform location and segmentation before labeling objects. We believe that the results for these two stages validate this approach: we can locate most objects, and segment with a high degree of accuracy. Therefore, it makes sense to perform further processing at the level of point clouds representing potential objects, rather than at the level of individual points. While both location and segmentation algorithms have room for improvement, we believe that the direction of future work that would most benefit recognition rates lies in the creation of additional features. Our system would benefit from more discriminative shape features as well as additional contextual features.

Chapter 5

Discussion

5.1 Contributions

This thesis describes several advances in geometry analysis algorithms. In particular, we made the following research contributions:

Symmetry We developed a pipeline for making mesh processing algorithms "symmetry-aware", using large-scale symmetries to aid the processing of 3D meshes. Our pipeline can be used to emphasize the symmetries of a mesh, establish correspondences between symmetric features of a mesh, and decompose a mesh into symmetric parts and asymmetric residuals. We made technical contributions towards two of the main steps in this pipeline: a method for symmetrizing the geometry of an object, and a method for remeshing an object to have a symmetric triangulation. We offered several applications of this pipeline: modeling, beautification, attribute transfer, and simplification of approximately symmetric surfaces. This work was published in [44].

Part Decomposition We conducted several investigations into part decomposition of 3D objects. We introduced a hierarchical segmentation method based on an error similar to normalized cuts. We then extended this algorithm to consistently segment a set of meshes. We showed how our method of consistent segmentations can be adapted to the more specific applications of symmetric segmentation and segmentation transfer. This work was published in [41].

Then, we described a probabilistic equivalent of mesh segmentation, which we called a "partition function", that aims to estimate the likelihood that a given mesh edge is on a segmentation boundary. We showed several ways of computing this structure, and demonstrated its robustness to noise, tessellation, and pose and intra-class shape variation. We showed the utility of the partition function for mesh visualization, segmentation, deformation, and registration. This work was presented in [40].

Scene Analysis Finally, we developed a system for object recognition in 3D scenes, and tested it on a large point cloud representing a city. We made technical contributions towards three key steps of our system: localizing objects, segmenting them from the background, and extracting features that describe them. We also made contributions in the evaluation of the system: we performed quantitative evaluation on a point cloud consisting of about 100 million points, with about 1000 objects of interest belonging to 16 classes. We evaluated our system as a whole, as well as each individual step, trying several alternatives for each component. The overall system was described in [43], and the segmentation stage was described in [42].

5.2 Future Work

We presented several contributions towards shape analysis. However, as is often the case, the results achieved by fully automatic algorithms are not perfect (in many cases, far from perfect). An important, and often productive, research strategy is to incrementally build on these methods to produce better results. There are many avenues to incrementally improve the work presented in this thesis, some of which have been delineated in the appropriate sections. However, beyond such incremental research, we suggest two research directions that may more dramatically bridge the gap between current and desired results. The first strategy we would suggest is an effort to combine existing tools. The second involves a greater focus on interactive methods.

There are many proposed tools that focus on individual aspects of shape properties: e.g., concavities, symmetries, and spectral properties. However, shapes are composed of a variety of these cues, and more work needs to be done on integrating these concepts. Within the context of detection, for example, planar regions may help identify cars, symmetry properties may help with light posts, and spectral properties with trees. At several points in the thesis, we moved in this direction: we combined some disparate properties (such as connected components, concavity, and segment boundary lengths) in our hierarchical segmentation algorithm in Section 3.3.3, and aggregated features describing various properties (volume, spin images, contextual information) in the feature extraction stage of our detection framework in Section 4.3.4. However, we did this in an ad-hoc manner, and a great deal of work remains to unify these properties and properly balance them.

Just as with properties, there is a great deal of work tackling specific problems in shape analysis: for example, segmentation, classification, and registration. However, these are not unrelated problems: segmenting an object may help with both classification and registration, and knowing an object's class certainly helps with segmentation and registration, and so on. In our object detection framework, for example, we segmented objects from background and then classified them. Our method would likely benefit from feedback from classification to correct segmentations. While this is a specific example, shape analysis research in general may benefit from combining methods developed for more specific problems.

However, despite the best efforts, automatically generated results will often continue to fall short of necessary accuracy, and manual intervention will be required. For example, in many scenarios in real-life computer graphics applications (virtual worlds, games, movies, etc.), close to perfect results are needed. In industry, it is standard practice to insert a quality assurance (QA) mechanism to verify the results of any automatic shape analysis (or processing) stage. It would be fruitful to reconcile research with this reality, and instead of aiming for fully automatic systems that are inevitably error-prone, develop interactive systems that cleverly leverage human operator time.

There are several ways in which human interaction may be helpful. For data of limited size, such as during analysis a single model, the human can control the entire analysis (for example, in the system we describe to interactively segment point clouds in Section 4.3.3). For problems of larger scale, it has been typical to use human input as training data, as we do in our object detection system. However, most systems (such as ours), have a training phase, followed by the analysis, and do not consider the problem of QA: an application that requires 99% accuracy does not benefit from a fully automatic algorithm that delivers 90% accuracy, if a human is required to verify or adjust every result to ensure sufficient quality.

To address this concern, we suggest the development of interactive systems that are responsive to performance demands, in which the QA is folded into the system. This proposition is in the direction of online learning in machine learning research: our suggested systems would identify possibly incorrect results, query the user to aid the algorithm, adjust results, and iterate until a sufficient level of estimated accuracy is reached. Most of the problems addressed in this thesis would benefit from such a framework. For symmetry-aware processing, for example, the system might detect regions of high deformation or local shape mis-match, and query the user to adjust or confirm the automatically-generated mapping. For object detection, the user would be asked to re-classify those objects whose classification is uncertain, and the system will learn from that.

With interactive systems that integrate multiple shape analysis cues, it should be possible to create semi-automatic but time-efficient solutions to the problems addressed in this thesis.

Bibliography

[1] Open street map - the free wiki world map, 2009. http://www.openstreetmap.org.

[2] Agathos, A., Pratikakis, I., Perantonis, S., Sapidis, N., and Azariadis, P. 3D mesh segmentation methodologies for CAD applications. *Computer-Aided Design & Applications 4*, 6 (2007), 827–841.

[3] Ahn, M., Lee, S., and Seidel, H. Connectivity transformation for mesh metamorphosis. In *Eurographics/ACM SIGGRAPH symposium on Geometry processing* (2004), pp. 75–82.

[4] Alexa, M. Merging polyhedral shapes with scattered features. *The Visual Computer 16*, 1 (2000), 26–37.

[5] Alias Abdul-Rahman, S. Z., and Coors, V. Improving the realism of existing 3d city models. *Sensors 8*, 11 (2008), 7323–7343.

[6] Allen, B., Curless, B., and Popović, Z. The space of human body shapes: reconstruction and parameterization from range scans. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers* (New York, NY, USA, 2003), ACM, pp. 587–594.

[7] Anguelov, D., Taskar, B., Chatalbashev, V., Koller, D., Gupta, D., Heitz, G., and Ng, A. Discriminative learning of markov random fields for segmentation of 3d scan data. In *CVPR* (2005), pp. 169–176.

[8] Ankerst, M., Kastenmüller, G., Kriegel, H.-P., and Seidl, T. 3d shape histograms for similarity search and classification in spatial databases. In *SSD '99: Proceedings of the 6th International Symposium on Advances in Spatial Databases* (London, UK, 1999), Springer-Verlag, pp. 207–226.

[9] Antini, G., Berretti, S., Del Bimbo, A., and Pala, P. 3D mesh partitioning for retrieval by parts applications. In *Multimedia and Expo* (2005).

[10] Attene, M., Falcidieno, B., and Spagnuolo, M. Hierarchical mesh segmentation based on fitting primitives. *The Visual Computer 22*, 3 (2006), 181–193.

[11] Attene, M., Katz, S., Mortara, M., Patane, G., Spagnuolo, M., and Tal, A. Mesh segmentation - a comparative study. In *SMI '06: Proceedings of the IEEE International Conference on Shape Modeling and Applications 2006 (SMI'06)* (Washington, DC, USA, 2006), IEEE Computer Society, p. 7.

[12] Attene, M., Robbiano, F., Spagnuolo, M., and Falcidieno, B. Semantic annotation of 3D surface meshes based on feature characterization. *Lecture Notes in Computer Science 4816* (2007), 126–139.

[13] Au, O. K.-C., Tai, C.-L., Chu, H.-K., Cohen-Or, D., and Lee, T.-Y. Skeleton extraction by mesh contraction. *ACM Trans. Graph. 27*, 3 (2008), 1–10.

[14] Audette, M. A., Ferrie, F. P., and Peters, T. M. An algorithmic overview of surface registration techniques for medical imaging. *Medical Image Analysis 4*, 3 (2000), 201–217.

[15] Belongie, S., Malik, J., and Puzicha, J. Shape context: A new descriptor for shape matching and object recognition. In *NIPS* (2000), pp. 831–837.

[16] Besl, P. J., and McKay, N. D. A method for registration of 3-D shapes. *IEEE Trans. PAMI 14*, 2 (1992), 239–256.

[17] Botsch, M., Pauly, M., Gross, M., and Kobbelt, L. Primo: coupled prisms for intuitive surface modeling. In *SGP '06: Proceedings of the fourth Eurographics symposium on Geometry processing* (Aire-la-Ville, Switzerland, Switzerland, 2006), Eurographics Association, pp. 11–20.

[18] Boykov, Y., and Funka-Lea, G. Graph cuts and efficient n-d image segmentation. *IJCV 70*, 2 (2006), 109–131.

[19] Bredif, M., Boldo, D., Pierrot-Deseilligny, M., and Maitre, H. 3d building reconstruction with parametric roof superstructures. *International Conference on Image Processing, 2007 2* (2007), 537–540.

[20] C. Frueh, S. Jain, A. Z. Data processing algorithms for generating textured 3d building facade meshes from laser scans and camera images. *International Journal of Computer Vision, 2005 61*, 2 (February 2005), 159–184.

[21] Chazelle, B., Dobkin, D., Shourhura, N., and Tal, A. Strategies for polyhedral surface decomposition: An experimental study. *Computational Geometry: Theory and Applications 7*, 4-5 (1997), 327–342.

[22] Chen, D.-Y., Tian, X.-P., Shen, Y.-T., and Ouhyoung, M. On visual similarity based 3d model retrieval. *Comput. Graph. Forum 22*, 3 (2003), 223–232.

[23] Chen, J., and Chen, B. Architectural modeling from sparsely scanned range data. *IJCV 78*, 2-3 (2008), 223–236.

[24] Chen, X., Golovinskiy, A., , and Funkhouser, T. A benchmark for 3D mesh segmentation. *ACM Transactions on Graphics (Proc. SIGGRAPH) 28*, 3 (Aug. 2009).

[25] Coifman, R. R., Lafon, S., Lee, A. B., Maggioni, M., Nadler, B., Warner, F., and Zucker, S. W. Geometric diffusions as a tool for harmonic analysis and structure definition of data: Multiscale methods. *Proceedings of the National Academy of Sciences of the United States of America 102*, 21 (May 2005), 7432–7437.

[26] Dong, S., Bremer, P.-T., Garland, M., Pascucci, V., and Hart, J. C. Spectral surface quadrangulation. *ACM Trans. Graph. 25*, 3 (2006), 1057–1066.

[27] Dorninger, P., and Pfeifer, N. A comprehensive automated 3d approach for building extraction, reconstruction, and regularization from airborne laser scanning point clouds. *Sensors 8*, 11 (2008), 7323–7343.

[28] Edmonds, J., and Karp, R. M. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM 19*, 2 (1972).

[29] Fellbaum, C., Ed. *WordNet: An Electronic Lexical Database (Language, Speech, and Communication).* The MIT Press, May 1998.

[30] Ferguson, R. W. Modeling orientation effects in symmetry detection: The role of visual structure. In *Proc. Conf. Cognitive Science Society* (2000).

[31] Frome, A., Huber, D., Kolluri, R., Bulow, T., and Malik, J. Recognizing objects in range data using regional point descriptors. In *ECCV* (May 2004).

[32] Fukunaga, K., and Hostetler, L. The estimation of the gradient of a density function, with applications in pattern recognition. *Information Theory, IEEE Transactions on 21*, 1 (1975), 32–40.

[33] Funkhouser, T., Kazhdan, M., Shilane, P., Min, P., Kiefer, W., Tal, A., Rusinkiewicz, S., and Dobkin, D. Modeling by example. *ACM Transactions on Graphics (Siggraph 2004)* (Aug. 2004).

[34] Gal, R., and Cohen-Or, D. Salient geometric features for partial shape matching and similarity. In *ACM Transaction on Graphics* (2005).

[35] Garland, M., and Heckbert, P. S. Surface simplification using quadric error metrics. In *Proceedings of SIGGRAPH 1997* (Aug. 1997), Computer Graphics Proceedings, Annual Conference Series, pp. 209–216.

[36] Garland, M., Willmott, A., and Heckbert, P. S. Hierarchical face clustering on polygonal surfaces. In *SI3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics* (New York, NY, USA, 2001), ACM Press, pp. 49–58.

[37] Gdalyahu, Y., Weinshall, D., and Werman, M. Self-organization in vision: Stochastic clustering for image segmentation, perceptual grouping, and image database organization. *IEEE Transactions on Pattern Analysis and Machine Intelligence 23*, 10 (2001), 1053–1074.

[38] Gelfand, N., and Guibas, L. Shape segmentation using local slippage analysis. In *Symposium on Geometry Processing* (2004), pp. 214–223.

[39] Giorgi, D., Biasotti, S., and Paraboschi, L. SHape REtrieval Contest 2007: Watertight models track. In *SHREC competition* (2007).

[40] Golovinskiy, A., and Funkhouser, T. Randomized cuts for 3D mesh analysis. *ACM Transactions on Graphics (Proc. SIGGRAPH ASIA) 27*, 5 (Dec. 2008).

[41] Golovinskiy, A., and Funkhouser, T. Consistent segmentation of 3D models. *Computers and Graphics (Shape Modeling International 09) 33*, 3 (June 2009), 262–269.

[42] Golovinskiy, A., and Funkhouser, T. Min-cut based segmentation of point clouds. In *IEEE Workshop on Search in 3D and Video (S3DV) at ICCV* (2009).

[43] Golovinskiy, A., Kim, V. G., and Funkhouser, T. Shape-based recognition of 3D point clouds in urban environments. *International Conference on Computer Vision (ICCV)* (2009).

[44] Golovinskiy, A., Podolak, J., and Funkhouser, T. Symmetry-aware mesh processing. *Mathematics of Surfaces 2009 (invited paper)* (2009).

[45] Hanke, S., Ottmann, T., and Schuierer, S. The edge-flipping distance of triangulations. *Journal of Universal Computer Science 2*, 8 (1996), 570–579.

[46] Hilaga, M., Shinagawa, Y., Kohmura, T., and Kunii, T. L. Topology matching for fully automatic similarity estimation of 3D shapes. In *Proceedings of SIGGRAPH 2001* (August 2001), Computer Graphics Proceedings, Annual Conference Series, pp. 203–212.

[47] Igarashi, T., Matsuoka, S., and Tanaka, H. Teddy: A sketching interface for 3D freeform design. In *Computer Graphics (Siggraph 1999)* (1999), Addison Wesley Longman, pp. 409–416.

[48] Inoue, K., Takayuki, I., Atsushi, Y., Tomotake, F., and Kenji, S. Face clustering of a large-scale cad model for surface mesh generation. *Computer-Aided Design 33* (2001), 251–261.

[49] Jaakkola, A., Hyypp, J., Hyypp, H., and Kukko, A. Retrieval algorithms for road surface modelling using laser-based mobile mapping. *Sensors 8*, 9 (2008), 5238–5249.

[50] Jain, V., and Zhang, H. Robust 3d shape correspondence in the spectral domain. In *SMI '06: Proceedings of the IEEE International Conference on Shape Modeling and Applications 2006* (Washington, DC, USA, 2006), IEEE Computer Society, p. 19.

[51] Johnson, A. E., and Hebert, M. Using spin images for efficient object recognition in cluttered 3d scenes. *IEEE PAMI 21* (1999), 433–449.

[52] Karger, D. R., and Stein, C. A new approach to the minimum cut problem. *Journal of the ACM 43*, 4 (1996), 601–640.

[53] Karni, Z., and Gotsman, C. Spectral compression of mesh geometry. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2000), ACM Press/Addison-Wesley Publishing Co., pp. 279–286.

[54] Katz, S., Leifman, G., and Tal, A. Mesh segmentation using feature point and core extraction. *The Visual Computer (Pacific Graphics) 21*, 8-10 (October 2005), 649–658.

[55] Katz, S., and Tal, A. Hierarchical mesh decomposition using fuzzy clustering and cuts. *ACM Transactions on Graphics (TOG) 22*, 3 (2003), 954–961.

[56] Kazhdan, M., Chazelle, B., Dobkin, D., Funkhouser, T., and Rusinkiewicz, S. A reflective symmetry descriptor for 3D models. *Algorithmica 38*, 1 (Oct. 2003).

[57] Kazhdan, M., Funkhouser, T., and Rusinkiewicz, S. Rotation invariant spherical harmonic representation of 3D shape descriptors. In *Symposium on Geometry Processing* (June 2003).

[58] Kraevoy, V., Julius, D., and Sheffer, A. Shuffler: Modeling with interchangeable parts. *Visual Computer Journal* (2007).

[59] Kraevoy, V., and Sheffer, A. Cross-parameterization and compatible remeshing of 3d models. *ACM Trans. Graph. 23*, 3 (2004), 861–869.

[60] Lafarge, F., Descombes, X., Zerubia, J., and Pierrot-Deseilligny, M. Building reconstruction from a single dem. *CVPR* (June 2008), 1–8.

[61] Lai, Y., Hu, S., Martin, R., and Rosin, P. Fast mesh segmentation using random walks. In *ACM Symposium on Solid and Physical Modeling* (2008).

[62] Lai, Y., Hu, S., Martin, R., and Rosin, P. Rapid and effective segmentation of 3d models using random walks. In *Computer Aided Geometric Design* (2009).

[63] Lalonde, J.-F., Unnikrishnan, R., Vandapel, N., and Hebert, M. Scale selection for classification of point-sampled 3-d surfaces. In *Fifth International Conference on 3-D Digital Imaging and Modeling (3DIM 2005)* (June 2005), pp. 285 – 292.

[64] Lalonde, J.-F., Vandapel, N., Huber, D., and Hebert, M. Natural terrain classification using three-dimensional ladar data for ground robot mobility. *Journal of Field Robotics 23*, 1 (November 2006), 839 – 861.

[65] Lee, A., Dobkin, D., Sweldens, W., and Schroeder, P. Multiresolution mesh morphing. *ACM Transactions on Graphics (Proc. SIGGRAPH 2001)* (1999), 343–350.

[66] Lee, C. H., Varshney, A., and Jacobs, D. W. Mesh saliency. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers* (New York, NY, USA, 2005), ACM, pp. 659–666.

[67] Lee, Y., Lee, S., Shamir, A., Cohen-Or, D., and Seidel, H. Mesh scissoring with minima rule and part salience. *Computer-Aided Geometric Design 22*, 5 (2005), 444–465.

[68] Leibe, B., Leonardis, A., and Schiele, B. Combined object categorization and segmentation with an implicit shape model. In *In ECCV workshop on statistical learning in computer vision* (2004), pp. 17–32.

[69] Li, X., Woon, T. W., Tan, T. S., and Huang, Z. Decomposing polygon meshes for interactive applications. In *Proc. Symposium on Interactive 3D Graphics* (Research Triangle Park, NC, March 2001), ACM, pp. 35–42.

[70] Lin, H., Liao, H., and Lin, J. Visual salience-guided mesh decomposition. In *IEEE Int. Workshop on Multimedia Signal Processing* (2004), pp. 331–334.

[71] Lipman, Y., Sorkine, O., Levin, D., and Cohen-Or, D. Linear rotation-invariant coordinates for meshes. In *Proceedings of ACM SIGGRAPH 2005* (2005), ACM Press, pp. 479–487.

[72] Lipman, Y., Sorkine, O., Levin, D., and Cohen-Or, D. Linear rotation-invariant coordinates for meshes. *ACM Trans. Graph. 24*, 3 (2005), 479–487.

[73] Liu, R., and Zhang, H. Segmentation of 3d meshes through spectral clustering. In *PG '04: Proceedings of the Computer Graphics and Applications, 12th Pacific Conference* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 298–305.

[74] Liu, R. F., Zhang, H., Shamir, A., and Cohen-Or, D. A part-aware surface metric for shape analysis. *Computer Graphics Forum, (Proceedings Eurographics 2009) 28*, 2 (2009).

[75] M. Carlberg, J. Andrews, P. G., and Zakhor, A. Fast surface reconstruction and segmentation with ground-based and airborne lidar range data. In *3DPVT* (June 2008).

[76] Mangan, A., and Whitaker, R. Partitioning 3D surface meshes using watershed segmentation. *IEEE Transactions on Visualization and Computer Graphics 5*, 4 (1999), 308–321.

[77] Martinet, A., Soler, C., Holzschuch, N., and Sillion, F. Accurately detecting symmetries of 3D shapes. Tech. Rep. RR-5692, INRIA, September 2005.

[78] Matei, B., Shan, Y., Sawhney, H., Tan, Y., Kumar, R., Huber, D., and Hebert, M. Rapid object indexing using locality sensitive hashing and joint 3d-signature space estimation. 1111 – 1126.

[79] Matei, B. C., Tan, Y., Sawhney, H. S., and Kumar, R. Rapid and scalable 3D object recognition using LIDAR data. In *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series* (June 2006), vol. 6234 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series.*

[80] Miller, G. Efficient algorithms for local and global accessibility shading. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1994), ACM, pp. 319–326.

[81] Mills, B. I., Langbein, F. C., Marshall, A. D., and Martin, R. R. Approximate symmetry detection for reverse engineering. In *SMA '01: Proceedings of the sixth ACM symposium on Solid modeling and applications* (New York, NY, USA, 2001), ACM Press, pp. 241–248.

[82] Mitra, N. J., Guibas, L., and Pauly, M. Symmetrization. In *ACM Transactions on Graphics* (2007), vol. 26, pp. #63, 1–8.

[83] Mitra, N. J., Guibas, L. J., and Pauly, M. Partial and approximate symmetry detection for 3d geometry. *ACM Trans. Graph. 25*, 3 (2006), 560–568.

[84] Mortara, M., Patan, G., Spagnuolo, M., Falcidieno, B., and Rossignac, J. Plumber: a method for a multi-scale decomposition of 3D shapes into tubular primitives and bodies. In *ACM Symposium on Solid Modeling and Applications* (2004).

[85] Mortara, M., Patane, G., Spagnuolo, M., Falcidieno, B., and Rossignac, J. Blowing bubbles for multi-scale analysis and decomposition of triangle meshes. *Algorithmica 38*, 1 (2003), 227–248.

[86] Neptec. Wright state 100, 2009. http://www.daytaohio.com/Wright_State100.php.

[87] Ortner, M., Descombes, X., and Zerubia, J. Building outline extraction from digital elevation models using marked point processes. *Int. J. Comput. Vision 72*, 2 (2007), 107–132.

[88] Osada, R., Funkhouser, T., Chazelle, B., and Dobkin, D. Shape distributions. *ACM Transactions on Graphics 21*, 4 (Oct. 2002), 807–832.

[89] P. Cignoni, C. R., and Scopigno, R. Metro: measuring error on simplified surfaces. *Computer Graphics Forum 17*, 2 (June 1998), 167–174.

[90] Pauly, M., Mitra, N. J., Giesen, J., Gross, M., and Guibas, L. Example-based 3D scan completion. In *Symposium on Geometry Processing* (2005), pp. 23–32.

[91] Pauly, M., Mitra, N. J., Wallner, J., Pottmann, H., and Guibas, L. Discovering structural regularity in 3D geometry. *ACM Transactions on Graphics 27*, 3 (2008), #43, 1–11.

[92] Podolak, J., Shilane, P., Golovinskiy, A., Rusinkiewicz, S., and Funkhouser, T. A planar-reflective symmetry transform for 3D shapes. *ACM Transactions on Graphics (Proc. SIGGRAPH) 25*, 3 (July 2006).

[93] Popa, T., Julius, D., and Sheffer, A. Material-aware mesh deformations. In *SMI '06: Proceedings of the IEEE International Conference on Shape Modeling and Applications 2006* (Washington, DC, USA, 2006), IEEE Computer Society, p. 22.

[94] Praun, E., Sweldens, W., and Schroeder, P. Consistent mesh parameterizations. *ACM Transactions on Graphics (Proc. SIGGRAPH 2001)* (August 2001), 179–184.

[95] Rabbani, T., van den Heuvel, F., and Vosselmann, G. Segmentation of point clouds using smoothness constraint. In *IEVM06* (2006).

[96] Ren, X., and Malik, J. Learning a classification model for segmentation. In *In Proc. 9th Int. Conf. Computer Vision* (2003), pp. 10–17.

[97] Robbiano, F., Attene, M., Spagnuolo, M., and Falcidieno, B. Part-based annotation of virtual 3d shapes. *cw 0* (2007), 427–436.

[98] Rother, C., Minka, T., Blake, A., and Kolmogorov, V. Cosegmentation of image pairs by histogram matching - incorporating a global constraint into mrfs. In *CVPR '06: Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 993–1000.

[99] Rustamov, R., Lipman, Y., and Funkhouser, T. Interior distance using barycentric coordinates. *Computer Graphics Forum (Symposium on Geometry Processing) 28*, 5 (July 2009).

[100] Saupe, D., and Vranic, D. V. 3d model retrieval with spherical harmonics and moments. In *Proceedings of the 23rd DAGM-Symposium on Pattern Recognition* (London, UK, 2001), Springer-Verlag, pp. 392–397.

[101] Schreiner, J., Asirvatham, A., Praun, E., and Hoppe, H. Inter-surface mapping. *ACM Transactions on Graphics (Proc. SIGGRAPH 2004) 23*, 3 (2004), 870–877.

[102] Shalom, S., Shapira, L., Shamir, A., and Cohen-Or, D. Part analogies in sets of objects. *Eurographics Workshop on 3D Object Retrieval* (2008).

[103] Shamir, A. Segmentation and shape extraction of 3d boundary meshes (state-of-the-art report). In *Eurographics* (2006), pp. 137–149.

[104] Shapira, L., Shamir, A., and Cohen-Or, D. Consistent mesh partitioning and skeletonisation using the shape diameter function. *Vis. Comput. 24*, 4 (2008), 249–259.

[105] Shi, J., and Malik, J. Normalized cuts and image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence 22*, 8 (2000), 888–905.

[106] Shlafman, S., Tal, A., and Katz, S. Metamorphosis of polyhedral surfaces using decomposition. In *Eurographics 2002* (September 2002), pp. 219–228.

[107] Simari, P., Kalogerakis, E., and Singh, K. Folding meshes: Hierarchical mesh segmentation based on planar symmetry. In *Proceedings of the Symposium on Geometry Processing (SGP '06)* (June 2006), pp. 111–119.

[108] Sorkine, O., Cohen-Or, D., Lipman, Y., Alexa, M., Rössl, C., and Seidel, H.-P. Laplacian surface editing. In *SGP '04: Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing* (New York, NY, USA, 2004), ACM, pp. 175–184.

[109] Sumner, R. W., and Popović, J. Deformation transfer for triangle meshes. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers* (New York, NY, USA, 2004), ACM Press, pp. 399–405.

[110] Suter, D., and Lim, E. Conditional random field for 3d point clouds with adaptive data reduction. *Cyberworlds* (2007).

[111] Terzopoulos, D., Witkin, A., and Kass, M. Symmetry-seeking models and 3D object reconstruction. 221–221.

[112] Thrun, S., and Wegbreit, B. Shape from symmetry. In *Proceedings of the International Conference on Computer Vision (ICCV)* (Bejing, China, 2005), IEEE.

[113] Thrun, S., and Wegbreit, B. Shape from symmetry. In *Proceedings of the International Conference on Computer Vision (ICCV)* (Bejing, China, 2005), IEEE.

[114] Toshev, A., Shi, J., and Daniilidis, K. Image matching via saliency region correspondences. *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on 0* (2007), 1–8.

[115] Unnikrishnan, R., and Hebert, M. Robust extraction of multiple structures from non-uniformly sampled data. In *IROS* (October 2003), vol. 2, pp. 1322–29.

[116] Witten, I. H., and Frank, E. *Data Mining: Practical Machine Learning Tools and Techniques*, 2 ed. Morgan Kaufmann, 2005.

[117] Wolf, D. F., Sukhatme, G. S., Fox, D., and Burgard, W. Autonomous terrain mapping and classification using hidden markov models. In *IEEE ICRA* (April 2005), pp. 2038–2043.

[118] Wu, K., and Levine, M. 3D part segmetnation using simulationed electrical charge distributions. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI) 19*, 11 (1997), 1223–1235.

[119] Xu, H., Gossett, N., and Chen, B. Knowledge and heuristic-based modeling of laser-scanned trees. *ACM Trans. Graph. 26*, 4 (2007), 19.

[120] Yoshizawa, S., Belyaev, A., and Seidel, H. Smoothing by example: Mesh denoising by averaging with similarity-based weights. In *Shape Modeling International* (June 2006), pp. 38–44.

[121] You, S., Hu, J., Neumann, U., and Fox, P. Urban site modeling from lidar. 579 – 588.

[122] Yu, S. X., and Shi, J. Multiclass spectral clustering. In *International Conference on Computer Vision* (2003), pp. 313–319.

[123] Yunsheng Wang, H. W., and Koch, B. A lidar point cloud based procedure for vertical canopy structure analysis and 3d single tree modelling in forest. *Sensors 8* (2008), 1424–8220.

[124] Zabrodsky, H., Peleg, S., and Avnir, D. Completion of occluded shapes using symmetry. In *Proc. CVPR* (1993), pp. 678–679.

[125] Zabrodsky, H., Peleg, S., and Avnir, D. Symmetry as a continuous feature. *Trans. PAMI 17*, 12 (1995), 1154–1166.

[126] Zabrodsky, H., and Weinshall, D. Using bilateral symmetry to improve 3D reconstruction from image sequences. *Comput. Vis. Image Underst. 67*, 1 (1997), 48–57.

[127] Zhang, H., van Kaick, O., and Dyer, R. Spectral methods for mesh processing and analysis. In *Proc. of Eurographics State-of-the-art Report* (2007), pp. 1–22.

[128] Zhang, H., van Kaick, O., and Dyer, R. Spectral methods for mesh processing and analysis. In *Eurographics State of the Art Report* (2007).

[129] Zhukov, S., Inoes, A., and Kronin, G. An ambient light illumination model. In *Rendering Techniques* (1998), pp. 45–56.

[130] Zuckerberger, E., Tal, A., and Shlafman, S. Polyhedral surface decomposition with applications. *Computers & Graphics 26*, 5 (2002), 733–743.