MODFLOWS: METHODS FOR STUDYING AND MANAGING MESH EDITING WORKFLOWS

A Thesis

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Doctor of Philosophy

in

Computer Science

by

Jonathan D. Denning

DARTMOUTH COLLEGE

Hanover, New Hampshire

May 2014

Examining Committee:

Andrew T. Campbell, Chair

Devin J. Balkcom

Thomas H. Cormen

Marco Fratarcangeli

Fabio Pellacini

F. Jon Kull, Ph.D.
Dean of Graduate Studies

Abstract

At the heart of computer games and computer generated films lies 3D content creation. A student wanting to learn how to create and edit 3D meshes can quickly find thousands of videos explaining the workflow process. These videos are a popular medium due to a simple setup that minimally interrupts the artist's workflow, but video recordings can be quite challenging to watch. Typical mesh editing sessions involve several hours of work and thousands of operations, which means the video recording can be too long to stay interesting if played back at real-time speed or lose too much information when sped up. Moreover, regardless of the playback speed, a high-level overview is quite difficult to construct from long editing sessions.

In this thesis, we present our research into methods for studying how artists create and edit polygonal models and for helping manage collaborative work. We start by describing two approaches to automatically summarizing long editing workflows to provide a high-level overview as well as details on demand. The summarized results are presented in an interactive viewer with many features, including overlaying visual annotations to indicate the artist's actions, coloring regions to indicate strength of change, and filtering the workflow to specific 3D regions of interest. We evaluate the robustness of our two approaches by testing against a variety of workflows, holding a small case study, and asking artists for feedback.

Next we describe a way to construct a plausible and intuitive low-level workflow that turns one of two given meshes into the second by building mesh correspondences. Analogous to text version control tools, we visualize the mesh changes in a two-way, three-way, or sequence diff, and we demonstrate how to merge independent edits of a single original mesh, handling conflicts in a way that preserves the artists' original intentions.

We then discuss methods of comparing multiple artists performing similar mesh editing tasks. We build intra- and inter-correspondences, compute pairwise edit distances, and then visualize the distances as a heat map or by embedding into 3D space. We evaluate our methods by asking a professional artist and instructor for feedback.

Finally, we discuss possible future directions for this research.

Acknowledgements

To my God and my Lord and Savior Jesus, I thank you for grace and all your blessings. Without you as my foundation and refuge, surely I would have wandered far and wide. I pray that my work is pleasing in your sight and brings glory to your name. I look forward to discovering further your will for me and my family.

To my wife, I thank you for walking along with me through this wild adventure. I cannot express how grateful I am to have you at my side, for supporting me and our family, and for praying for me. I love you!

To my parents, I thank you for your unending support and encouragement and for the many books. To my cousin Scott, thank you for introducing me to computer programming so many years ago! To my dear friends Nick and Kirby, thank you for the long days and nights of coding and the programming contests!

To my advisor Fabio, thank you for your patience, guidance, perspective, motivation, and direction.

I would like to thank my committee members, Fabio Pellacini, Andrew Campbell, Devin Balkcom, Thomas Cormen, and Marco Fratarcangeli and my coauthors and collaborators, Brandon Kerr, Daniel An, Xin Tong, and Jiawei Ou. Their constructive comments and insightful questions have significantly improved the quality of my own questions as well as my work.

I would also like to thank the authors of the meshes and workflows used, especially Jonathan Williamson, Roberto Roch, Pablo Vazquez, Andreas Goralczyk, and those of Blender Foundation/Institute [50, 31, 12, 78, 76, 70, 8, 33, 42, 81, 58]; the original authors of tutorials used [24, 27, 37, 73, 80]; and the authors of the matching algorithms for providing source code and support.

This work was partially supported by NSF (CNS-070820, CCF-0746117), Intel, and the Sloan Foundation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Digital 3D content creation is a rapidly growing and thriving community. Not only are large production houses such as Pixar, Dreamworks Animation, and Blizzard built around generating 3D content, but content generation is central to much smaller productions and even home-based efforts, from realizing computer generated films such as Sintel [10], to adding 3D assets into live-action movies such as in Tears of Steel [11], to 3D-printing user-created objects such as found at Thingiverse [51]. Further, many modern video game engines now ship with sandbox tools, allowing users to create novel content and gaming experiences. Despite the surge in generation of 3D content creation, methods for understanding and managing these workflows are still ad-hoc, done by hand, or simply non-existent.

Static document- and video-based tutorials have flooded the internet to augment or even replace traditional "over-the-shoulder" master-apprentice interactive education. A tutorial amortizes the author's efforts through record-once/play-many and therefore reach far more students, but tutorials comes at a cost. Tutorials remove most, if not all, of the interactivity and ability to tailor the education to fit the students' needs that the traditional one-on-one or one-to-few settings can provide. Helpful annotations, alternative views (especially important with 3D data), high-level summaries, and low-level details are ultimately frozen at the time of creation. The author must spend considerable time planning out, practicing, and editing the educational material to work within the medium's limitations in order to produce an effective and concise tutorial that suits the needs of a broad audience.

Text version control is an indispensable tool for programmers, from a single coder to a large and geographically-disbursed group. Version control systems such as Subversion [1] or git [77] are often used to manage file versions and file syncing. For versioning, these systems provide text diff and merge tools to view changes to text and source code files and to combine independent work, which is crucial for collaborative editing. However analogous tools for binary data have been largely missing. Typically version control systems treat binary files, such as polygonal meshes and images, as blackboxes of information, agnostic to any underlying structure. This means that changes to these binary files cannot be directly viewed beyond a before-and-after, and two independent versions of a file, even with trivial changes, are marked as conflicting and require manual inspection and conflict resolution, leading to lost time and work. Recent research has worked to solve this but requires full instrumentation of the editing software.

Informative visualizations are key to understanding the structure and details of a workflow, but comparing two or more workflows gives insight into the quality and variety of them. To our knowledge, only recently has there been research done on this topic. Kong et al. [41] allow users to compare image editing workflows by before-and-after images or viewing the edit operations directly. Using a *union graph* on the edit operation names and their parameter settings, they compare and contrast two similar workflows. They focused on short workflows, between 5 and 30 steps, which is typical for short image manipulation tasks. However the problem is still open for comparing mesh editing workflows which range to several thousand operations.

In this thesis, we focus on 3D polygonal models typically used in subdivision modeling. We assume that the artist designed each mesh with an explicit topology. In other words, we assume that the faces that make up the mesh describe the object's shape and structure and are not simply an approximation of the shape. Therefore, we attempt to maintain the original intent of the artist and avoid interpolating or extrapolating data, in terms of shape and structure. Although this assumption limits the scope of our projects, this assumption is common for a large artist-driven domain of workflows and sets our work apart from other related work.

The remaining chapters address three major areas of studying mesh editing workflows, detailed below, and possible future directions of this research.

Visualization. Chapters 2 and 3 describe an interactive visualization system for mesh editing workflows. Our focus is to automatically generate a summarized visualization of a workflow sequence to provide a high-level overview or low-level details along with visual annotations of the data. We present two approaches for summarizing the workflows. In Chapter 2, we discuss linearly clustering the modeling workflow using the operation name or type alone, clustering edits together by editing patterns, in a system called MeshFlow. In Chapter 3, we discuss non-linear clustering the workflow using a metric on the effect of the operation(s), summarizing edits together based on strength and distance, in a system called 3DFlow. With both of these methods, the viewer is able to choose the level of summary appropriate to the task.

Management. In Chapter 4, we show how to diff and merge meshes without needing to instrument the editing software or use a special data structure or naming conventions, similarly to the diff and merge workflows for text version control. MeshGit does this by approximating the mesh edit distance, a measure of change between two meshes analogous to the string edit distance, and then converting it to low-level mesh edit operations. We evaluate MeshGit by diffing and merging a variety of meshes and find it to work well for all.

Comparison. In Chapter 5, we discuss some early work in comparing multiple artists performing similar mesh editing tasks. Four modeling subjects reproduce four different spacecraft either by following the step-by-step instruction of a video tutorial or by reconstructing from scratch a given target mesh. From the workflows, we building correspondences along and between the workflows and then computing pairwise edit distances. The distances are visualized as a heat map and by embedding in 3D using nonlinear dimensionality reduction techniques.

Chapter 2

MeshFlow: Interactive Visualization of Mesh Construction Sequences

This chapter describes how to visualize and summarize a mesh construction sequence.

2.1 Overview

The construction of polygonal meshes remains a complex task in Computer Graphics, taking tens of thousands of individual operations over several hours of modeling time. The complexity of modeling in terms of number of operations and time makes it difficult for artists to understand all details of how meshes are constructed. We present MeshFlow, an interactive system for visualizing mesh construction sequences. MeshFlow hierarchically clusters mesh editing operations to provide viewers with an overview of the model construction while still allowing them to view more details on demand. We base our clustering on an analysis of the frequency of repeated operations and implement it using substituting regular expressions. By filtering operations based on either their type or which vertices they affect, MeshFlow also ensures that viewers can interactively focus on the relevant parts of the modeling process. Automatically generated graphical annotations visualize the clustered operations. We have tested MeshFlow by visualizing five mesh sequences each taking a few hours to model, and we found it to work well for all. We have also evaluated MeshFlow with a case study using modeling students. We conclude that our system provides useful visualizations that are found to be more helpful than video or document-form instructions in understanding mesh construction.

2.2 Introduction

Mesh Construction For many applications in Computer Graphics the shape of objects is represented as polygonal meshes, either rendered directly or as subdivision surfaces. In most cases, these meshes are modeled by designers using polygonal modeling packages, such as Maya [4], 3ds Max [3], or Blender [9]. Even for relatively simple shapes, such as the ones shown in Figure 2.1, the construction of polygonal meshes remains a complex task, taking tens of thousands of individual operations over several hours of modeling time. The complexity of the modeling tasks in terms of number of operations and time makes it difficult for artists to understand all details of how meshes they did not build are constructed.

Without access to an instructor, it is common to use tutorials in either video or document format, e.g., from a book or website. For mesh construction, both of these formats have severe drawbacks. On the one hand, a video tutorial contains all the necessary details to construct the mesh, but long recording time (several hours) makes it hard to get an overview of the whole process. On the other hand, a carefully prepared document provides a good overview of the whole process, but skips many details that are necessary for correct construction.

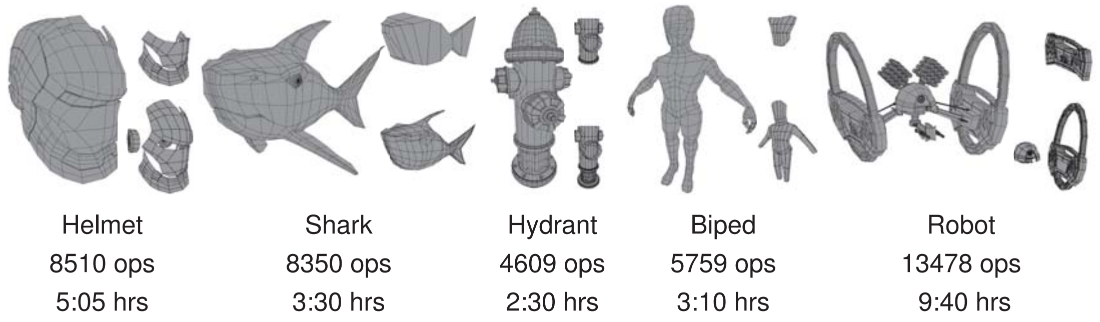


Figure 2.1: *Five input models, number of operations in construction history, and approximate time to complete.*

MeshFlow In this paper we present MeshFlow, a system for the interactive visualization of mesh construction sequences. These sequences are obtained by instrumenting a modeling program, in our case Blender, to record all operations performed by an artist during mesh construction. In its simplest form, MeshFlow can be used to play back every operation made by the artist, similarly to a video, while allowing the viewer to control the camera. The real strength of our system, though, is a hierarchical clustering of the construction sequence that groups similar operations together at different levels of detail. We motivate our clustering by an analysis of the frequency of repeated operations found in mesh construction sequences. To visualize the clustered operations, we introduce graphical annotations that we overlay on the model. Figure 2.2 shows examples of annotated clustered operations for the mesh sequences used to create the models in Figure 2.1.

In MeshFlow, the top level clusters provide an *overview* of the construction process, while the ability to change the level of *detail on demand*, all the way down to individual operations, ensures that viewer has all the information needed to reproduce the model exactly. Furthermore, we allow the viewer to *focus* on specific aspects of the construction process by filtering operations based on either their type or which parts of

			Operations					
Model	**Vertices**	**Time**	***total***	***view***	***select***	***trans***	***topo$_a$***	***topo$_b$***
Helmet	1342	5h05m	8510	4941	2020	1264	126	64
Shark	940	3h30m	8350	4668	1986	1563	61	51
Hydrant	10435	2h30m	4609	2430	1364	519	157	84
Biped	564	3h10m	5759	2741	1669	1236	60	31
Robot	16081	9h40m	13478	8296	2877	1648	347	151

Table 2.1: *Input data statistics. This table breaks down the construction statistics of the five models visualized by our system. Definitions of operations follow. cam: camera changes; vis: visibility changes; view: cam or vis; select: selection operations; trans: transformation operations; topo$_a$: loopcut, subdivide, extrude, delete; topo$_b$: add edge/face, merge vertices/triangles*

the model they affect.

Contributions We believe that by combining automatically generated annotations with the functionality for overview, detail-on-demand, and focus, MeshFlow has the benefits of both video and document tutorials. We have validated this intuition by asking eight subjects to compare MeshFlow with traditional tutorials, finding that our tool is highly preferable. To the best of our knowledge, MeshFlow is the first system to support this type of interactive visualization of mesh construction sequences.

2.3 Related Work

Design-workflow Visualization Our system for interactively visualizing mesh construction sequences is inspired by several recent works on visualizing designers' workflow. VisTrails [79], the closest system to our work, is a workflow provenance system. The system records actions performed in the application, displaying states as nodes in a graph, and allows the viewer to jump to any state in the workflow history (similar to an undo). Changes made to a previous state creates a version branch, and navigating

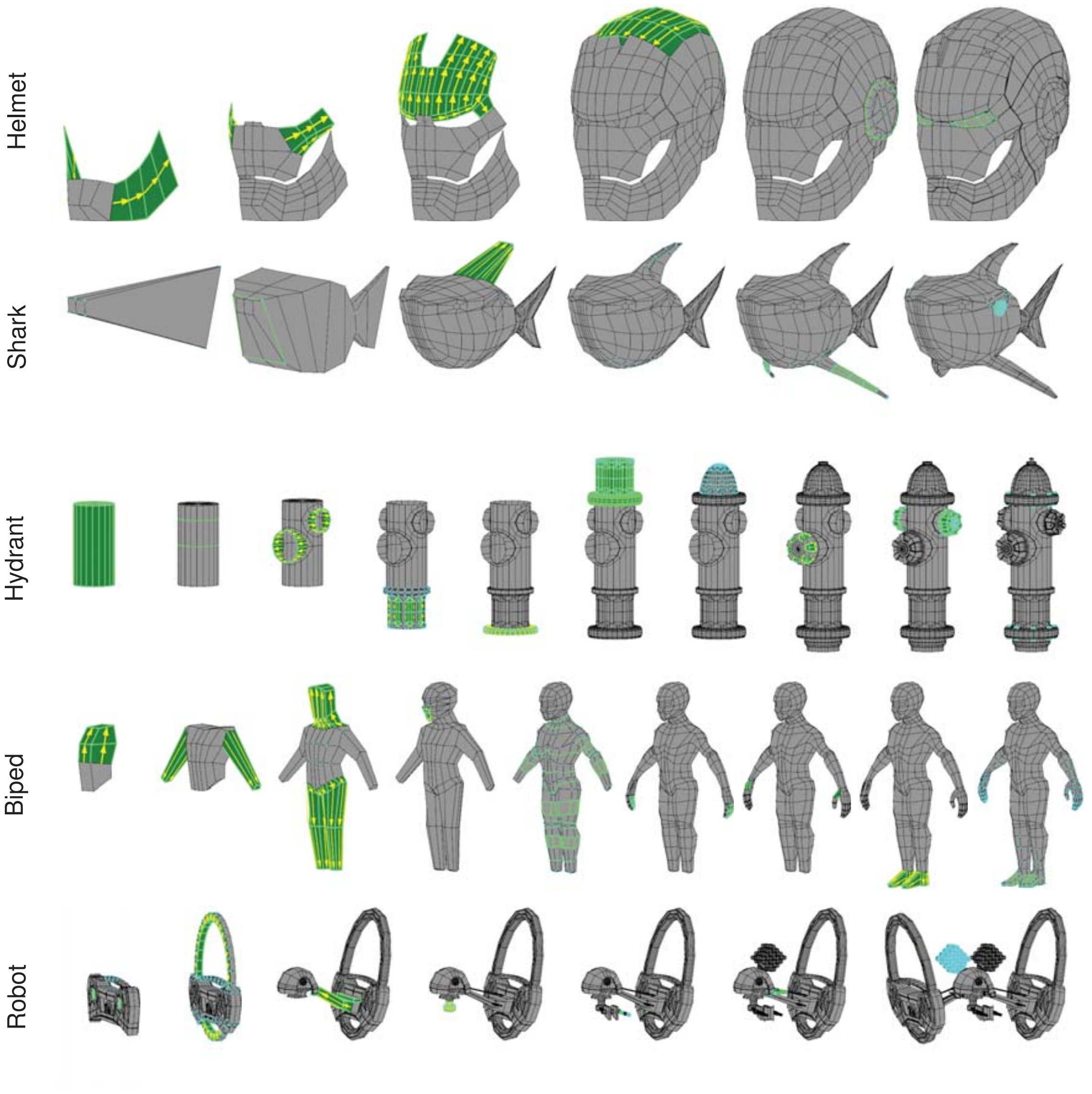


Figure 2.2: *Subset of clusters with annotations from level 10 for helmet, hydrant, biped, and robot; level 9 for shark. Green highlights indicate new, constructed geometry. Blue highlights indicate translated vertices. Yellow arrows indicate direction of extrusion.*

the history involves traversing a version tree. However, when a single version grows deeper than a few hundred edits, exploring the branch becomes similar to searching a long video sequence. In MeshFlow, we assume that undos are performed to correct mistakes, and we concentrate on a specific aspect of the provenance visualization problem: the practical and effective visualization of long sequences of editing actions through hierarchical clustering. In future work it would be interesting to combine their model version branching with our hierarchical operation clustering. Additionally, MeshFlow annotates the mesh by the edits performed. Grossman et al. [32] have developed a system to automatically generate a photo manipulation tutorial directly from the recorded steps of the artist. The system was designed to handle sequences of operations that are orders of magnitude shorter than ours. While parameter tuning and repeated operations are grouped into single steps, long sequences of different operations are not grouped. Chronicle [34] is an interactive system for visualizing and exploring long image editing histories. While their system is scalable to record and navigate several hours of work, the exploration of the edit sequence involves using a detailed timeline and before-and-after thumbnails, delimited first by save-times and then by edit-times. While this is effective for image manipulations, we found instead that for mesh modeling sequences clustering is necessary to provide a clear overview. Visualizing workflows is a well-explored topic in HCI research [6, 7, 43, 54, 72]. Because they focus on a smaller number of individual steps rather than summarizing long sequences, these methods are not well-suited for very long sequences as navigation becomes difficult.

Summarizing Video Sequences There is a large body of work on finding and visualizing a small set of representative keyframes for a video sequence [2, 5, 22, 38]. These approaches use image analysis and optimization to determine keyframes that are semantically important and should be present in the summary. In MeshFlow we take a different approach and summarize mesh sequences by only analyzing operation tags. We plan to extend our system to include geometry analysis to reap some of the benefits of the summaries presented in these works.

Tutorials Palmiter and Elkerton [60] and Harrison [35] have shown that image-based tutorials are far more effective than video-based instructions, due to the fact that users are able to work at their own pace. Narayanan and Hegarty [56] report that the structure and content of instructional materials are important for learning and understanding. Kelleher and Pausch [39] has shown that graphical overlays help with focus and reduce confusion. Many of these previous studies focus on relatively short design tasks. In MeshFlow, we focus on design tasks that take several hours to compute. In our domain, we found that video and document tutorials fundamentally work at different levels of detail and each have strong benefits but significant drawbacks. In MeshFlow, we let the viewer choose the level of detail interactively to capture the benefits while avoiding the drawbacks. For a more in-depth comparison, refer to Section 2.7.

Complex Model Visualization Many recent papers show how to effectively explore a complex model by showing how parts relate spatially and interactively to one another in the finished model. In order to focus on a particular part of a model, occluding parts are cut into or hidden [49] or split and separated [47]. Nakamura and Igarashi [53] visualize models of mechanical assemblies by indicating motions with annotations and causal chains. While all of these approaches isolate parts in a finished model, MeshFlow focuses on visualizing the temporal construction. For future work, we would be interested in combining these techniques with our work.

2.4 Mesh Construction Sequences

Data Capture The input to our visualization system is a mesh construction sequence, where each step is defined by a polygonal mesh, a tag that indicates the operation performed by the modeler, the current camera view and the current selection. In

	Level 1	Level 2	Level 3	Level 4	Level 5	Level 6	Level 7	Level 8
Helmet	*cam cam* (46)	*select select* (18)	*select select* (19)	*select trans* (22)	*trans trans* (40)	*topo*$_a$ *trans* (18)	*topo*$_a$ *trans* (21)	*cam topo*$_a$ (24)
	select select (15)	*select trans* (16)	*select trans* (17)	*trans select* (18)	*trans cam* (23)	*trans cam* (14)	*trans cam* (16)	*topo*$_a$ *cam* (24)
	select trans (13)	*trans select* (13)	*trans select* (13)	*trans cam* (14)	*cam trans* (21)	*cam topo*$_a$ (14)	*cam topo*$_a$ (15)	*topo*$_a$ *topo*$_a$ (18)
	trans select (11)	*cam select* (12)	*cam select* (12)	*cam select* (12)	*topo*$_a$ *trans* (4)	*trans topo*$_a$ (11)	*trans topo*$_a$ (13)	*topo*$_a$ *topo*$_b$ (7)
Shark	*cam cam* (45)	*select trans* (22)	*select trans* (22)	*select trans* (27)	*trans trans* (45)	*trans cam* (18)	*trans cam* (21)	*cam topo*$_a$ (29)
	select trans (12)	*trans select* (16)	*trans select* (16)	*trans select* (20)	*trans cam* (24)	*cam topo*$_a$ (16)	*cam topo*$_a$ (18)	*topo*$_a$ *cam* (26)
	trans select (9)	*cam select* (13)	*cam select* (14)	*trans cam* (14)	*cam trans* (23)	*cam trans* (12)	*cam trans* (14)	*topo*$_b$ *cam* (12)
	cam select (7)	*select select* (12)	*select select* (12)	*cam select* (13)	*cam topo*$_a$ (2)	*topo*$_a$ *trans* (10)	*topo*$_a$ *trans* (12)	*cam topo*$_b$ (9)
Hydrant	*cam cam* (43)	*select select* (32)	*select select* (33)	*select trans* (16)	*trans trans* (30)	*cam topo*$_a$ (15)	*trans cam* (17)	*topo*$_a$ *cam* (26)
	select select (18)	*select trans* (9)	*select trans* (9)	*trans select* (11)	*trans cam* (15)	*trans cam* (12)	*topo*$_a$ *trans* (16)	*cam topo*$_a$ (24)
	select trans (5)	*cam select* (9)	*cam select* (9)	*trans cam* (11)	*cam trans* (12)	*topo*$_a$ *trans* (11)	*cam topo*$_a$ (15)	*topo*$_a$ *topo*$_a$ (14)
	cam select (5)	*trans select* (6)	*trans select* (7)	*trans trans* (11)	*cam topo*$_a$ (7)	*topo*$_b$ *topo*$_b$ (10)	*trans topo*$_a$ (10)	*cam topo*$_c$ (10)
Biped	*cam cam* (33)	*select trans* (22)	*select trans* (22)	*select trans* (27)	*trans trans* (40)	*trans cam* (17)	*trans cam* (20)	*cam topo*$_a$ (30)
	select trans (15)	*trans select* (16)	*trans select* (16)	*trans select* (20)	*trans cam* (26)	*cam topo*$_a$ (15)	*cam topo*$_a$ (17)	*topo*$_a$ *cam* (29)
	trans select (11)	*cam select* (13)	*cam select* (14)	*trans cam* (15)	*cam trans* (25)	*cam trans* (14)	*cam trans* (16)	*topo*$_a$ *topo*$_a$ (9)
	cam select (9)	*select select* (13)	*select select* (13)	*cam select* (14)	*cam topo*$_a$ (2)	*topo*$_a$ *trans* (11)	*topo*$_a$ *trans* (12)	*topo*$_b$ *cam* (7)
Robot	*cam cam* (48)	*select select* (21)	*select select* (22)	*trans cam* (15)	*trans trans* (28)	*cam topo*$_a$ (16)	*trans cam* (18)	*cam topo*$_a$ (28)
	select select (11)	*cam select* (14)	*cam select* (14)	*trans trans* (15)	*trans cam* (22)	*trans cam* (15)	*cam topo*$_a$ (18)	*topo*$_a$ *cam* (28)
	cam select (7)	*select cam* (10)	*select cam* (10)	*cam select* (13)	*cam trans* (19)	*topo*$_a$ *trans* (14)	*topo*$_a$ *trans* (18)	*topo*$_a$ *topo*$_a$ (15)
	select cam (5)	*trans cam* (10)	*trans cam* (10)	*cam trans* (10)	*cam topo*$_a$ (6)	*topo*$_a$ *cam* (9)	*topo*$_a$ *cam* (10)	*cam topo*$_c$ (6)

Table 2.2: *Top four bigrams for each of the models at levels 1–8 (all levels available in supplemental material). Numbers in parenthesis indicate percentage of all bigrams from the sequence at that level.*

our sequences, we capture a step for each operation that changes the mesh, its per-component visibility, the viewing camera, or the mesh's per-component selection. We store the mesh as a list of vertices, uniquely labeled, defining its geometry and a list of faces represented as vertex lists.

We record this sequence by instrumenting Blender [9], an open source animation package, comparable, with regard to polygonal modeling, to commercial systems such as Maya [4] or 3ds Max [3] Our mesh construction sequences are generated automatically while the modeler is building a mesh; this is in contrast to tutorials that need to be authored after the modeler has built the mesh. We supply our instrumentation as supplemental material.

Mesh Sequences While our data capture works for any mesh, we focus on visualizing mesh construction histories of single objects, rather than full scenes. To demonstrate the usefulness of MeshFlow, we recorded the construction of five meshes, shown in Figure 2.1. Figure 2.2 shows a few steps of the construction process annotated by our system. We built the models using tutorials found on the web. The helmet and shark models were based on document tutorials [37, 27]; the hydrant, biped, and robot models were based on video tutorials [73, 80, 24]. Three different modeling "techniques" were used: box modeling (where a single mesh is subdivided to add detail), surface extrusion (where the surface is grown using successive extrusions), and modeling by parts (where individual components are modeled separately). All five sequences are supplied as supplemental material.

Table 4.1 shows various statistics for each of our models. Note how even for these simple models, hours of modeling time was employed. This is due to the need for several thousands of operations to construct the meshes, even in cases where only half of the model is built due to symmetry. The construction process is traditionally documented by video recordings or documents with textual explanations and images. When the process takes many hours, a video recording becomes tedious and difficult to search, for a viewer. It can be useful to condense this information into a document with illustrations, but even with considerable work in authorship, details will be selected and aggregated in a static way.

Operations in the modeling sequence range from user interface commands, to geometric transformations, to topological changes in the mesh. We define five groups of operation types, listed in Table 4.1: *view* for operations that either change the camera (*cam*) or hide/show geometry (*vis*), *select* for operations where geometry components are selected to be modified, *trans* for translation, rotation, or scaling transformations, $topo_a$ for the topological operations of loopcut, subdivision, extrusion, and deletion, and $topo_b$ for the topological operations of add edge, add face, merge vertices, and merge triangles. We split topological operations into $topo_a$ and $topo_b$ because $topo_b$ operations are typically used as patchwork edits in conjunction with members of $topo_a$. We include a third type, $topo_c$, for the creation of disjoint geometric primitives such as creating spheres and boxes, but these operations are very uncommon compared to the others.

To gain the benefits of video and document tutorials without their drawbacks, we need to provide a way to view modeling sequences at different levels of detail. Our analysis of construction sequences on the five models revealed a great deal of repetition within and between operation types (see Table 4.1 and Section 2.6 for an analysis). We use this repetition to hierarchically group operations into clusters, from a high-level overview of the modeling process, all the way down to the individual low-level operations needed for reproducing the mesh exactly. We allow users to interactively choose the desired levels of detail gaining the benefits of both overview and detail-on-demand.

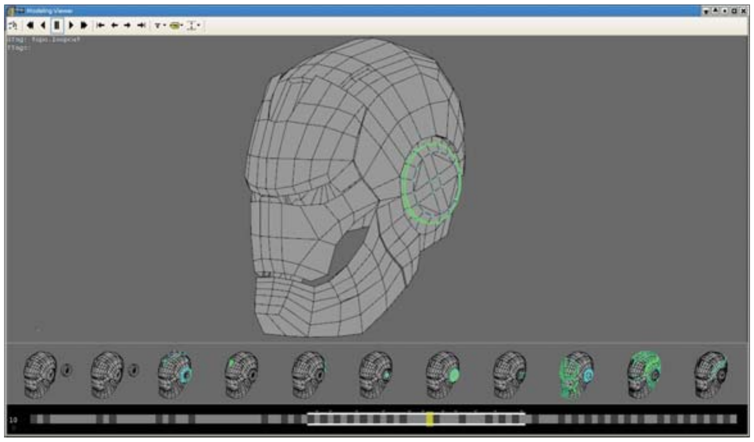


Figure 2.3: *User interface. A large view shows the mesh of the current cluster. Across the bottom is the timeline with indicators of the current cluster and any filtered clusters. The thumbnails show changes at different places along the timeline.*

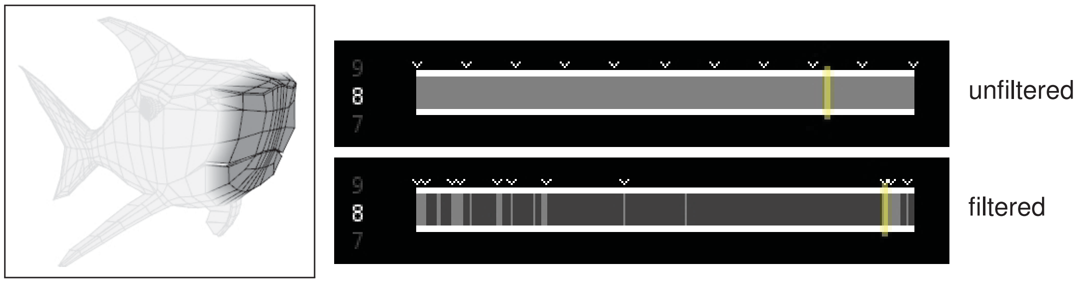


Figure 2.4: *Shark model with snout highlighted and the corresponding timeline with and without filtering. To focus on edits affecting only specific regions of the mesh, the viewer highlights the areas of interest, and the timeline is filtered to show the clusters that modify these areas.*

2.5 MeshFlow

Visualization System In this section we describe briefly our visualization system from a user perspective. We suggest that the reader consult our video for a demonstration of the various concepts listed here. MeshFlow provides an interface for interactively exploring the mesh construction history. The interface includes a large view of the mesh, a timeline, and thumbnail views of the mesh at different places along the timeline. Figure 2.3 shows a screenshot of our user interface. In its simplest form, the visualization system can be used to play back every operation made by the modeler, similarly to a video, except the viewer can control the camera in addition to using the original modeler's camera views.

Operation Clustering The real strength of our system compared to traditional recordings comes from the use of interactive level of detail through operation clustering. Our approach differs from the work of Grabler et al. [32] and Nakamura and Igarashi [54] in that we group operations together in a hierarchical fashion, where lower cluster levels have more details than higher ones. This allows us to get a visual summary of high-level changes in the mesh, while providing several levels of detail that can be accessed on demand. By changing the level of detail, a viewer can choose to see a summary of the edits or get details on demand. The timeline at the bottom of the interface is discretized into clusters, such that only one cluster is viewable at a time. The main view of our interface displays the resulting mesh from the clustered operations viewed from the average camera location (or a user controlled camera if so desired). To determine our clustering, we analyzed the recurrance of patterns of operations in the input sequence and found that clustering based solely on operation tags works well, without requiring geometric analysis. Section 2.6 covers our clustering methodology in detail.

Visual Annotations We added graphical annotations to illustrate the types of operations that were performed in a cluster, which can be seen in Figure 2.2, similar to Grabler et al. [32] and Su et al. [72]. These annotations color vertices, edges, and faces of the mesh to indicate mesh changes like adding topology (green), moving vertices (blue), and selection (orange). We further add annotations to indicate common operations such as arrows for extrusion and lines for loop cuts. Selection is usually active in many places on the mesh, so we allow it to be turned on and off when necessary to reduce clutter. The main view includes annotations indicating all operations performed in the current cluster. The thumbnails contain annotations indicating changes since the previous thumbnail, emphasizing modifications as in the timeline at that location. Section 2.6 covers these annotations in detail.

Filtering We have found it useful to be able to focus quickly on subsets of operations. To achieve this we give viewers the ability to filter operations and clusters. This can be important for speeding up the viewing process, but also for visualizing how operations group over time and at what frequency. When a filter is activated, all clusters that match the filter are darkened in the timeline (see Figure 2.4), made unselectable, and skipped during playback. We support two main filtering modes. First, filtering by operation type allows for operations and clusters tagged with that type (selection, transform, etc.) to be easily identified and skipped. This allows for focusing on different "techniques" used when modeling. Second, inspired by the Data Probe in Chronicle [34], filtering by vertex selection allows the viewer to highlight vertices and skip clusters that do not affect those vertices. This allows the viewer to focus on how specific parts of the model are built in their entirety. For geometry filtering, we further highlight the region of interest by deemphasizing the remainder of the model (see Figure 2.4). Our system will automatically tag data during capture, but both modelers and viewers can provide their own custom tags, e.g., tagging operations spatially with labels like "torso" or "wheel", or temporally with labels like "blocking phase" or "refinement phase".

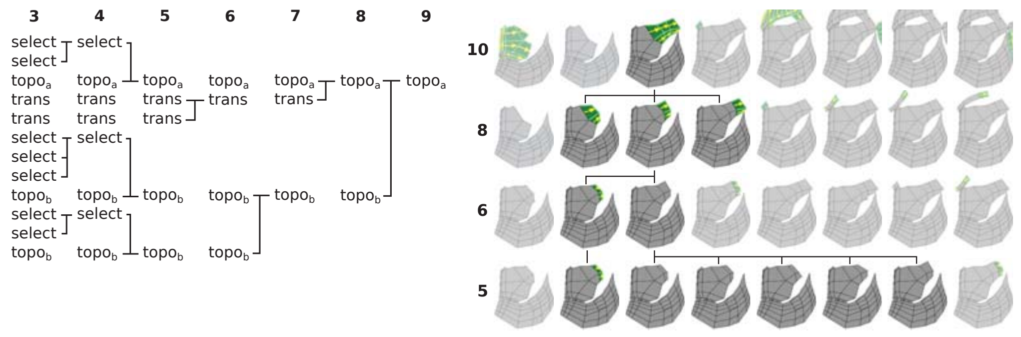


Figure 2.5: *Two examples of successively applying levels of clustering. The left figure shows the operation names for levels 3–9, while right figure shows screenshots of the model for levels 5, 6, 8, and 10. See Table 2.3 for clustering rules.*

2.6 Operation Clustering

Clustering by Regular Expressions The mesh sequences described in Section 2.4 contain a great deal of repeated operations. In order to provide a clear overview of how the model is built we need to group low-level operations into clusters representing high-level structural changes. To identify such groups, one might attempt to analyze geometric properties to learn when large semantic changes to the mesh have occurred. We have discovered, though, that clustering based solely on operation tags can establish meaningful levels of detail without attempting to learn semantics within the sequence (see Section 2.7). To group operations together, we apply substituting regular expressions defined on the operation tags. We derive these regular expressions by identifying repeated patterns of operations and combine them into clusters that can be visualized at once. As two examples, selections and vertex transformations are often achieved by many repeated atomic selection or transform operations. We can cluster these into a single cluster representing the net change in selection state or vertex locations.

To create a hierarchy of detail levels, we apply successive regular expression substitutions and let the user interactively choose the displayed level. In our implementation

	Clustering Regular Expressions
2	$(cam)+\ (cam)^{\diamond} \mapsto (cam)^{\diamond}$
3	$(view)\ (view)+ \mapsto (cam)$
4	$(select)\ (view\|select)*\ (select)^{\diamond} \mapsto (select)^{\diamond}$
5	$(select)\ (view)*\ (topo\|trans)^{\diamond} \mapsto (\cdot)^{\diamond}$
6	$(trans)+\ (view)*\ (trans)^{\diamond} \mapsto (\cdot)^{\diamond}$
7	$(\cdot)^{\diamond}\ (view\|(\cdot)^{\diamond})*\ (\cdot)^{\diamond} \mapsto (\cdot)^{\diamond}$
8	$(topo)^{\diamond}\ (view\|trans)*\ (trans) \mapsto (\cdot)^{\diamond}$
9	$(topo_a)^{\diamond}\ (view\|topo_b)*\ (topo_b) \mapsto (\cdot)^{\diamond}$
10	$(\cdot)^{\diamond}\ (view\|(\cdot)^{\diamond})*\ (\cdot)^{\diamond} \mapsto (\cdot)^{\diamond}$

Table 2.3: *Regular expressions used to generate levels 2 to 10. For each level the group of elements that matches the regular expression is replaced with a single cluster. Legend:* $*$ *and* $+$ *match 0-or-more and 1-or-more repetitions respectively;* $(\cdot)$ *matches anything;* $(a|b)$ *matches either a or b;* $\diamond$ *indicates a back-reference group.*

we provide 11 successive levels of detail. Table 2.3 shows a list of regular expressions used for each level of detail. Figure 2.5 shows an example of executing the regular expressions at different levels of detail. In the latter example, we show start and end states of a group of repeated extrusions and vertex movements, and then see each separate extrusion without viewing every individual selection and transform of vertices.

Removing Undos The original sequence, called *Level 0*, will contain all operations a modeler has performed, including work that is undone. We assume that undos are used to correct mistakes, rather than used for exploration purposes. Our first cluster level, referred to as *Level 1*, cleans up the data stream by removing undone work. We look in the stream for identical mesh states and remove all operations in between, effectively making undos invisible.

Initial Clustering To choose regular expressions that represent effective levels of detail, we analyze the mesh sequences for our data set. We measure the frequency of bigrams, or instances of pairs of operation types. Table 2.2 lists the four most frequent bigrams for cluster levels 1–8. Note that after we cluster undos (*Level 1*), repeated camera changes are the most frequent, roughly half of all bigrams in some cases, followed by repeated selections. Repeated camera movements likely come from the artists either viewing the model from different angles or simply adjusting the view carefully. Visibility operations (show/hide geometry), albeit not as frequent, are similarly motivated. Note also that repeated adjustments to display the mesh do not alter the mesh. For repeated selections, it is likely that the modeler was building up a large selection set for a successive operation, thus we can safely group them together. Similarly to view changes, these also do not alter the mesh. These observations motivate the next three levels of clustering.

In *Level 2* we replace repeated camera view changes with a single view cluster, picking the last camera view as the cluster view. *Level 3* clusters all repeated visibility and camera operations together. Visibility is clustered at this level for semantic reasons rather than a bigram frequency because it forms clusters affecting only view operations. We then cluster repeated selections together, in *Level 4*, as this is a highly common bigram and since this likely prepares larger selections for successive operations. We set the selection of the resulting cluster as the net result of the successive selections. At this point we have clustered together all operations that do not affect the mesh.

Clustering Editing Operations After *Level 4* we begin to cluster operations that alter the mesh. At this point, transforms that follow selections are the most frequent bigrams on our sequences. This makes sense, since something must be selected to be edited. Thus, in *Level 5*, we cluster selection with the subsequent editing operation. The next most common bigram is repeated transformation. The combined effect of repeated translation, rotation, and scaling operations can be thought of as simply modifying the positions of vertices. We can cluster these together in *Level 6* such that the resulting vertex positions are the net change in position. Now we have another situation where semantics outweigh our bigram analysis. We take this opportunity to create a level of detail that clusters all repeated operations no matter what they are (essentially cleaning up repeated homogeneous topology changes), forming *Level 7*. In practice, we found this to be an effective level of detail with easily recognizable meaning. Note that topology operations are only clustered if they have the same tag, e.g., extrude with extrude, not extrude with loopcut.

Clustering Groups of Editing Operations So far we have clustered together editing operations of the same type. We will now combine these clusters with each other to form higher level groups of operations with more heterogeneity. The most common bigram in *Level 7* is topology operations followed by transformations. This makes sense, since new topology is often shaped after being created. In *Level 8* we cluster topology changes with any subsequent transform cluster, combining, in most cases, the creation of new geometry with the shaping of that geometry. A good example of this is seen in *Level 8* of Figure 2.5.

Until now we have been thinking of topological operations together, but we now introduce the classification of types $topo_a$ and $topo_b$ (see Section 2.4 and Table 4.1). Operations in $topo_a$ represent major structural change to the mesh, often changing the number of edge loops or overall complexity, whereas $topo_b$ operations are used as patchwork in conjunction with $topo_a$ operations, filling holes and cracks by merging or connecting things. For example, on the crown of the helmet each edge loop is extruded and then attached to the head before starting the next extrusion. *Level 9* clusters instances of $topo_a$ with subsequent $topo_b$ operations. Finally, in *Level 10*, we cluster repeated instances of the case from *Level 9*, visualizing large components of the mesh being constructed all at once. Depending on the model, *Level 9* or *Level 10* yields a

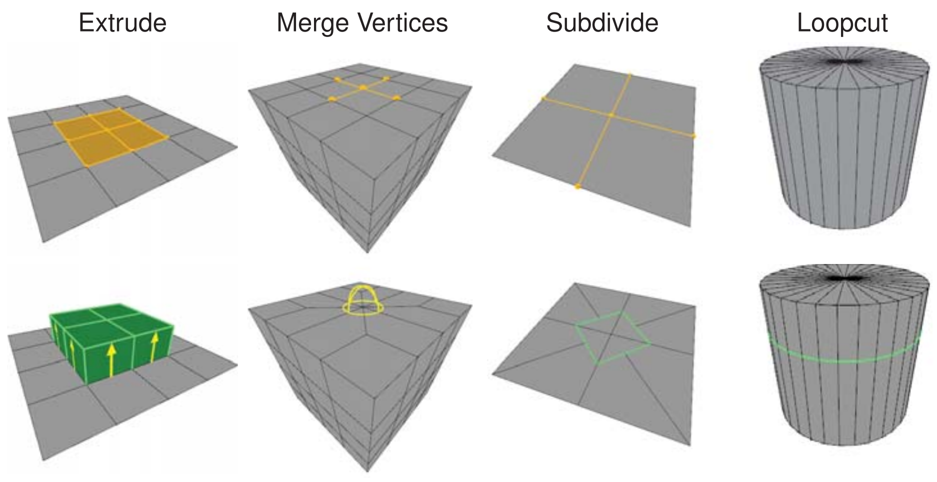


Figure 2.6: *Various automatically-generated annotatians. For illustrative purposes, the top row has selections drawn; the bottom row does not.*

concise overview that is easily visualized in a matter of seconds.Though heterogeneous $topo_a$ pairs are our most common non-camera bigram past *Level 8*, we do not combine them here, because we find that this causes semantically ambiguous situations and unclear level of detail.

Visual Annotations When drawing the mesh corresponding to each cluster, we highlight changes performed in the cluster to draw user attention. We use color coding to indicate simple changes: green for added geometry, cyan the transformed vertices, and orange for selection. For the most common topology operations, we overlay visual annotations on the resulting mesh to indicate what operations types are performed in each cluster. Figure 2.6 shows a summary of such annotations. We annotated *extrusions* by drawing yellow arrows on both sides of newly created faces. For *subdivisions* and *loopcuts*, the edges involved are highlighted in green. For vertex or face *merge* operation, we draw yellow circles at the location of the final vertex or face respectively.

Model/Level	0	1	2	3	4	5	6	7	8	9	10
Helmet	8510	8203	4274	4235	3190	1912	381	335	212	183	108
Shark	8350	8303	4587	4567	3762	2245	252	217	133	100	61
Hydrant	4609	4496	2579	2542	1483	1034	528	361	227	214	124
Biped	5759	5704	3826	3781	3118	1843	252	225	129	115	58
Robot	13478	13137	6809	6639	4321	3073	1247	998	639	596	326

Table 2.4: *Number of clusters for five models at each level of detail.*

2.7 Evaluation

Overview We run our system on an Intel Core2 3.0GHz quad-core processor with 4GB of RAM, and an NVIDIA GeForce 9600GT GPU. On this system, exploring mesh sequences on all meshes in our dataset is interactive. We provide all source code and mesh sequence data files to allow readers to experience our visualization. We also include our Blender instrumentation. All visualization features and annotations shown in the paper and supplemental videos are automatically generated by our system with no authoring overhead for the modeler. We found that our regular expression grouping consistently works very well in reducing sequence complexity. Table 2.4 shows the number of operations at each level of detail, going from several thousands operations to just hundreds. This supports our claim that a simple frequency analysis of the operation is sufficient for reduction. We include videos showing three levels of detail for each mesh as supplemental materials, as well as an overview of the interface in our video submission. In the next subsection we will introduce a case study that supports our claim that these operation reductions are effective in aiding understanding for viewers.

2.7.1 Limitations

The most obvious limitation of MeshFlow is that it focuses solely on polygonal meshes, whereas other surface representations are also useful. It is our belief, though, that the

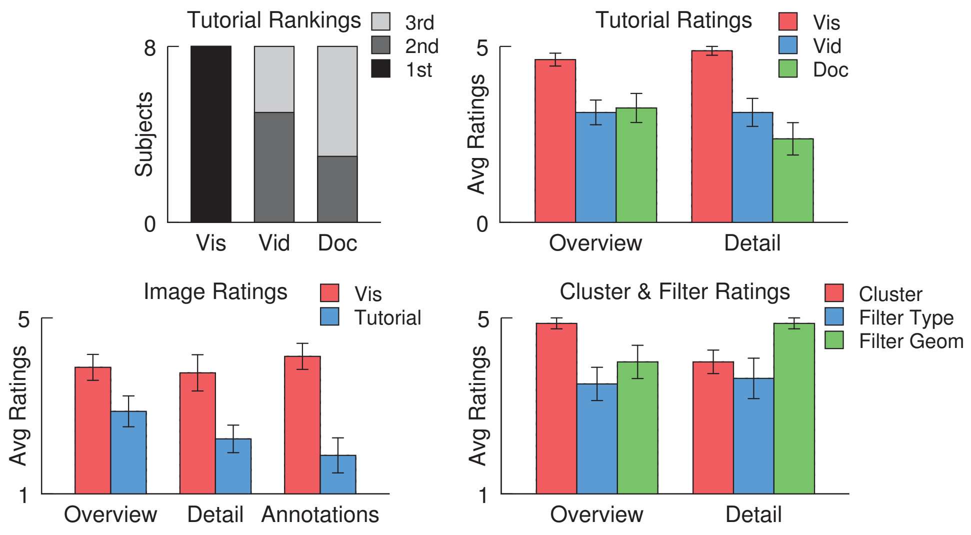


Figure 2.7: *Data from our case study. From left to right: preference rankings for MeshFlow (vis) compared to traditional video (vid) and document (doc) tutorials; ratings for MeshFlow compared to traditional tutorials for overview and detail usefulness; ratings for MeshFlow images compared to authored tutorial images for overview, detail, and graphical annotation usefulness; ratings for clustering and filtering features for overview and detail usefulness. Error bars represent standard error.*

majority of MeshFlow can be extended to support other surface representations such as NURBS. The primary limitation in performing such extension is that our clustering algorithm would need to address the presence of new operators, specific to modeling other geometric representations. We are confident, though, that this can be accomplished by analyzing operation frequencies and following our methodology. Additionally, the sequences used for analysis contained only a subset of the operations available in Blender. While this subset was able to construct a variety of models, in future work we would like to explore sequences containing operations from other modeling styles, such as sculpting. For polygonal meshes, our clustering based on only regular expressions could be improved. First, we only support clustering expressions sequentially, but it could be useful to investigate methods to cluster operations out of order to better highlight patterns on different parts of the model. Second, we made no attempt to determine what clusters have more semantic importance when editing a mesh. This would require some form of geometry analysis that could quantify the importance of mesh changes. Third, it would useful to be able to recognize parts of the model to create even higher level clusters. For example if we could recognize that a set of vertices is modeling the nose (rather than the eyes), we could automatically cluster all those together; this was done for images in Grabler et al. [32] using face recognition. Such semantics would allow us to automatically generate audio and text annotations. Last, because it is a completely automated system, MeshFlow is not a replacement for hand-authored tutorials. However, MeshFlow can be easily extended to allow author-specified hints and tips using the tagging metaphor. (see Filtering in Sect. 2.5) While we are interested in addressing these limitations in future work, the following section will show that artists found our current system very useful and a significant improvement over available methods.

2.7.2 Case Study

We conducted a case study in which subjects were asked to evaluate MeshFlow compared to video and document tutorials. The study included 8 college modeling students, all of whom had previously completed at least one course in mesh modeling and had experience in creating models by following tutorials. When asked to rate their confidence level in completing a mesh they had never tried before using a tutorial, all but one rated themselves 4 or higher on a scale of 1 to 5, with the other rating a 3. We are confident that all subjects have enough experience to put MeshFlow in context with real modeling tasks.

Methodology We ask our subjects to make five comparisons of using MeshFlow to other options. For each comparison, subjects have 10 minutes to investigate a modeling sequence using MeshFlow and 10 minutes to investigate using the alternative. We guide the exploration by asking subjects to answer three specific questions about the modeling sequence (e.g., for the robot model, subjects were asked how the wheel was made to fit into the chassis). The investigator introduces the questions before the subject begins and remains on hand to guide the subject in using the interface. At the end of all five comparisons, we ask subjects to rate various aspect of MeshFlow and leave open comments regarding different aspects of the experience. Scanned questionnaires are supplied with supplemental materials.

First, we compare MeshFlow to traditional modeling tutorials for three of our models to determine whether MeshFlow is in fact effective as a visualization tool. We compare the helmet model in MeshFlow to its original document tutorial [37]. We then compare the biped model in MeshFlow to the original video tutorial [80]. Finally, for the shark model, we compare just still screenshots automatically generated by MeshFlow at level 9 to authored images taken from the original tutorial [27] with the text removed. For each of these comparisons, half the subjects where shown MeshFlow first, and the other half were shown the traditional tutorial first.

Second, we compare the use of MeshFlow with and without the ability to filter and cluster operations, to evaluate the relative importance of these features in model sequence exploration. We use the robot and hydrant models for these comparisons. First, the subject is given the model at the lowest level of detail and asked to answer our questions without using clustering or filtering. We then allow the user all clustering levels of detail and filtering methods to compare.

Results Figure 2.7 summarizes the ratings of subjects for the comparisons performed. In general, subjects are very enthusiastic about MeshFlow, and rate its features highly. We ask subjects to compare MeshFlow to video and document tutorials by rating the usefulness of each with respect to getting a general overview and in understanding details. Subjects rate MeshFlow superior to video and document tutorials in each of the two categories. This shows that MeshFlow not only has the benefits of traditional tutorials, but it outclasses them even in the area of their individual strengths. We also ask subjects to strictly rank their general preference between the three (MeshFlow, video, document). All subjects ranked MeshFlow as their preferred method. We also ask subjects to rate the set of images automatically generated by MeshFlow compared to the ones manually created for a document tutorial. Subjects rate each with respect to how useful they are in understanding an overview, the modeling details, as well as the clarity of the annotations. MeshFlow was rated much higher in all categories, with only one subject rating it lower than the tutorial images. We found this to be surprising, since MeshFlow was not designed to generate static image sequences, but interactive visualizations. Still, when comparing the automatically annotated clusters to hand-authored images, MeshFlow was found to be superior. Finally, we ask the subjects to rate the usefulness of clustering and filtering when trying to understand overview and details. For the most part, subjects rate all features high, indicating that clustering is the most useful feature for getting overviews, and filtering on specific vertices is the most useful for investigating details.

Observation To support our previous analysis, we collected open feedback from subjects' questionnaires, and now report the following quotes. All subjects preferred MeshFlow over the traditional alternatives. When asked why, one responded with "the ability to customarily look at parts of the geometry and changes to it that I was interested in, rather than being dependent on what the tutorial author thought I would want to to know." And another subject, "the interactive vis gives me the option of the level of detail. [...] It has more detail than a document and can leave out irrelevant detail that a video often comes with." When comparing the MeshFlow images to hand-authored ones, "I thought that the interactive vis better explained how the model was built. I liked the color scheme / familiar interface, as well as the ability to easily distinguish/identify what was being altered." And another, "the graphical annotation [in MeshFlow] says much more than a normal tutorial." Regarding the ability to filter, many subjects found this useful, commenting "the painting tool which then shows you where changes pertaining to that which was selected on the timeline is a fantastic time saver if you're focused on a detail". And another subject, "filtering by selected parts seemed very useful. Definitely fixed problem of having to guess or remember where in a tutorial or video a certain area is worked on." In terms of clustering, we found that subjects all had different interests, highlighting the importance of choosing the level of detail interactively. For example, one subject commented "clustering is key to finding the parts that you want to focus on" and another subject "clustering gives a good, rapid overview of the build," and yet another, "there are times when a general view is more helpful (clustering) and also times when a more detailed view is preferred (filtering). What sets the interactive vis apart is the ability to cater to both needs at any time." At least three of the subjects asked us after the study if we were going to release MeshFlow to the public, so they could start using it. One even wrote in the questionnaire "I would love to use this interactive vis tutorial in a digital arts modeling class. Though I suppose with it, the professor would not need to do much."

2.8 Conclusion

We have presented MeshFlow, a system for visualizing the construction process of polygonal meshes. MeshFlow combines overview, detail-on-demand, and focus by hierarchically clustering and filtering edits from a recorded modeling session. We based our clustering on an analysis of the frequency of repeated operations and implement it using substituting regular expressions on operations tags. We have tested Mesh-Flow on five mesh sequences and evaluated it with a case study. We conclude that our system provides useful visualizations that are found to be more helpful than video or document-form instructions in understanding mesh construction. For future work, we would like to focus on improving our clustering to support out-of-order grouping, highlight semantical changes and add voice and text annotations.

Chapter 3

3DFlow: Continuous Summarization of Mesh Editing Workflows

This chapter describes how to visualize mesh construction sequences using continuous summarization.

3.1 Overview

In Chapter 2, long edit sequences were summarized based on the type or name of the edit operation performed by the artist. Summarization rules were created based on operation n-gram analysis and discovery of operation patterns. This summarization technique works well when the edit operations are highly structured, where semantic of edit can be derived from operation type. However this may not be the case in other 3D workflows. For example, digital sculpting workflows do not have clear edit operation patterns that can be easily summarized in intuitive way. Furthermore, MeshFlow has a fixed number of levels of summary, all of which may not fit well the input data sequence.

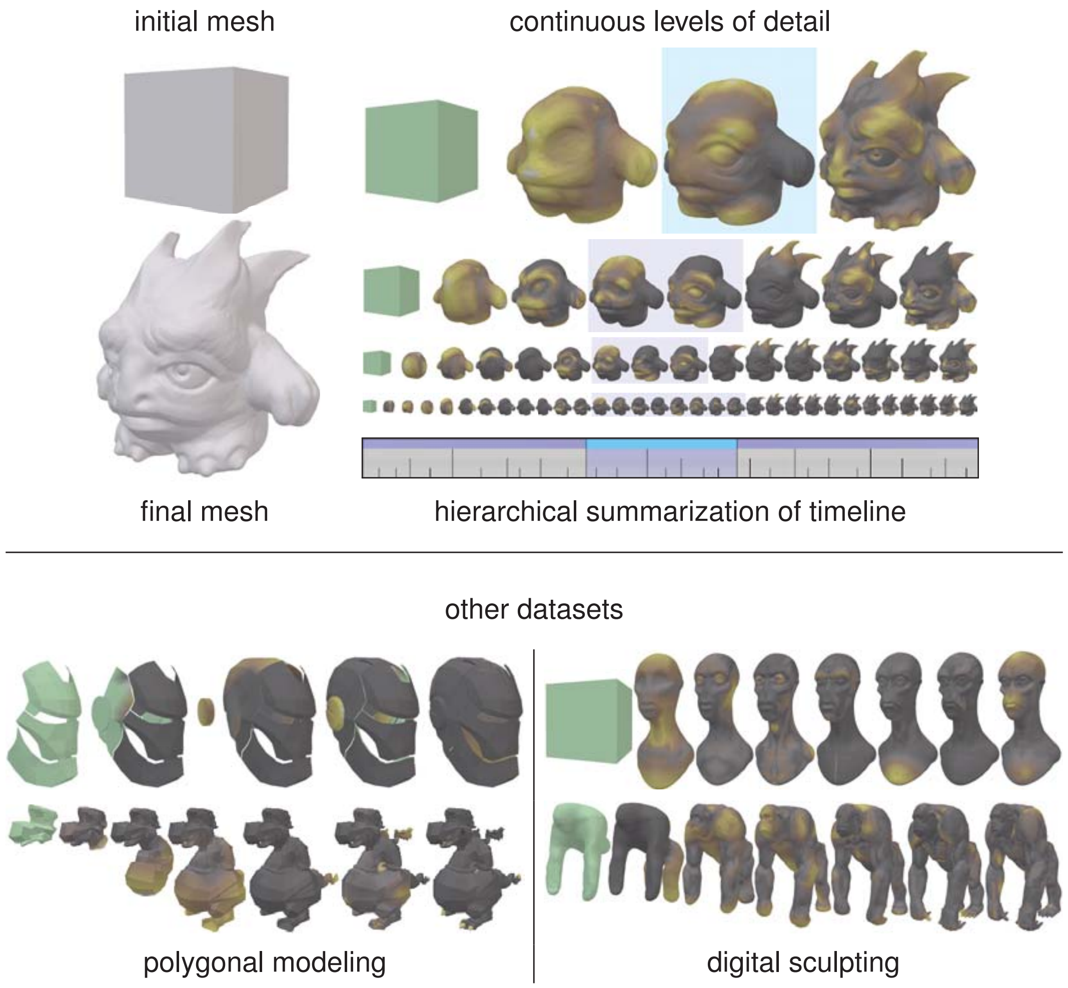


Figure 3.1: *(Top) Continuous levels of details automatically constructed from a 30 minute digital sculpting session of a professional artist, and (Bottom) four additional sequences shown at different levels of details. The top row shows how an artist sculpted a cube into a monster (left) in 797 strokes using dynamic remeshing techniques. The top-right shows the sequence summarized in 4, 8, 16, and 32 steps (top) and the corresponding timeline (bottom). The mesh is colored green to indicate created geometry and golden to indicate the strength of change from the previous mesh. Blue highlighting and vertical black lines indicate the hierarchical summarization. Two polygonal modeling workflows and two digital sculpting workflows are shown in the bottom row at different levels of detail.*

In the following sections, we describe a method of summarizing the edit sequence that depends on the *effect* of each operation or a group of operations rather than the name or type of the operation. This method provides continuous levels of summarization, from the raw input down to a single step, allowing the viewer to choose the most appropriate summarization level for the input data. We show that this method works well for digital sculpting as well as polygonal modeling sequences seen with MeshFlow.

Finally, the method does not require the input data to be a linear sequence. We show that delinearizing the data can allow the summarization method to cluster edit operations out-of-order.

3.2 Introduction

Various methods are used for learning how talented artists create polygonal meshes. Although document-based tutorials are an option, artists commonly showcase their workflows via a time-lapse or sped-up video recording of their editing session, since these videos are simple to create without interrupting their workflow. Even for relatively simple models, though, mesh editing workflows are long, ranging from tens of minutes to several hours of work, involving thousands of operations. Time-lapses are not very effective for these lengths since the artist must make a trade-off between presenting the details of their workflow and keeping the presentation as short as possible. Motivated by this concern, recent research has explored ways to visualize and navigate lengthy recordings of artists at work, for modeling as well as image editing. For example, *VisTrails* [79] helps in navigating non-linear undo histories in 3D software, while Chen et al. [21] present non-linear navigation of edits in images. *MeshFlow* (chapter 2) combines clustering of edits with annotations to get a summary of a polygonal modeling session. *Delta* [41] helps in comparing workflows in image editing. ZBrush [62] has a workflow playback feature just for creating time lapses.

In this paper, we focus on summarizing mesh editing workflows, including digital sculpting and low-poly modeling. In sculpting, artists alter the shape of a mesh as though they were sculpting a block of clay using physical tools. The digital brushes can have different effects, such as creating new features, smoothing out uneven areas, or reposing parts of the mesh. Sculpting is particularly well suited for modeling organic shapes like characters. In low-poly modeling, artists directly manipulate the surface representation of the mesh by issuing commands such as extrude edge, split face, add new cube, etc. This workflow is particularly well suited for modeling hard-surface objects, meshes that will be animated or base meshes for subdivision surfaces.

There are two major working phases with modeling workflows: *blocking* and *refinement*. In blocking, the main shape of an object is roughed out. Blocking edits have strong magnitude and are applied over large regions usually relatively quickly. Finer details are carefully added during refinement. These details are more precise and are repeated many times over smaller areas. In a sense, blocking and refinement edits work at different scales, both spatially and temporally.

In this paper, we present *3DFlow*, an algorithm for providing continuous summarizations of mesh editing workflows. Figure 3.1 shows at different levels of detail the summaries of several workflows, including low-poly modeling and sculpting sessions using dynamic or subdivision remeshing. *3DFlow* is inspired by two prior works. As in *Video Tapestries* [5], we support continuous levels of summaries to allow arbitrary temporal zooming of the editing sequence. As in *MeshFlow*, we add visual annotations to highlight important changes and summarize the artist's edits.

3DFlow takes as input a sequence of meshes with optional annotations, such as brush strokes, and outputs a continuously summarized mesh sequence with visual annotations. To do so, we first compute *mesh deltas*, one for each input mesh, that describe the changes performed in the current edit. A dependency graph, *depgraph*, is constructed with nodes for each delta and edges representing the spatial and temporal dependencies of the deltas. We then repeatedly contract the edge of least weight, computed by a cost function over the strength and distance of changes in the spatial and temporal dimensions, and merge the corresponding deltas to produce continuously summarized dependency graphs. When only one delta remains, we split the merged deltas in reverse contracting order to produce continuous levels of detail. In the interactive viewer, we highlight changes to the mesh to emphasize the magnitude of the edit and, if supplied, overlay visual annotations to illustrate the artist's edits, such as summarized brush strokes for sculpting.

We tested *3DFlow* using digital sculpting sessions by professional artists obtained with a lightweight software instrumentation, the polygonal modeling sessions from *MeshFlow*, and committed snapshots from movie and tutorial production files [31, 78, 10]. The sculpting artists modeled a variety of organic models, from detailed heads to full bodies, with different workflows based on their personal preference, and using uniform subdivision remeshing or adaptive, dynamic remeshing. The length of sequences generated from instrumented software varied from several hundred to a few thousand individual edits, while those generated from production repositories varied from about ten to a couple hundred. We found that *3DFlow* worked well across all the datasets tested. We refer the reader to the supplemental video for a comparison between *3DFlow* summaries and the fast-forwarded original sequence. We release all workflow data as well as code for both *3DFlow* and our instrumentation as supplemental material, so that artists can take advantage of our algorithm in their daily work and so that other researchers have datasets readily available to test other approaches.

3.3 Related Work

Workflow Visualization. As software packages for image and 3D scene creation become more complicated, both developers and users benefit from understanding common workflows. Developers can optimize the user interface for particular usage scenarios, as proposed by Terry et al. [75] in the case of image editing. In a similar context, Kong et al. [41] presented to users a corpus of workflows at three levels of granularity in order to understand how the users compared the workflows and which granularity was most preferred. Software users learn by studying the workflows of others through tutorials and teaching tools. For example, GamiCAD [48] is an AutoCAD tutorial system for teaching first time users commonly used tools and workflow patterns. Matejka et al. [52] proposes an algorithm and user interface that present command recommendations to the user based on history of command usage. Grossman et al. [34] and VisTrails [79] present systems with which users can explore the provenance of how images or 3D models were constructed. Nakamura and Igarashi [55] present a system for visualizing user operation history with annotations. Nonlinear Revision Control for Images [21] visualizes the workflow of artists manipulating images with a focus on the non-linear relationships between operations induced by their spatial and semantic overlap. More recently, a few papers have shown complementary methods of visualizing workflows. MeshGit (chapter 4) and 3D Timeline [25] estimate and visualize mesh construction provenance as a sequence of mesh diffs. Chen et al. [20] present a way to assist an artist in choosing viewpoints to showcase their 3D editing workflow.

Video Summaries. Video Tapestries [5] summarizes a video sequence into a multi-scale tapestry with the ability to continuously zoom into the tapestry to expose fine temporal detail. This feature allows the summary visualization to adapt to the changes in the sequence as well as the user's preference, rather than forcing the summarized data to fit arbitrarily chosen intervals which may produce unintuitive results. We adopt a similar framework for summarizing workflows.

Polygonal Modeling Summaries. Most similar to our work, MeshFlow (chapter 2) provides summaries of mesh construction sequences by hierarchically clustering the steps in the sequence. Two types of visual annotations are used to indicate the operations performed by the artist that were clustered: highlighting changed elements and overlaying visual annotations to indicate types of change. For example, when a face extrusion followed by vertex movements are clustered together, the individual operations are still visible to the user by highlighting the moved vertices, coloring the newly created face, and drawing an arrow to indicate direction of extrusion. While this work takes inspiration from MeshFlow, *3DFlow* significantly differs in the approach to summarization and addresses key limitations of their work. Specifically, *3DFlow* provides continuous summarization of the workflows based on a cost function over edit strength and distance, where MeshFlow uses a fixed set of rules based on editing patterns. We performed n-gram analyses on the digital sculpting workflows (available in supplemental materials), but the results did not yield a clear set of summarization rules. We believe that MeshFlow-type summarization is not possible on digital sculpting workflows due to the vastly different editing patterns and that a single sculpting tool can produce widely different effects. Moreover, because *3DFlow* uses a cost function, the input to the summarization algorithm does not require tightly-instrumented editing software. As a final point of difference, MeshFlow summarizes the workflow linearly with respect to time, but *3DFlow* summarize over two dimensions (spatial and temporal) to allow for temporal reordering, producing more succinct summaries.

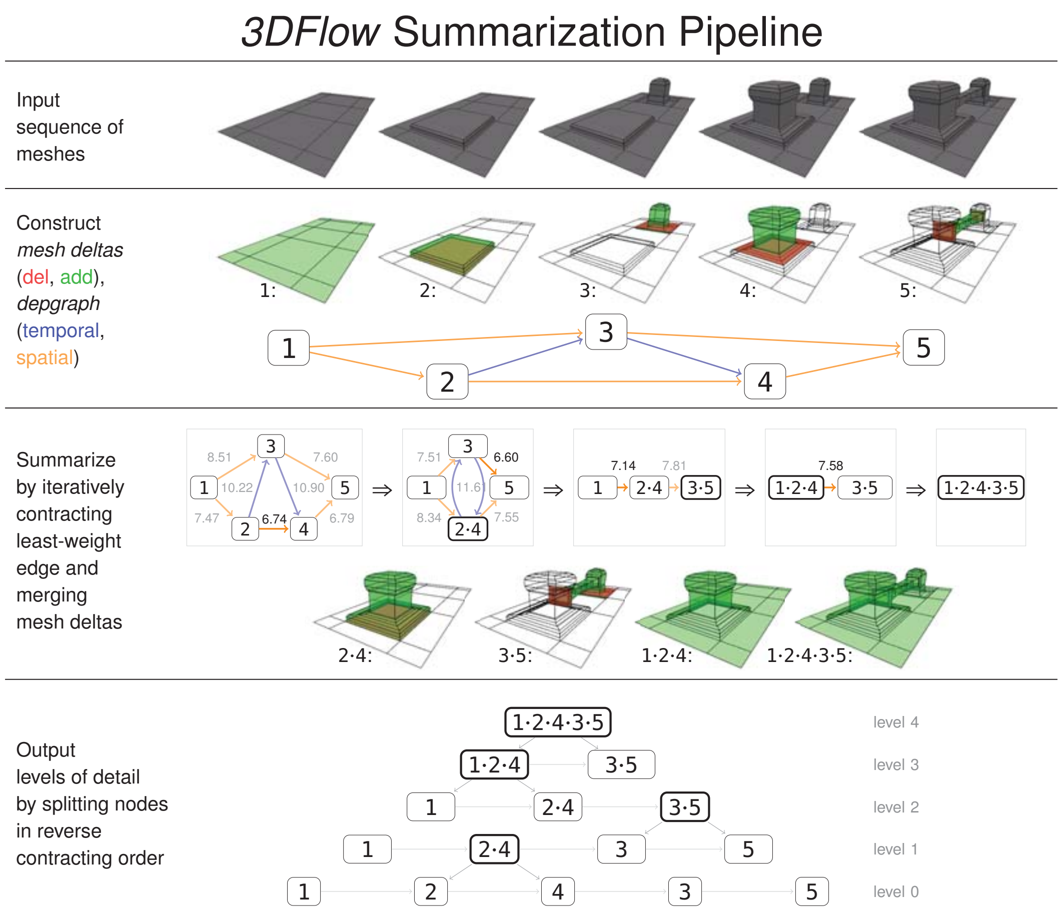

3DFlow Summarization Pipeline
Input sequence of meshes
Construct mesh deltas (del, add), depgraph (temporal, spatial)
1: 2: 3: 4: 5:
1 2 3 4 5
Summarize by iteratively contracting least-weight edge and merging mesh deltas
8.51 7.60 10.22 10.90 7.47 6.74 6.79
7.51 6.60 11.6 8.34 7.55
7.14 7.81
7.58
1·2·4·3·5
2·4: 3·5: 1·2·4: 1·2·4·3·5:
Output levels of detail by splitting nodes in reverse contracting order
level 4
level 3
level 2
level 1
level 0

Figure 3.2: *The input is a sequence of meshes. In this example, each mesh is a single component and was created by performing a series of extrusions. Mesh deltas are constructed for each snapshot to find which faces are deleted (red) from and which are added (green) to the previous snapshot. These deltas capture any modification to the mesh, including translating vertices, creating new geometry, and subdividing the mesh. A dependency graph (depgraph) is created to capture temporal (blue) and spatial (orange) dependencies with a node for each delta and a directed edge for each dependence. For example, delta 4 deletes a face that is created in delta 2, so the node corresponding to delta 4 is spatially dependent on the node of delta 2. The node of delta 3 is temporally dependent on the node of delta 2, because delta 3 immediately follows delta 2 in the original sequence. Every edge is weighted by the cost of merging the mesh deltas corresponding to the two nodes of the edge. We iteratively contract the least-weighted edge and merge the mesh deltas corresponding to the two nodes until no edges remain. The final remaining node corresponds to the mesh delta that is equivalent to adding the final mesh of the input sequence. Finally, we iteratively split the node(s) in reverse contracting order, creating continuous levels of details of the sequence as output.*

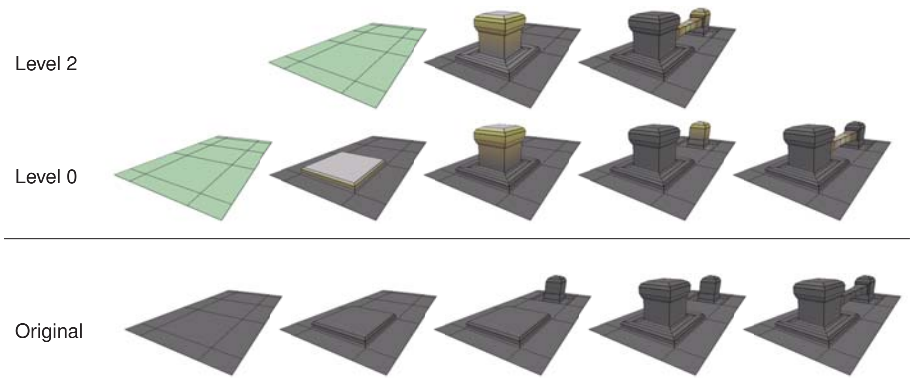


Figure 3.3: *The input to Fig. 3.2 visualized as original sequence and at summarized levels 2 and 0. Notice that the workflow of Level 0 has been temporally reordered from the original. For example, the delta that created the front pillar (4) now immediately follows the delta that created the front base (2).*

Stroke Summaries. When viewing a summary of the sculpting sequence, the artist's strokes are helpful for understanding how the artist worked. But for heavily summarized sequence, the presence of all strokes obscures the object shape and remains too cluttered to provide a high level intuition. Recent work has presented ways to visualize large numbers of edges in a dense graph and to cluster artist strokes in order to provide a high-level overview of the underlying data. Holten and van Wijk [36] show how a force-based system can organize edges in a graph visualization into bundles, which reduces the clutter and exposes underlying connections that might otherwise be obscured. When applied to our brush stroke data, we found that the artist's strokes get organized into patterns that suggest workflows not present in the original sequence. More recently, Orbay and Kara [59] propose a method of beautifying design sketches by first clustering them and then fitting curves to the strokes. Their approach requires training of the clustering method and assumes that each stroke contributes directly to the final sketch. With our data, however, we found that the sculpting strokes affect the final result indirectly. For example, the smooth scultping tool, used to smooth out abrupt features in the mesh, is typically used in a highly unstructured way, where the artist simply paints over a region they wish to smooth. *3DFlow* de-clutters stroke display by providing continuous filtering of strokes based on the strength of the underlying edit.

3.4 Sequence Summarization

The input to *3DFlow* is a sequence of mesh snapshots along with any associated software or edit information such as artist viewing orientation or sculpting stroke data. A sequence can be created in several ways by saving snapshots of the mesh

- after every change using instrumented software,
- periodically (e.g. every 5 minutes), or
- after every logical group of changes as is done during normal creation workflows or with repository commits.

Note that the associated edit information is not required for summarization, as it is only used to overlay optional visual annotations to the sequence visualization.

The following subsections describe the summarization pipeline in detail. Figure 3.2 presents an intuitive overview of this section using a simple example input sequence.

3.4.1 Constructing *Mesh Deltas*

First we convert the spatially normalized sequence of mesh snapshots into a sequence of mesh differences, which we call *mesh deltas*. We normalize the spatial dimension of the sequence by scaling all the meshes so the union of all bounding boxes fits in a unit cube. Each delta tracks the spatial changes and the temporal range the delta covers, which is initially a single snapshot of the sequence. More specifically we store in each delta three sets: a set of deleted faces, a set of added faces, and a set of the original

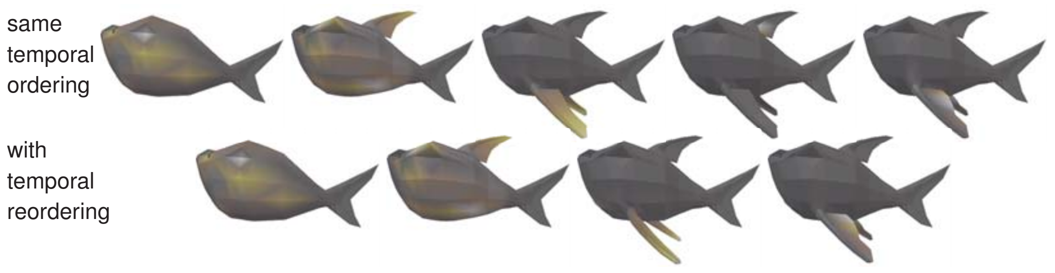


Figure 3.4: *Temporally reordering edits of shark sequence. The artist first created the dorsal fin and then worked on the pectoral fins. The latter work was interrupted by a single change to dorsal fin. The top row shows a summary of the edits using the same temporal ordering of the original input. The summary shown in bottom row is more succinct, because the sequence is allowed to be temporally reordered so the single edit can be summarized with the other edits to the dorsal fin and not interrupt the pectoral fin edits.*

snapshot indices that the delta covers. Note that every mesh in the original sequence can be perfectly reconstructed by successively applying the sequence of deltas in the same temporal order and then inversely rescaling by the normalization factor.

We use a simple rule to build a mesh delta between two subsequent snapshots in a sequence: a face in the former snapshot that also exists in exactly the same position in the latter is considered unchanged; all other faces in former snapshot are *deleted*, and all other faces in latter are *added*. Under this rule, a transformed face is represented as a deletion of the face in the old position and an addition of the face in the new position. Despite its simplicity, this simple mesh delta creation rule works surprisingly well. Furthermore, faces do not need to be matched and tracked but only determined to be left unchanged, deleted, or added, which is inexpensive to compute and handles all types of mesh edits, including subdivision.

The changes of two mesh deltas can be *merged* into a single mesh delta. The merged mesh delta is constructed by computing the unions of corresponding sets from the two mesh deltas. Because the former mesh delta can add a face that is deleted by the latter, we subtract from both the union of added faces and the union of deleted faces the faces that are in the intersection of the two deltas. For example in Fig. 3.2, delta 4 deletes a face that is created in delta 2. When constructing the merged delta 2·4, this face is removed from both unions of faces.

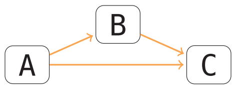
Merging two mesh deltas effectively summarizes in one delta the effects of the two individual mesh deltas. We summarize the sequence into continuous levels of details by iteratively merging mesh deltas.

3.4.2 Constructing a *depgraph*

A key observation is that two temporally subsequent mesh deltas may not spatially overlap, where the intersection of the set of the added faces in the former mesh delta and the set of deleted faces of the latter is empty. This implies that although one mesh delta may temporally follow another (having been performed by the artist subsequently), it is not necessary that the deltas are merged in the same temporal order. For example, Figure 3.4 shows two summaries of the construction of shark fins. The artist first creates the dorsal fin and then begins working on the pectoral fins, but the pectoral fin workflow is interrupted by a single, spatially disconnected edit on the dorsal fin. The summary in the top row maintains the original temporal order and therefore contains the single interrupting edit. By temporally reordering the edits so the single, interruptive dorsal fin edit is summarized with the other dorsal fin edits, the bottom summary is much more intuitive and succinct.

While temporally reordering is useful, it is important to maintain spatial dependence of the mesh deltas. For example, if delta B deletes a face added by A, then temporally reordering B to be before A should not be allowed.

We build a dependency graph, *depgraph*, that captures and enforces the *temporal dependence* and *spatial dependence* of the mesh deltas. A node exists for each mesh delta, and a directed edge exists between a pair of nodes if one node depends on the other. We color the edges by the type of dependence. In order to simplify the depgraph and make summarization faster, we remove temporal edges between nodes that are also spatially dependent, and we remove any edge between two nodes that are also indirectly spatially dependent. For an example of the latter, the depgraph below shows that delta `C` depends both directly and indirectly on `A`. We can remove the `A` → `C` edge and therefore simplify the depgraph without changing the spatial dependencies.



It is important to note that although we maintain spatial dependence, temporal dependence is still a critical data point to maintain. This note becomes obvious with workflows that create spatially disconnected meshes. Without temporal dependence, the depgraph would contain disconnected subgraphs. Although two disconnected meshes are spatially independent, one of the meshes may have influence over the changes of the other. For example, in order to get the shape and proportions correct when working on the eye socket area of a face mesh, the artist may insert a sphere representing the eye. Although this eye mesh is spatially independent from the rest of the mesh, its addition heavily influences the shaping of the face.

3.4.3 Summarizing a depgraph

We summarize a depgraph by contracting one of the edges in the graph and merging the mesh deltas corresponding to the nodes of the edge. The choice of which edge to contract (or which deltas to merge) affects the summary. For *3DFlow*, we motivate our choice with two intuitive and straightforward guidelines that apply to the temporal and spatial dimensions of the sequence:

- A merged delta should not contain too much change.
- A merged delta should not contain edits that are too far apart.

Choosing to merge deltas with strong changes might lose too many details in the summary. Choosing to merge distant deltas may divide the focus of the summary.

From these guidelines, we derive a cost function C for merging a pair of deltas `A` and `B` as a weighted sum of four terms, reflecting the two guidelines for each dimension of the data (spatial and temporal). We use the cost function to determine which edge to contract in the depgraphin order to create a summary. Note that in this notation, each delta may be the result of a previous merge of deltas. The merging cost function is defined as:

$$C(\mathtt{A},\mathtt{B}) = \underbrace{w_0 S_t + w_1 D_t}_{\text{temporal}} + \underbrace{w_2 S_x + w_3 D_x}_{\text{spatial}} \tag{3.1}$$

where S_t, D_t are temporal strength and distance costs and S_x, D_x are spatial strength and distance costs. Formally these individual costs are defined as:

$$S_t = \frac{|\Delta_t(\mathtt{A})| + |\Delta_t(\mathtt{B})|}{\operatorname{avg}|\Delta_t|} \tag{3.2}$$

$$D_t = \min_{a,b \in \Delta_t(\mathtt{A}) \times \Delta_t(\mathtt{B})} \frac{|a-b|-1}{\operatorname{avg}|\Delta_t|} \tag{3.3}$$

$$S_x = \frac{|\operatorname{area}[\Delta_x^+(\mathtt{A}\cdot\mathtt{B})] - \operatorname{area}[\Delta_x^-(\mathtt{A}\cdot\mathtt{B})]|}{\max(\operatorname{area}[\Delta_x^+(\mathtt{A}\cdot\mathtt{B})], \operatorname{area}[\Delta_x^-(\mathtt{A}\cdot\mathtt{B})])} \tag{3.4}$$

$$D_x = \min_{u,v \in \Delta_x(\mathtt{A}) \times \Delta_x(\mathtt{B})} \text{min-dist}(u,v) \tag{3.5}$$

where $\Delta_t(\mathtt{A})$ is the set of original delta indices covered by delta `A`, $\Delta_x^+(\mathtt{A})$ is the set of faces added by `A`, $\Delta_x^-(\mathtt{A})$ the set of faces deleted by `A`, $\Delta_x(\mathtt{A})$ the set of faces either added or deleted by `A`, the dot operator $(\cdot)$ indicates a merging of deltas, $\operatorname{avg}|\Delta_t|$ computes the average size of snapshot indices sets for the deltas in the depgraph, area is a function that returns the total surface area for a given set of faces, and min-dist is a function that returns the minimum Euclidean distance between the given faces.

The temporal strength term, S_t, is the total number of original snapshots covered by merging deltas `A` and `B`. The temporal distance term, D_t, is defined as the minimum temporal distance between the `A` and `B`. This term is computed as the minimum absolute difference between all snapshot indices of `A` and of `B` minus one. For example, if delta `A` covers snapshot 1 and `B` covers snapshots 2 and 4, the temporal distance cost of merging `A` and `B` is 0. Both of the temporal terms are regularized by the average number of snapshots covered by the deltas to prevent the temporal terms from dominating the cost function.

The spatial strength term, S_x, is the absolute net change in surface area after merging both `A` and `B` regularized by dividing by either the net added surface area or the net deleted surface area, whichever is larger. The denominator regularizes spatial changes to be relative to the size of region affected. In other words, spatial changes that are small in the absolute sense are relatively large if they affect a small region, and large spatial changes that affect large regions may be relatively small. The spatial distance term, D_x, is the minimum Euclidean distance between the added and deleted faces of `A` and the added and deleted faces of `B`. Note that the spatial distance term is already regularized when the input was processed to fit in a unit cube.

The four terms of equation 3.1 address the two guidelines mentioned earlier across both dimensions of the data. Each of the terms are linearly weighted to emphasize different types of clustering. For example, setting w_0 to 1 and the remaining weights to 0 will allow for hierarchical uniform clustering. We experimentally found the weights 2, 1, 4, and 14 (respectively) work well to give intuitive results across all shown datasets, including the polygonal modeling workflows of MeshFlow and MeshGit. All figures in this paper and the supplemental materials use these weights.

We consecutively summarize the depgraph, recording the order of edges we contract, until only one node remains. The delta corresponding to the remaining node covers all of the original deltas (possibly reordered) and adds all of the faces of the final mesh. As a note, to help in presenting the most intuitive summaries to the viewer at every level of detail, the initial mesh (e.g. cube, bust, etc.) of the sculpting workflows is held out from being merged until only two nodes remain.

3.4.4 Outputting Levels of Detail

We create the highest summary level as a single delta, the delta corresponding to the single remaining node. This single node is then split into two nodes according to the last edge contraction performed during summarization. Note that the contracted edge encoded the dependence of the nodes, and we maintain this dependence by placing the dependent node temporally after the other node. The corresponding deltas of these two nodes define the second highest summary level. Now, we repeatedly split the nodes in reversed order of edge contraction to produce continuous levels of detail. Reconstructing the deltas in this manner produces linear, but also hierarchical, levels of detail, similar to the levels produced by MeshFlow.

3.4.5 Discussion

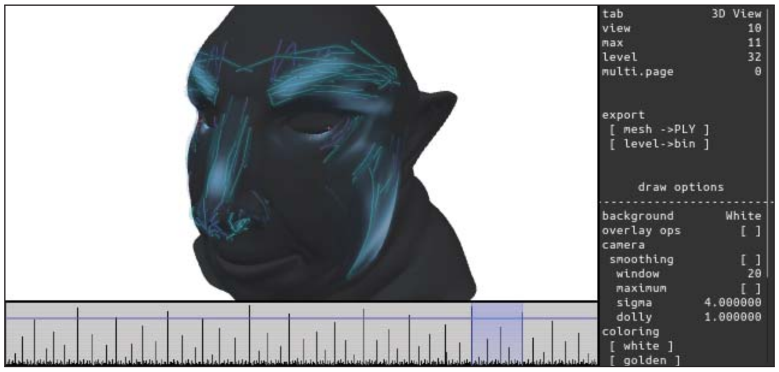
We chose to define our cost function using surface area of deltas to measure shape differences since, compared to other metrics (see [63, 71] for a review), it is efficient to compute, it is well-defined even on non-manifold meshes or meshes with holes, and it does not require a registration between two meshes beyond finding which faces have been altered. Despite the simplicity of the terms introduced above, we found that the cost function worked well over a range of sculpting and polygonal modeling datasets.

Furthermore, we tested more expensive cost functions (e.g. mean curvature, volume delta, hausdorff distance, distance between corresponding points), and found that they did not improve upon the results enough to warrant the additional computation. We leave further investigations to future work.

Unlike MeshFlow, we do not consider the category or name of the edit operation or even editing patterns when clustering. We did perform n-gram analysis on the digital sculpting workflows (see supplemental materials), but it is unclear how to construct clustering patterns that would produce intuitive results. Furthermore, by only considering the edited region and not the name or category of edit operation, *3DFlow* can summarize more general workflows such as those where instrumentation was not used. For an example see the supplemental material where we used as input to *3DFlow* every version of the character Sintel from the Subversion repository of the open movie Sintel [10].

Limitations. While we believe that equation 3.1 performs well in regards to our guidelines, it does not capture the semantic of an edit. For example, it might make sense to cluster together edits that work on the eyes or those that add wrinkles across the face. The formulation above does not infer any semantical meaning from the edit itself or from the region being changed.

Finally, although the spatial distance computations are highly parallelizable and many other computations can be cached, the nature of greedily choosing a single edge to collapse in the depgraph imposes sequential constraint on the algorithm. We focused on computing accurate values or highly-accurate approximations when possible, and we leave further optimization for future work.



```
tab              3D View
view                  10
max                   11
level                 32
multi.page             0


export
 [ mesh ->PLY ]
 [ level->bin ]


     draw options
------------------------
background         White
overlay ops          [ ]
camera
 smoothing           [ ]
  window              20
  maximum            [ ]
  sigma         4.000000
  dolly         1.000000
coloring
 [ white ]
 [ golden ]
```

Figure 3.5: *User interface for 3DFlow. The mesh is shown at the top-left for the selected delta and level of detail with surface changes highlighted and sculpting stroke annotations visualized. The timeline at the bottom-left visualizes the deltas at different levels of detail, from every original delta (bottom) to the highest summary (top). The blue highlight indicates the selected level of detail, selected delta, and the deltas of lower levels of detail that are covered by the selected delta. Visualization settings are shown on the right.*

3.5 Visualizations

In this section, we describe some of the ways we visualize different features of the data. We also discuss a few ways for a viewer to interact with the data.

Basic User Interface. Figure 3.5 shows the user interface. To maintain simplicity, we use a basic layout that is similar to a simple video player. At the top-left is the main 3D view, where the mesh is seen at the selected time and level of detail. Regions of the mesh that are altered by the selected delta are highlighted. The timeline at the bottom-right acts much like a scrub bar in a video player. The vertical axis of the timeline is the level of detail, with highest summary at the top and greatest details (deltas of original sequence) at the bottom. Black vertical lines indicate where each delta begins and ends. The blue vertical bar indicates the coverage of the selected delta, and the blue horizontal bar indicates the selected level of detail. The visualization options on the right allow the viewer to control how the mesh is rendered.

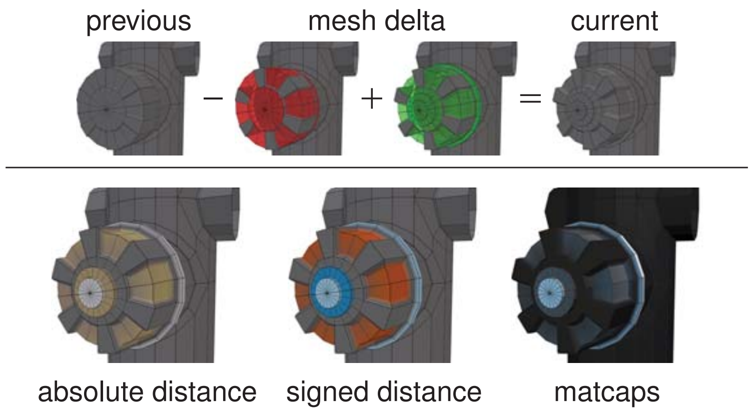


Figure 3.6: *Emphasizing surface changes in mesh delta. Applying the mesh delta (top-middle) to the previous mesh (top-left) results in the current mesh (top-right). The mesh delta covers 31 deltas in the original sequence. The bottom row shows three different ways to highlight and emphasize the magnitude and direction of changes to the surface. See Section 3.5 for more details.*

While *3DFlow* generates continuous levels of detail from every delta down to a single delta, by default we simplify the user interface to show only a subset of the levels. We choose the levels that are at a log-scale of the original deltas (all, half, quarter, etc.), and then we add the levels with 2–20 deltas and the levels with odd number of deltas in the 20–50 range. This simplification can be turned off.

Highlighting Changes. The changes in a mesh delta are emphasized by highlighting the added faces, where the *magnitude* of the change modulates the visual strength of the highlight. For each delta, we approximate a magnitude of change for each vertex of an added face as the minimum distance between the vertex to the surface defined by the deleted faces. If in a delta no faces were deleted, then all of the vertices of the added faces are marked as *added*. This can happen, for example, whenever the artist creates new disconnected geometry to the mesh. We visualize added geometry in green and modified geometry by using it as a mixing value. To adapt highlighting for edits that are globally large (e.g. creating a large appendage) and for edits that are globally small but locally large (e.g. adding wrinkles), *3DFlow* can individually rescale

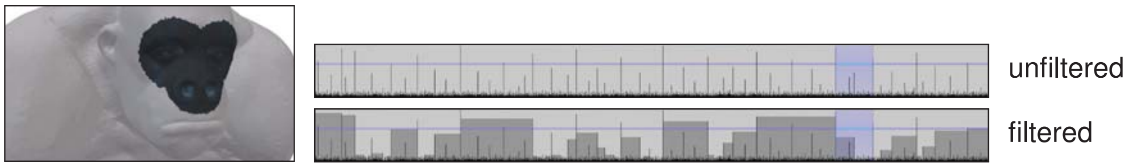


Figure 3.7: *Spatial filtering on gorilla sequence. The mesh on the left is partially deemphasized to indicate the selected regions. The timelines on the right show without (top) and with (bottom) filtering. The deltas that do not modify the selected region are darkened and are not viewable.*

the magnitudes by the local or global maximum.

3DFlow offers several highlighting options for the vertices. Figure 3.6 demonstrates a few different possible visualizations which are briefly explained below. One option is to linearly map the magnitude to a color gradient, where unchanged vertices are colored a neutral gray, moderately changed vertices are yellow, and vertices with strong magnitude of change are white. A multi-color gradient provides better resolution to help resolve strong changes from minor changes. Another option is choosing different color gradients based on the *sign* of change. Specifically, the vertex has a *positive* change if it was moved "outside" the deleted surface and *negative* if moved "inside", where sidedness is determined by the surface normal. Positive changes are colored blue, while negative changes are colored orange. This option of highlighting visualizes the approximate magnitude and direction the vertex was moved, giving a sense of the change in volume. Lastly, rather than mapping the magnitude to a color gradient, the magnitude can influence a mixing value between two *matcaps*. Matcaps simulate complex material and lighting setups and are often used to help sculpting artists focus on certain characteristics of the mesh, such as the contour and overall shape or the high-frequency details and creases.

Spatial Filtering. In order to help the viewer find deltas that modify particular spatial regions, *3DFlow* provides spatial filtering. When the viewer clicks on the mesh, every face in the entire sequence that is within a given radius of the point on the mesh is selected. Unselected regions of the mesh are deemphasized in the main 3D view by desaturation and brightening. All deltas that do not affect a selected face is made unviewable and is darkened in the timeline, indicating to the viewer when the selected region was modified. Figure 3.7 show the timeline filtered to the deltas that modify the face of the gorilla.

Visualizing Sculpting Annotations. While highlighting indicates how much regions of the mesh have changed, it is not very descriptive of which sculpting tool the artist used or how the tool was used. When tool usage metadata is provided, *3DFlow* can visualize the artist's tool usage by overlaying visual annotations. In *3DFlow*, we visualize the artist's sculpting strokes as lines drawn over the mesh. Because the sculpting stroke may fall inside or behind the mesh, we render the strokes in two passes: once with a thick, transparent line without performing depth tests, and then another with a thin, opaque line with depth testing. The first pass allows the viewer to see strokes that are obscured by the mesh but without adding too much clutter. Strokes are colored by brush type: pulling in blue, smoothing in cyan, creasing in orange, and grabbing or nudging in pink. Although we visualize only the sculpting strokes, visualizing other types of edits, such as extrude edge and merge vertices, can be trivially added in *3DFlow*.

Filtering Annotations. As the number of covered deltas increases, visualizing all of the tool annotations can obscure the view of the mesh and may overwhelm the viewer. Similarly to providing levels of detail and summary of mesh deltas, *3DFlow* provides continuous levels of detail and summary for tool annotations through filtering. Filtering removes the annotations that change the mesh the least. The filtering can be continuously adjusted to show any number of annotations from all down to none.

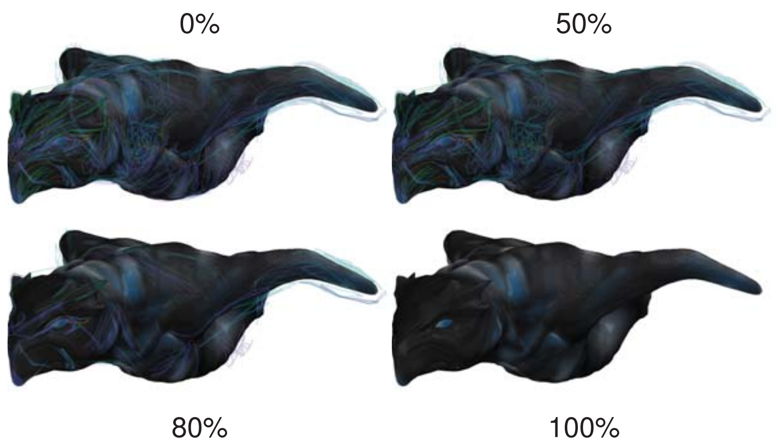


Figure 3.8: *Filtering annotations at 0%, 50%, 80%, and 100%. The mesh is heavily obscured when visualizing the sculpting stroke annotations of all 343 merged deltas (top-left). With the annotations sorted by a computed weight of change, 3DFlow provides continuous filtering to show anywhere from all annotations (0%) to none (100%).*

Each edit annotation is assigned a weight equal to equation 3.4 of the corresponding delta. The annotations are sorted by their weight, and *3DFlow* visualizes only the annotations with an order that is above a user-specified threshold. Figure 3.8 shows the effect of filtering tool annotations at varying levels, where 0% filtering shows all tool annotations, 50% shows only half of the annotations, and 100% shows none.

We considered two clutter-reducing alternatives to sculpting stroke annotation filtering: determine a representative through spatial clustering or performing edge-bundling [36]. Unfortunately we found that these alternatives were of little help for uncorrelated tool usage or suggested tool usage patterns that were not representative of the artist's workflow, as in the case of spatially-close sets of correlated edits.

Other Visualization Options. We refer the reader to the supplemental material for a demonstration of other visualization options. These include: render the summarized workflow using external software; render with a mirror effect to see edits on front- and back-side of mesh at the same time; smoothly interpolate or warp the surface to simulate the artist's summarized work; and center-on and zoom-into the region of the

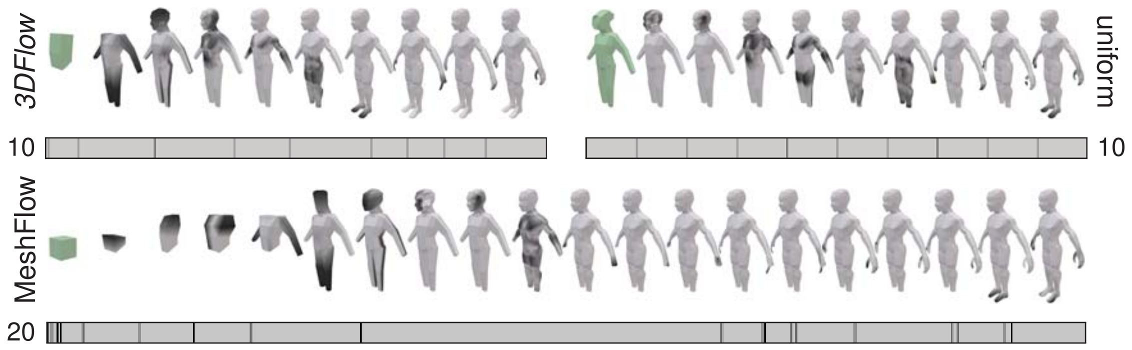


Figure 3.9: *Comparing summaries produced by 3DFlow (top-left), uniform intervals (top-right), and MeshFlow (bottom). Changes are highlighted in black, and the timelines show the coverage of deltas for each summary. While MeshFlow can only summarize the biped sequence down to 20 steps, 3DFlow and uniform intervals can provide continuous summarization (10). See Section 3.6 for detailed analysis of this figure.*

mesh that are edited.

3.6 Results

In this section we report about the input workflows and briefly discuss the results.

Input Workflows. We tested *3DFlow* on a variety of mesh editing workflows, shown throughout the paper and in supplemental material. Source code and all datasets are available in supplemental material. Table 4.1 summarizes statistics for all of the input workflows.

Our sculpting data was obtained by two professional artists with different working styles. One artist has a stronger tendency to explore while editing, making strong changes often throughout the sequence. The other artist prefers a more structured blocking followed by refinement approach. Both artists sculpted using both subdivision and dynamic remeshing to control mesh resolution. Workflow lengths in terms of the number of sculpting edits varies from several hundreds to a few thousand. The initial meshes consisted of a cube, a generic human bust, and a full-body human basemesh.

	model	fig.	mesh deltas	added faces	record type	process time
subdivision sculpting	*ogre*	3.5	1459	1,660,475	i	1:26
	merman	3.10	2218	2,171,310	i	3:40
	sage	3.10	1686	1,961,133	i	2:19
	engineer	3.10	863	2,919,865	i	2:54
	elder		2958	1,500,632	i	3:01
	alien	3.1	2118	6,094,173	i	8:49
	man	3.10	1459	1,953,859	i	3:03
	fighter	3.10	1532	1,156,686	i	2:06
dynamic sculpting	*gargoyle*	3.8	819	1,090,882	i	0:33
	monster	3.1	797	1,389,906	i	0:47
	elf		4125	4,791,845	i	2:46
	gorilla	3.1	2482	4,241,528	i	3:32
	explorer		1699	3,416,354	i	2:21
polygonal modeling	*helmet*	3.1	1321	17,579	i	0:05
	hydrant	3.6	691	49,892	i	0:04
	robot		1810	139,527	i	0:15
	shark	3.4	1457	19,177	i	0:06
	biped	3.9	1267	18,162	i	0:05
	durano	3.1	11	7,165	c	0:01
	creature		123	280,338	c	0:14
	sintel		210	2,948,611	c	2:11

Table 3.1: *Statistics of input workflows. The first eight workflows are digital sculpting sessions that used subdivision surface rules to generate higher resolution meshes. The middle five workflows are sculpting sessions that used dynamic remeshing techniques. The last eight workflows were constructed using polygonal modeling techniques. The added faces column reports the number of unique faces added by the original deltas. The record type column reports whether the workflow was created using instrumented software (i) or by committing versions (c). The final column indicates how much processing time (mm:ss) was needed to summarize the workflow. All meshes are shown in supplemental materials.*

3DFlow was able to summarize well all sculpting scenarios for both artists, essentially adapting to different workflow styles.

The sculpting artists used an instrumented version of Blender that saves a copy of the mesh along with any associated tool usage information after each change. The summarization process is performed off-line in order to keep the mesh editing interface fluid for the artists.

The helmet, hydrant, robot, shark, and biped polygonal modeling workflows were imported from the MeshFlow dataset, which is publicly available online. The durano and creature workflows are from two Blender Open Movie Workshop DVDs, Venom's Lab! [78] and Creature Factory [31], respectively. The sintel [10] workflow is from the Subversion repository of the open movie Sintel [65] available online. The *3DFlow* workflows for durano, creature, and sintel were created directly from the committed files without processing or manual filtering.

We tested *3DFlow* on a quad-core 2.93GHz Intel Core i7 with 16GB RAM and an ATI Radeon HD 5750 graphics card. The rightmost column of table 4.1 indicates the time to summarize each workflow. The alien workflow has the longest summarization time at 8 minutes 49 seconds, and the average time for all 21 workflows is just under 2 minutes. We leave further optimization to future work.

Discussion. We compare results of summarizing the biped workflow using *3DFlow*, uniform intervals (similar to a timelapse), and MeshFlow in Figure 3.9. Due to having continuous summarization, *3DFlow* and uniform intervals can summarize the workflow anywhere down to a single step, while MeshFlow can only summarize to discrete steps because of using fixed clustering rules. In this example, we summarized the workflow to ten steps for *3DFlow* and uniform intervals and twenty steps for MeshFlow (the minimum possible number of steps for this data). The timelines below the rows of meshes report the coverage of deltas for each workflow summary. Notice that *3DFlow* summarizes changes into small, localized groups, such as blocked-out figure, face, upper body, lower body, feet, and hands. On the other hand, uniform intervals and MeshFlow summaries contain merged edits that are spatially distant (e.g. mixing edits to feet and hands) or contain many strong edits (e.g., the first step of uniform summary and the tenth step of MeshFlow). Another important note is that in the original sequence, the hands were created before the feet, but the arms shortened last. With temporal reordering, *3DFlow* summarized together all of the edits to the forearm and hands.

Figure 3.10 show five sculpting workflows that started with a base mesh and used subdivision remeshing. One artist created the merman, engineer, and sage workflows, and the other artist created the alien (also from cube with subdivision; see Figure 3.1), fighter, and man workflows.

We asked the professional artists who authored the sculpting workflows to provide feedback on the results of *3DFlow*. They found the summarizations captured their workflows and the workflows of the other authors quite well, and both agreed that *3DFlow*'s interactive viewer with summarized workflow is a significant improvement over time-lapsed videos. One artist commented, "I've recently finished working on the materials for a sculpting course I'm teaching. Having 3DFlow available would have made it unnecessary to share both the final sculpture and the videos of the process, allowing students to better visualize changes to the mesh." The other artist commented that it is astonishing to see how *3DFlow* breaks down the workflow process.

Future Work. We tested *3DFlow* with a large set of workflows across a variety of techniques. There are several other common and interesting mesh editing workflows that we did not try, including retopologizing and sculpting using Boolean operations. We plan to extend the techniques developed with *3DFlow* to summarize these types of workflows as well as workflows that change the properties of the mesh, such as texturing or rigging, or workflows that modify full-scene data. When summarizing workflows, *3DFlow* does not consider the type nor the technical complexity of the edit operations performed. Further *3DFlow* does not consider the context of edits, e.g. adding wrinkles to forehead versus shaping the eye socket. We plan to investigate these areas in the future.

3.7 Conclusion

We presented *3DFlow*, an algorithm for providing continuous summarizations of mesh editing workflows. *3DFlow* summarizes the input sequence of meshes by constructing a corresponding dependency graph where nodes represent changes to the mesh and edges the spatial and temporal dependence of the edits, iteratively contracting the least-weighted edge according to a cost function until only one node remains, and then splitting the nodes in reverse order into levels of detail. The visualization of the workflow is enhanced by highlighting the changed regions and (optionally) overlaying visual annotations describing the artist's edits. We tested *3DFlow* with a large set of mesh editing workflows from a variety of sources and found *3DFlow* performed well with all. All source code and data is released as open source.

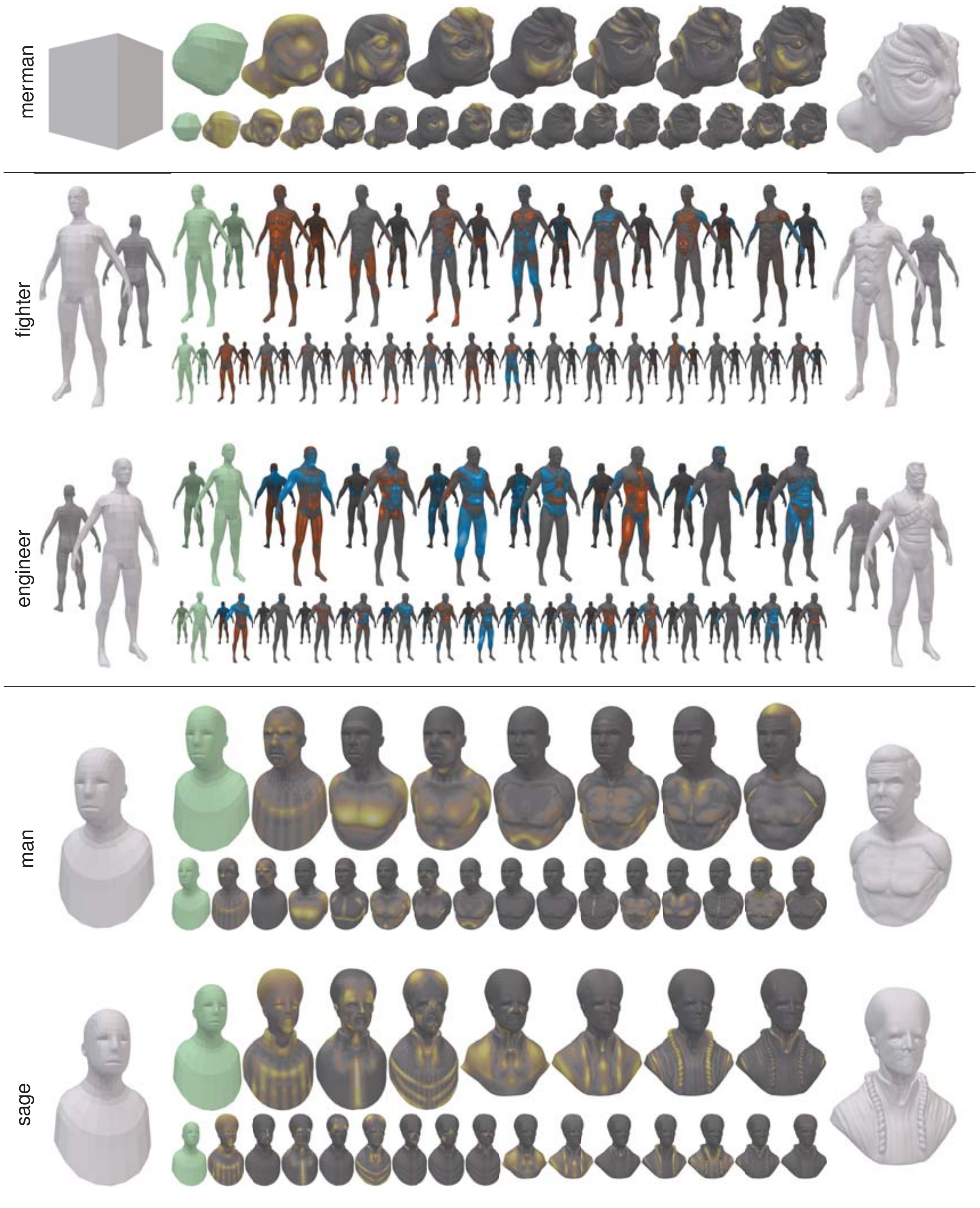


Figure 3.10: *Five sculpting workflows summarized in 8 and 16 steps. These workflows started with a base mesh (left column) and used subdivision remeshing. The initial and final meshes (right column) are shown without highlighting. The fighter and engineer workflows are visualized with a mirror effect to show both sides of the mesh.*

Chapter 4

MeshGit: Diffing and Merging Meshes for Polygonal Modeling

This chapter describes how to approximate the edit operations required to turn one mesh into another, how to visualize these mesh edit operations, and how to merge the edit operations from a single original mesh to two independent derivatives.

4.1 Overview

With MeshFlow (Chapter 2) and 3DFlow (Chapter 3), visualization of the edit operations required tight instrumentation of the 3D modeling software, where every operation the artist performs and its effect is recorded. This level of instrumentation may not always be available. In MeshGit, we aim to approximate the edit operations performed between two saved snapshots or versions of the mesh, similar to committing or checking-in a mesh into a version control system such as SVN or Git. Because we are able to approximate these edits without instrumentation, we can visualize mesh differences between two meshes that may not be exact derivatives of each other.

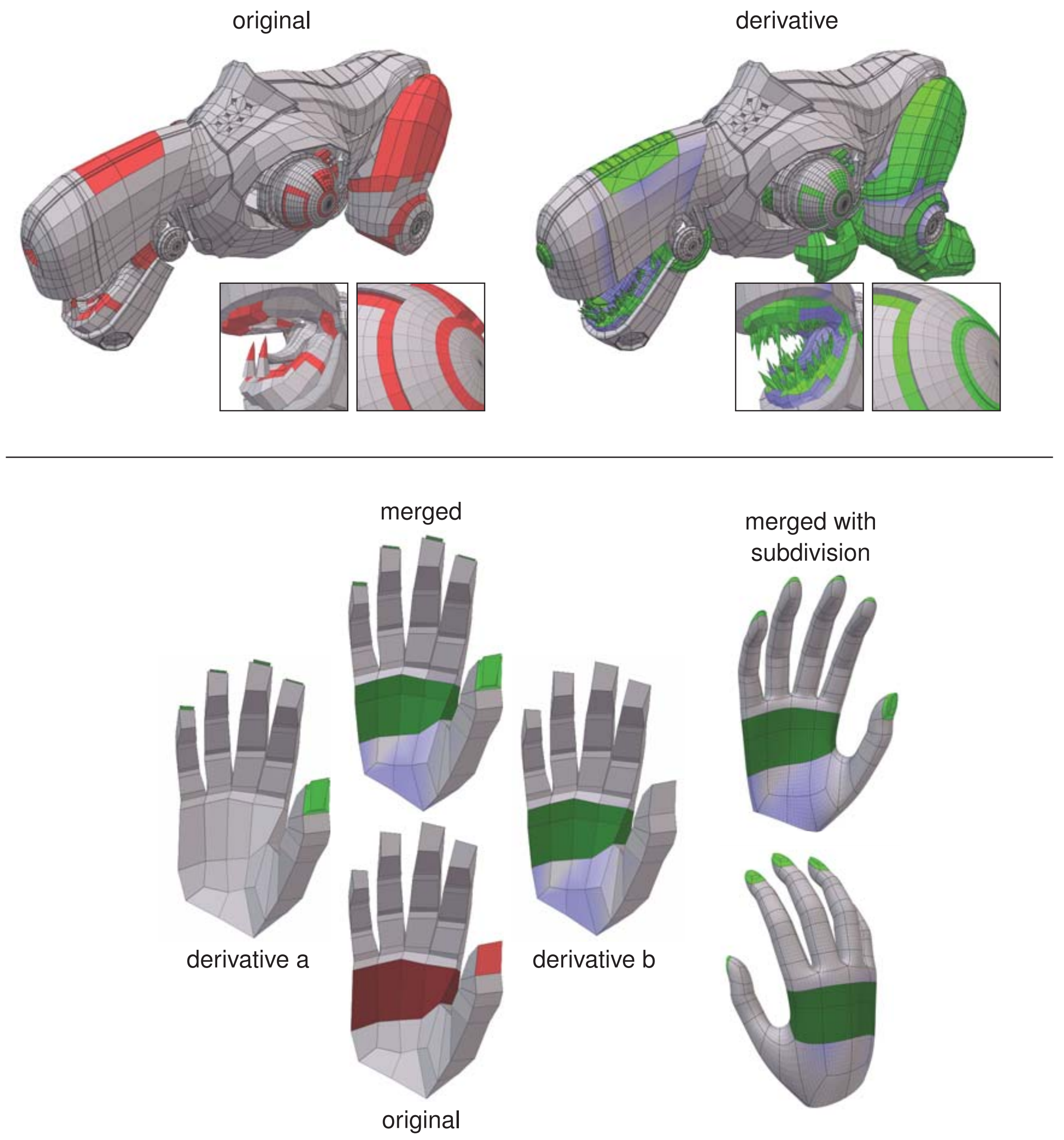


Figure 4.1: *Examples of diffing and merging polygonal meshes done automatically by MeshGit. Top: We visualize changes between two snapshots of the creation of a creature mesh as a two-way diff. The* derivative *mesh contains many changes, including significant changes in adjacency (red/green) and geometry (blue) of the gum line and tongue with many additional teeth (left inset) and an extra edge-loop and inset details on the shoulder ball (right inset). Bottom: We visualize changes performed between an original mesh and two derivatives as a three-way diff.* Derivative a *(left; light colors) adds fingernails, while* derivative b *(right; dark colors) adds an edge-loop across palm with reshaping. MeshGit automatically merges these two sets of non-conflicting edits, shown at the top. We show the merged mesh after applying Catmull-Clark subdivision rules to demonstrate that MeshGit maintains consistent face adjacencies.*

4.2 Introduction

When managing digital files, version control greatly simplifies the work of individuals and is indispensable for collaborative work. Version control systems such as Subversion and Git have a large variety of features. For text files, the features that have the most impact on workflow are the ability to store multiple versions of files, to visually compare, i.e. diff, the content of two revisions, and to merge the changes of two revisions into a final one. For 3D graphics files, version control is commonly used to maintain multiple versions of scene files, but artists are not able to diff and merge most scene data.

We focus on polygonal meshes used in today's subdivision and low-polygon modeling workflows, for which there is no practical approach to diff and merge. Text-based diffs of mesh files are unintuitive, and merging these files often breaks the models. Current common practice for diffing is simply to view meshes side-by-side, and merging is done manually. While this might be sufficient, albeit cumbersome, when a couple of artists are working on a model, version control becomes necessary as the number of artists increases and for crowd-sourcing efforts, just like text editing. Meshes used for subdivision tend to have relatively low face count, and both the geometry of the vertices and adjacencies of the faces have a significant impact on the subdivided mesh. Recent work has shown how to approximately find correspondences in complex meshes [16], and smoothly blend portion of them using remeshing techniques [67]. These algorithms are unfortunately not directly applicable to our problem since we want diffs that captures all differences precisely and robust merges that do not alter the mesh adjacencies. Dobos and Steed [26] recently propose a version control system that works at the granularity of single object components, i.e. at the granularity of singular meshes in a scene graph. We are instead interested in determining differences of elements of each mesh, namely vertices and faces and their adjacency.

MeshGit. We present MeshGit, an algorithm that supports diffing and merging polygonal meshes. shows the results of diffing two versions of a model and an automatic merge of two non-conflicting edits. We take inspiration from text editing tools in both the underlying formalization of the problem and the proposed user workflow (see). Inspired by the string edit distance [46], we introduce the mesh edit distance as a measure of the dissimilarity between meshes. This distance is defined as the minimum cost of matching vertices and faces of one mesh to those of another mesh. The mesh edit distance is related to the maximum common subgraph-isomorphism problem, a problem known to be NP-hard. We propose an iterative greedy algorithm to efficiently approximate the mesh edit distance.

Once the matching from one mesh to another is computed, we translate the found correspondences into a set of mesh transformations that can transform the first mesh into the second. We consider vertex translations, additions, and deletions and face additions and deletions. With this set of transformations, we can easily display a *meaningful* visual difference between the meshes by just showing the modifications to vertices and faces, just like standard diff tools for text editing. For merging, we compute the difference between two versions and the original. We partition the transformations into groups that, when applied individually, respect the mesh adjacencies. This partitioning limits the granularity of the edits in the same way that grouping characters into lines does for text merging. To merge the changes from the two versions, we apply groups of transformations to the original mesh to obtain the merged model. Some groups can be applied automatically, while others are conflicted and require manual resolution. We robustly detect conflicts by determining whether two groups from the different versions modify the same parts of the original, i.e. they intersect on the original. In MeshGit, non-conflicting groups are applied automatically, while for conflicting edits, the user can either choose a version to apply or resolve the conflict manually. We took this approach, as commonly done in text merging, since it is unclear how to merge conflicting transformations in a way that respects the artists' intentions.

MeshGit Uses We evaluate MeshGit for a wide variety of meshes taken from user editing sessions in subdivision modeling workflows. Our tests include meshes that are a mixture of triangles and quads and can have highly regular or irregular adjacencies. We found that MeshGit worked well for all these tested meshes. We choose these types of meshes since they are commonly used by artists today in production environments. To allow readers to use MeshGit in their daily workflows, we include source code and executable in supplemental material.

While MeshGit works well in our context, we do not expect the computed diffs to be as informative in other modeling workflows where mesh adjacencies are not of paramount importance, e.g. free-form sculpting with dynamic topology or smooth shape manipulation with remeshing. In these workflows, artists are only concerned with manipulating geometry, while the system can change mesh adjacency if needed. For example, shows an example of two versions of a mesh obtained with workflows that allow for remeshing. While MeshGit computes correct mesh differences, these are, in our opinion, less informative for artists than just a geometry-only diff. These workflows are out of the scope of MeshGit, and we leave this for future work.

Contributions In summary, this paper proposes a practical framework for diffing and merging polygonal meshes typically used in low-polygon and subdivision surface modeling. MeshGit does this by (1) defining a mesh edit distance and describing a practical algorithm to approximate it, (2) defining a partitioning rule to reduce the granularity of mesh transformation conflicts, and (3) deriving diffing and merging tools for polygonal meshes that support a familiar text-editing-inspired workflow. We believe these are the main contributions of this paper. The remainder of this paper will describe the algorithm, present the diffing and merging tool, and analyze their performance.

4.3 Related Work

Revision Control. Recent work by Dobos and Steed [26] proposes an approach to revision control for 3D models by operating on the nodes of the scene graph. The edits of two different artists can be merged automatically when the edits do not affect the same component, while they need to be manually resolved otherwise. This effectively sets the granularity of supported mesh transformations to the individual components of the graph. This is common practice today, although done manually, as shown in the open source movie Sintel [8]. MeshGit supports arbitrary edits on meshes without explicitly requiring them to be split into components, and can merge the changes onto the same mesh (see Figure 4.1.b.). We leave for future work understanding how these two approaches might complement each other.

Shape Registration. A visual difference between two meshes could also be obtained by performing a partial shape registration of the meshes, and then converting that registration to a set of mesh transformations. Various mesh registration algorithms exist, as reviewed recently by Chang et al. [16]. Some of these methods [17, 13] are variants of iterative closest point [66] that determine piece-wise rigid transformations for different mesh regions and blend between them. In the case of heavily sculpted meshes, these algorithms require too many cuts and transformations to register the shapes. Others use spectral methods [45, 69] to determine a sparse correspondence between two shapes. Sharma et al. [68] uses heat diffusion as descriptors to overcome topological changes with seed-growing and EM stages to build a dense set of correspondences. Typically, these algorithms work by subsampling the mesh geometry since their computational complexity is too high. Zeng et al. [82] propose a hierarchical method to performing dense surface registration by first matching sparse features then building dense correspondences using the sparse features to constrain the search space. Kim et al. [40] propose using a weighted combination of intrinsic maps to estimate correspondence between two meshes. In general, we find that partial shape registration algorithms perform very well for finely tessellated meshes where matching accuracy of mesh adjacencies is not of paramount importance. When applied to our application though, these algorithms either do not scale very well, require the estimation of too many parameters, or are not sensitive enough to adjacency changes to produce precise and meaningful differences for the meshes typically used in subdivision modeling. Furthermore, it remains unclear whether converting these partial matches to transformations is robust for merging. MeshGit formalizes the problem directly by turning mesh matching solutions into mesh transformations that are easy to visualize and robust to merge.

Topology Matching. Eppstein et al. [29] propose an algorithm to match quadrilateral meshes that have been edited by using a matching of unique extraordinary vertices as a seed for a matching-propagation algorithm. Because the proposed algorithm does not take geometry into account, it is robust to posing and sculpting edits. Furthermore, coupled with an initial mesh-reducing technique, the proposed algorithm can solve the topological matching very quickly. However, when applied to the types of edits of the meshes in this paper, we found that the algorithm did not produce an intuitive matching. The limitations of topology matching is due to ignoring the geometry of the mesh. MeshGit strikes a balance between geometry and topology to produce intuitive results.

Graph Edit Distance. By describing a polygonal mesh as a properly defined attributed graph, we can reformulate the problem of determining the changes needed to turn one mesh into another as the problem of turning one graph into another, which is know as the graph edit distance [57]. Bunke [14] shows that computing the graph edit distance is equivalent to the maximum common subgraph-isomorphism problem, a problem know to be NP-hard. Several approximation algorithms have been proposed that differ in the expected properties of the input graph. We refer the reader to a survey by Gao et al. [30] for a recent review. We have experimented with a few of these methods, and found that they do not work well in our problem domain since they either scale poorly with model size or since they approximate too heavily the adjacency costs. For example, Riesen and Bunke [64] propose to approximate the distance computation as a bipartite graph matching problem. In doing so, they approximate heavily the adjacency costs, which we found to be problematic. Cour et al. [23] propose methods based on spectral matching, but we found them to scale poorly with model size and to be generally problematic when the graph spectrum changes. MeshGit introduces an iterative greedy algorithm that takes into account mesh adjacencies well.

Assembly-Based Modeling. Snappaste [67] allows users to create derivative meshes by smoothy blending separate mesh components either created specifically or found automatically by mesh segmentation. Recently, Chaudhuri and Koltun [19] and Chauduri et al. [18] demonstrate the feasibility of constructing 3D models from a large dictionary of model parts. These methods work by remeshing components together, so they inherently do not respect face adjacency in the merged regions. This works well for highly tessellated meshes, but not for meshes typically used in subvision surface modeling where we want to maintain precisely the mesh topology designed by artists.

Instrumenting Software. An alternative approach to provide diff and merge is to consider full software instrumentation to extract the editing operations. VisTrails [79] let the users explore their undos histories. MeshFlow (chapter 2) and 3DFlow (chapter 3) shows rich visual histories of mesh construction by highlighting and visually annotating changes to the mesh. Nonlinear Revision Control for Images [21] demonstrates non-linear image editing, including merging. All these approaches record and take advantage of the exact editing operations an artist is performing. These are semantically richer than the simpler editing operation that MeshGit recovers automatically. At the same time, these methods have the burden of a software instrumentation that is not available in today's software and would not allow artists to work with different softwares on the same meshes. Furthermore, despite having the construction history, it is unclear how to determine a difference between two similar meshes that were constructed independently or where there is no clear common original, such as the meshes in Figure 4.2.

4.4 Mesh Edit Distance

To display meaningful visual differences and provide robust merges, we need to determine which parts of a mesh have changed between revisions, and whether the changes have altered the geometry or adjacency of the mesh elements. Inspired by the string edit distance [46] used in text version control, we formalize this problem as determining the partial correspondence between two meshes by minimizing a cost function we term *mesh edit distance*. In this function, vertices and faces that are unaltered between revisions incur no cost, while we penalize changes in vertex and face geometry and adjacency. Optimizing this function is equivalent to determining a partial matching between two meshes, where vertices and faces are either unchanged, altered (either geometrically or in terms of their adjacency), or added and deleted.

Mesh Edit Distance Given two versions of a mesh M and M', we want to determine which elements of one corresponds to which elements in the other. In our metric, we consider vertices and faces as the mesh elements. An element e of M is matched if it has a corresponding one e' in M', while it is unmatched otherwise. A mesh matching is the set of correspondences O between all elements in M to the elements in M'. The matching is bidirectional and, in general, partial, in that some elements will be unmatched, corresponding to addition and deletion of elements during editing. To choose between the many possible matching, we minimize the *mesh edit distance* $C(O)$, written as the sum of three terms

$$C(O) = C_u(O) + C_g(O) + C_a(O)$$

Unmatched Cost C_u. We penalize unmatched elements, either vertices or faces, by adding a constant cost of 1 for each element. Without this cost, one could simply consider all elements of M as deleted and all elements of M' as added. This can be written as

$$C_u(O) = N_u + N_u'$$

where N_u and N_u' are the number of unmatched elements in M and M' respectively.

Geometric Cost C_g. Matched elements incur two costs. The first captures changes in the geometry of each element, namely its position and normal. In our initial implementation, we consider meshes with attributes, where vertex positions and face normals are given, vertex normals are the average normals of the adjacent faces, and face positions are the average position of adjacent vertices. The geometric cost is given by

$$C_g(O) = \sum_{e \in E} \left[\frac{d(\mathbf{x}_e, \mathbf{x}_{e'})}{d(\mathbf{x}_e, \mathbf{x}_{e'}) + 1} + (1 - \mathbf{n}_e \cdot \mathbf{n}_{e'}) \right]$$

where E is the set of matched elements e in M with corresponding elements e' in M', $\mathbf{x}$ and $\mathbf{n}$ are the position and normal of an element, and d is the Euclidean distance. We only write this term for M since it is identical in M'.

The position term is an increasing, limited function on the Euclidean distance between the elements locations. This favors matching elements of M to close-by elements in M' and has no cost for matching co-located elements. We limit the position term to allow for the matching of distant elements, albeit at a penalty. We also include an orientation term computed as the dot product between the elements' normals to help in cases where many small elements are located close to one another. To make the position and orientation terms comparable, we normalize both meshes so the average edge over both meshes has unit length. By including position and orientation costs for vertices and faces, MeshGit can compute directly a cost for matching two elements.

It should be noted that our implementation assumes that vertices are defined with respect to the same coordinate system during editing. We believe this is an acceptable assumption since this is common practice in mesh modeling as gross transformations and posing of the mesh are generally stored as a separate transformation matrix or armature by the modeling software. However, if necessary, we could run an initial global alignment based on ICP [13] or a shape-based alignment [28] or allow for a rough manual alignment by painting on corresponding regions. We leave this for future work.

Adjacency Cost C_a. The geometric costs alone are not sufficient to produce intuitive visual differences since it does not take into account changes in the elements adjacencies. The exact matching subfigure in Figure 4.3, discussed in the following section, shows a more complex example of the benefit of explicitly including element adjacencies. We assign adjacency costs to pairs of adjacent elements (e_1, e_2) in M and (e'_1, e'_2) in M'. We consider all adjacencies of faces and vertices (i.e. face-to-face, face-to-vertex, and vertex-to-vertex). We include costs for adjacencies that are mismatched between versions and costs for adjacencies that are matched but with strongly different geometries. The adjacency term can be written as

$$C_a(O) = \sum_{(e_1,e_2)\in U} \frac{1}{v(e_1)+v(e_2)} + \sum_{(e'_1,e'_2)\in U'} \frac{1}{v(e'_1)+v(e'_2)} + \\ + \sum_{(e_1,e_2)\in A} \frac{w(e_1,e_2,e'_1,e'_2)}{v(e_1)+v(e_2)} + \sum_{(e'_1,e'_2)\in A'} \frac{w(e_1,e_2,e'_1,e'_2)}{v(e'_1)+v(e'_2)}$$

with

$$w(e_1,e_2,e'_1,e'_2) = \frac{|d(\mathbf{x}_{e_1},\mathbf{x}_{e_2}) - d(\mathbf{x}_{e'_1},\mathbf{x}_{e'_2})|}{d(\mathbf{x}_{e_1},\mathbf{x}_{e_2}) + d(\mathbf{x}_{e'_1},\mathbf{x}_{e'_2})}$$

and where $v(e)$ is the valance of a node e, U are the sets of adjacent element pairs (e_1, e_2) in M that do not have matching adjacent pairs in M', U' is the corresponding set in M', A is the set of adjacent element pairs (e_1, e_2) in M that have matched elements in M', and A' is the corresponding set on M'.

The adjacency cost has two terms. The first one, defined symmetrically over both meshes, penalizes mismatches in adjacencies between the two meshes when two adjacent elements in a mesh end up not adjacent in the other. This can happen either if one of them is unmatched or if they are both matched but to non-adjacent elements. The cost of each mismatch is the inverse of the valence in the graph, i.e. the size of the local neighborhoods. This can be thought of as a normalization that ensures that elements with a large number of adjacencies (such as extraordinary vertices or poles) are not weighted significantly higher than elements with only a few adjacencies (such as vertices at the edges of the model). Moreover, this normalization works well with meshes that contain a mixture of triangles and quads or has highly regular or irregular adjancencies without the need for user-tunable parameters.

The second term, also defined symmetrically over both meshes, penalizes adjacent pairs that have very different locations in the two versions with a cost that is proportional to the relative change in location, normalized by the element valencies. This term ensures match adjacent pairs of elements to a pair of elements that are relatively the same distance apart, which helps when the mesh has been heavily sculpted. The term is divided by the size of the local neighborhoods so high-valence elements are not weighted more heavily than low-valence elements. Note that there is no cost for matched adjacencies when the distance between elements has not changed.

4.5 Algorithm

Equivalent Graph Matching Problem. Minimizing the mesh edit distance to determine the optimal mesh matching can be formulated as a matching problem on a appropriately constructed graph. Given a mesh, we define such a graph by first creating attributed nodes for each mesh element, where the attributes are the element's geometric properties. We then create an undirected edge between two nodes in the graph for each adjacency relation between pairs of elements in the mesh. We can then determine a good mesh matching by minimizing the mesh edit distance over the graph. Unfortunately, this matching problem is related to solving a maximum common subgraph isomorphism problem [57, 14], that is known to be NP-Hard in the general case. And, while many polynomial-time graph-matching approximation algorithms have been proposed [30], we found that they do not work well in our problem domain, because they either ignore adjacency (i.e. edges in the graph), approximate the adjacencies too greatly, or do not scale to thousands of nodes. In MeshGit, we propose to compute an approximate mesh matching using an iterative greedy algorithm that minimizes our cost function. We include source code and executable for our implementation in supplemental material.

4.5.1 Iterative Greedy Algorithm

We initialize the matching O by quickly determining which parts of the mesh have not moved. The algorithm then iteratively executes a greedy step and a backtracking step. The greedy step minimizes the cost $C(O)$ of the matching O by greedily matching (or removing the matching between) elements in M to elements in M'. The backtracking step removes matches that are likely to push the greedy algorithm into local minima of the cost function. We iteratively repeat these two steps a fixed small number of times (4 in our case). Figure 4.2 illustrates how O evolves for subsequent iterations.

Initialization. We initialize the matching O by setting each element in one mesh to match its nearest neighbor in the other mesh if their geometric distance is smaller than an a threshold (0.1 in our case). We leave unmatched all other elements. This initialization speeds up the matching in that it quickly match elements that have not changed geometrically and it is experimentally equivalent to initializing with the empty matching. Note that if incorrect assignments happen, they will be later undone.

Greedy Step. The greedy step updates the matching O by consecutively assigning unmatched elements or removing the assignment of matched ones. We greedily choose the change that reduces the cost $C(O)$ the most, and we remain in the greedy step

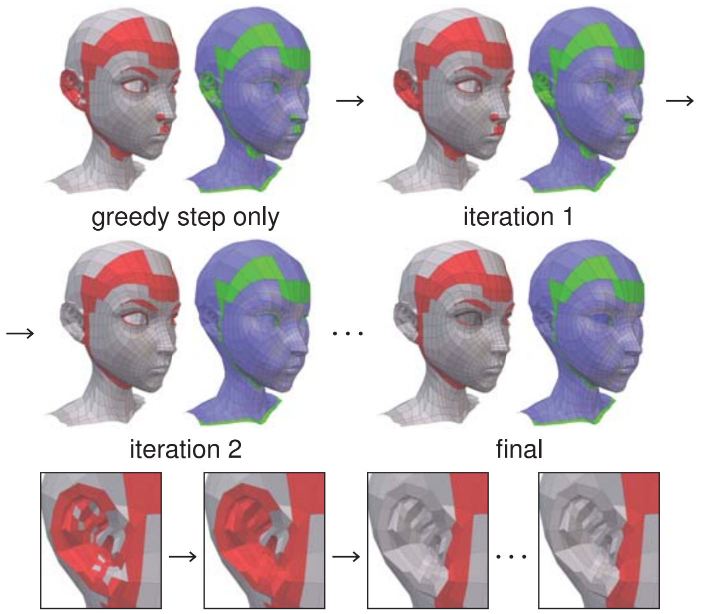


Figure 4.2: *Two-way diffs taken for subsequent steps of our iterative algorithm, where each iteration refines the differences to become more precise. These two versions were independently edited, so neither is the derivative of the other. This is the worst case for diffing. Nonethless MeshGit handles this case well.*

until no change is found that is cheaper to perform than keeping the current matching. Notice that this may leave some elements unmatched. In practice we found that the greedy step proceeds by growing patches. This is due to the adjacency term that favors assigning vertices and faces that are adjacent to already matched ones.

The greedy step may produce unintuitive results since it can get stuck in local minima, it may produce face matchings with vertices in an incorrect order, or require duplicating or merging elements. We handle the local minima with the backtracking step discussed below. A face match is ill-formed when the vertices are also matched but in an incorrect order. For example, suppose that a face f, defined by vertices (a, b, c, d), matches a face f', defined by vertices (a', b', c', d'), where a matches a', b to b', c to d', and d to c'. We eliminate these cases by unmatching the vertices of these faces. While allowing for duplication or merging of elements may be desirable for visualizing certain mesh operations (e.g. a loop cut), we take a simplified approach and seek to only visualize added, deleted, or moved elements. We thus remove such matches by finding and unmatching all adjacent pairs in one mesh that match elements in another mesh that are not adjacent, all matching faces with unmatched vertices, and all matched vertices with no matching faces. We leave visualizing element duplication and merging for future work.

Backtracking Step. While we found that in many cases the greedy step alone works well, we encountered a few instances where the algorithm gets stuck in a local minimum, as shown in Figure 4.2, caused by the order in which the greedy step grows patches. The geometric term favors assigning nearby elements. However, if part of the mesh has been sculpted, the geometric term might favor greedy element assignments that incur small adjacency costs locally, but large overall adjacency costs as more elements are later added to the matching. This is the case when a region of connected faces that have been matched meets the rest of the mesh over mismatched adjacencies. These disconnected regions are usually quite small relative to the size of the whole connected component upon which they reside. These regions are not due to the mesh edit distance we introduced, but to suboptimal initial greedy assignments, favored by the geometric term, in sculpted meshes that may also have edits that affect adjacencies. To eliminate these small regions, we backtrack by removing the assignments of all elements in matching regions whose size is small relative to the component size. The size of a region or component is defined as the number faces in the region or component, respectively. The threshold ratio is initially set to 8%. We run iteratively the greedy and backtracking step four times in total. To help with convergence and avoiding getting stuck in the same local minimum, at each iteration we reduce the geometric cost by a quarter and the backtracking threshold ratio by half.

Time Complexity. The cost of our algorithm is dominated by the iterative search for the minimum cost operations in the greedy step. Since we perform $O(n)$ assignments, each of which considers $O(n)$ possible cases, a naive implementation of the greedy step would run in $O(n^2)$ time. Given the geometric terms for vertices and faces in the cost function, we can prune the search space considerably. In our implementation, we only consider the k nearest neighbors for each unmatched vertex or face and the neighbors within r hops in the graph. We set $k = 10$ and $r = 2$. Because these prunings can severely decrease the search space, if an element e_1 is unmatched but an adjacent element e_2 is matched to e'_2, we also search the k nearest neighbors and r-ball graph neighborhood of e'_2 for potential matches for e_1. Such a locality of searching considerably reduces the computation time without compromising results even when the meshes have been heavily sculpted. This reduces the overall cost to $O(n \log n)$. Furthermore, we compute the change in the cost function with local updates only, since assigning or removing matches only affects the costs in their local neighborhoods.

4.5.2 Editing Operations

Given a matching O from a mesh M to another mesh M', we can define a corresponding set of low-level editing operations that will transform M into M'. Unmatched elements in M are considered deleted, while unmatched elements in M' are added. Matched vertices that have a geometric cost are considered transformed (i.e. translated), while those without geometric costs are considered unmodified (thus not highlighted in diffs nor acted on during merging). Matched faces are considered edited only when they have mismatched adjacencies; in this case, we can consider them as deleted from the ancestor and added back in the derivative. Notice that we do not explicitly account for changes in face geometry since they are implicitly taken into account in edits to vertex

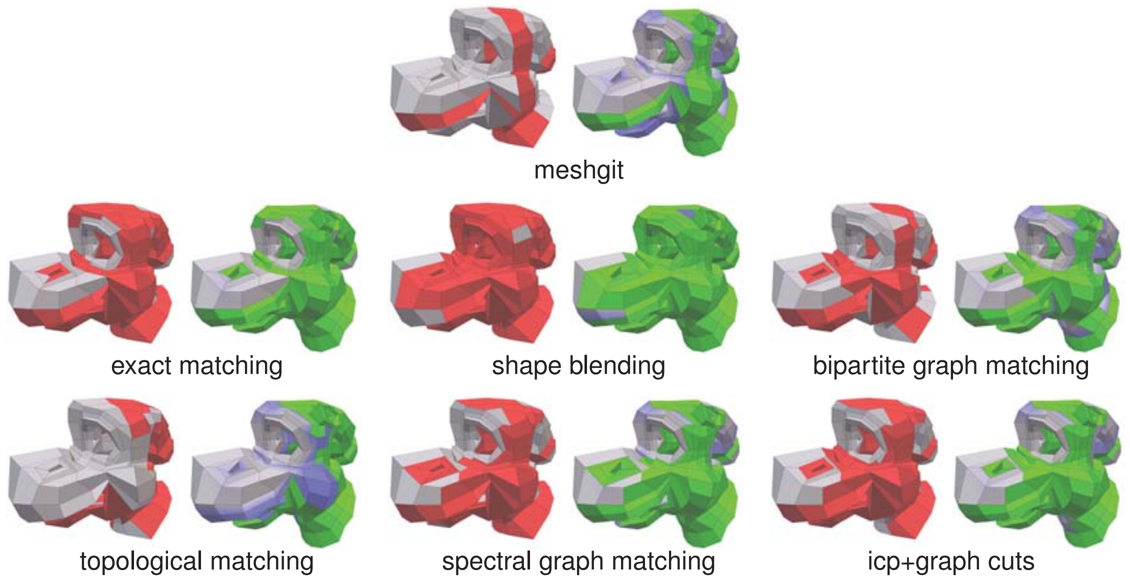


Figure 4.3: *Two-way diffs from different matching algorithms. Compared to MeshGit, the results of the prior methods contain more mismatched adjacencies, because the methods either do not account for adjacencies, do not account for geometry changes, or produce a fuzzy matching.*

geometry.

Although the set of mesh transformations produced by this process are very low-level compared to the mesh editing operations in a typical 3D modeling software (e.g. extrude, edge-split, merge vertices), we found that this provides intuitive visualizations and allows to robustly merge meshes. We leave the determination of high-level editing operations to future work.

4.5.3 Discussion

Comparison. Figure 4.3 shows the results of using different shape matching algorithms to show visual differences. We included our method, an "exact" match based where each element is just match to the closest one (i.e. our initialization step only), bipartite graph matching [64], spectral graph matching [23], shape blending [40],

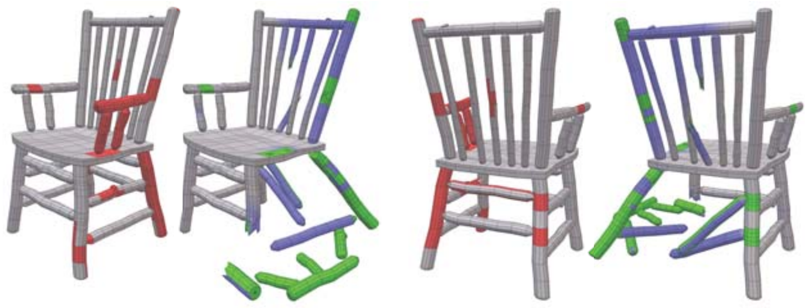


Figure 4.4: *Two-way diff showing the main limitation of our approach. While MeshGit detects most edits correctly, it fails to properly capture edits in the back leg since both geometry and adjacencies change significantly.*

topological matching [29], and iterative closest point with graph cuts [17]. The shape blending and iterative closest point algorithms match vertices only; to generate the visualization, face matches were inferred. The bipartite, spectral, and topological matching algorithms matched faces instead; we infer from them vertex matches to visualize our results. We use the same matching costs for all methods, when applicable. The input meshes are versions 3 and 4 of the modeling series shown in Figure 4.6[1].

Matching based on only the closest element within a given radius marks more changes than are actually performed since adjacency cannot be used to guide the match in sculpted areas. The bipartite graph matching algorithm matched elements, regardless of the implied changes to adjacent elements, producing a large number of mismatched adjacencies. The spectral matching and shape blending algorithms do consider adjacencies, but only implicitly, resulting in many mismatched adjacencies where the graph spectrum changes due to additional features or blending the matches becomes fuzzy with additional edge loops or sculpting. The topological matching algorithm produced topologically consistent matches regardless of the implied changes to geometry of the vertices, leading to matches that are clumped or shifted toward the

[1]Version 4 in Figure 4.3 was modified to contain only the largest connected component, since the shape blending algorithm requires a single connected mesh.

initial seed matching. The iterative closest point with graph cuts algorithm worked to align chunks of the mesh, but heavy sculpting causes the algorithm to require too many cuts. We found these trends to be present in a variety of other examples.

It is our opinion that MeshGit is able to better visualize complex edits that include both geometry and adjacency changes, since it strikes a balance between accounting for both types of changes, compared to other methods that favor one over the other. This in turn allows us to produce intuitive visualizations as seen throughout the paper. In our opinion, this is due to the fact that the shape matching algorithms we compared with were not designed specifically for our problem domain, but for other applications for which they remain remarkably effective. Since there are tradeoffs in determining good matches in the case of heavily edited meshes, each algorithm makes a tradeoff specific to their problem domain, and only MeshGit was specifically designed to address version control issues of polygonal meshes.

Limitations. The main limitation of MeshGit is that the inclusion of the geometric term has limitation when matching of components that were very close in one mesh, but have been heavily transformed in the other, if sharp adjacency changes occur also. Meshes that are heavily sculpted are still handled well since in most cases the adjacency changes are limited. An example of this limitation is shown in Figure 4.4, where some of the components of the original chair are split into separate components that are translated and rotated significantly (e.g. the front left leg and the left arm rest). While MeshGit matches well parts of the chairs, the most complex transformations are not detected. Performing hierarchical matching by matching connected components first followed by the elements of each components can help, but it would make edits that partition or bridge components difficult to detect. For an example of such an edit, the center back support is broken into two parts, and our algorithm can currently detect it. These issues might be alleviated by using a geodesic or diffusion distance in the geometric term, or additional terms inspired by iterative closest point [13] could be added. At the same time though, we think that changes such as these might make more common edits undetected, so we leave the exploration of these modifications to future work.

Furthermore, we believe that while MeshGit is very effective for mesh edited in typical subdivision modeling workflows, it is not as effective on fundamentally different editing workflows, namely the ones that make heavy use of remeshing, where artists are only concerned about mesh geometry and not adjacency. Figure 4.5 shows one such example. In these cases, the differences shown by MeshGit may be correct, but, in our opinion, are less informative for artists, since MeshGit is concerned about changes in both geometry and adjacency, while artists in these workflows are only concerned about overall shape. We believe that these different workflows are better served by algorithms specifically designed for them and leave this to future work.

4.6 Diffing and Merging

Mesh Diff. We visualize the mesh differences similarly to text diffs. In order to provide as much context as possible, we display all versions of the mesh side-by-side with vertices and faces colored to indicated the type or magnitude of the differences. A *two-way diff* illustrates the differences between two versions of a mesh, the original M and the derivate M', as in Figure 4.1.a. We display adjacency changes by coloring in red the deleted faces in M (unmatched or with mismatched adjacencies in M) and in green the added faces in M' (unmatched or with mismatched adjacencies in M'). We display geometric changes by coloring vertices in blue with a saturation proportional to magnitude of the movement. In our visualizations, we simplify the presentation by not

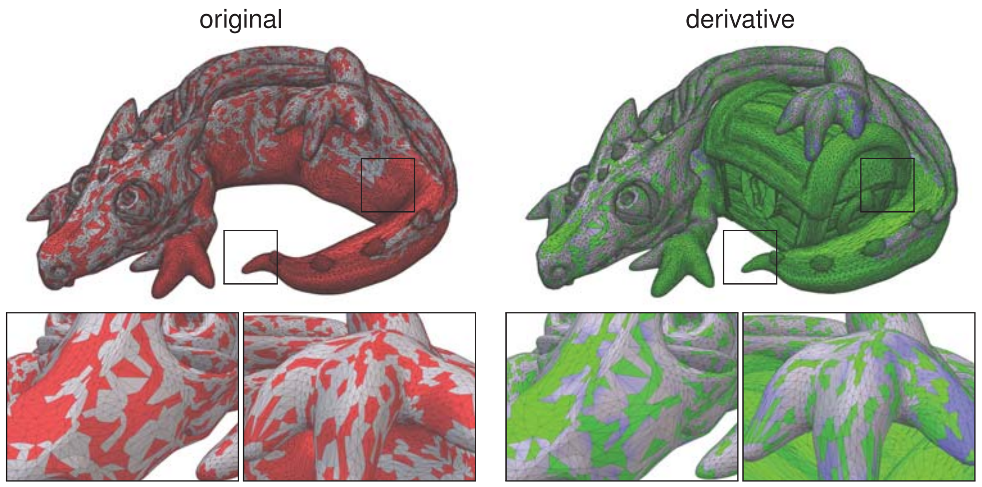


Figure 4.5: *Two-way diff of meshes with similar shape but different adjacencies due to remeshing. While MeshGit computes the diff correctly, the resulting visualization might not be as informative since in this workflow artists focus only on geometry changes.*

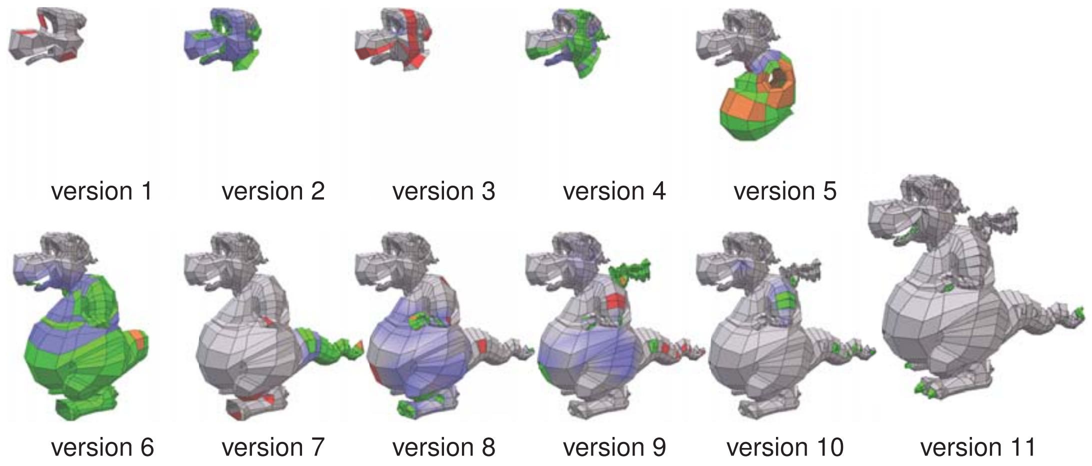


Figure 4.6: *MeshGit can be used to visualize construction sequences, here shown on twelve snapshots. Faces are green if added to the current snapshot or changed from the previous, red if deleted in the next or changed, and orange if added and then deleted or changed both times. Version 11 is enlarged to show better the fine features added, namely the teeth, claws on hand and feet, and the horn at tip of tail.*

drawing the vertices directly but linearly interpolating their colors across the adjacent faces, unless the face has been colored red or green. Unmodified faces and vertices are colored gray. When a mesh M has two derived versions, M^a and M^b, a *three-way diff* illustrates the changes between both derivatives and the original, as shown in Figure 4.1.b. We use a color scheme similar to the above, but the brightness of the color indicates from which derivative the operation comes. When a face has been modified in both derivatives it is indicated in yellow (Figure 4.8).

An artist can also use MeshGit to visualize the *progression* of work on a mesh, as shown in Figure 4.6. Each mesh snapshot is visualized similarly to a three-way diff. For each snapshot, a face is colored green if it was added, red if it is deleted, and orange if the face was added and then deleted. An alternative approach to visualizing mesh construction sequences is demonstrated in *MeshFlow* (chapter 2), that while providing a richer display, also requires full instrumentation of the modeling software.

Mesh Merge. Given a mesh M and two derivative meshes M^a and M^b, one may wish to incorporate the changes made in both derivatives into a single resulting mesh. For example, in Figure 4.1.b, one derivative has finger nails added to the hand, while the other has refined and sculpted the palm. Presently, the only way to merge mesh edits such as this is for an artist to determine the changes done and then manually perform the merge of modifications by hand. MeshGit supports a merging workflow similar to text editing. We first compute two sets of mesh transformations in order to transform M into M^a and into M^b. If the two sets of transformations do not modify the same elements of the original mesh, MeshGit merges them automatically by simply performing both sets of transformations on M. However, if the sets *overlap* on M, then they are in conflict. In this case, it is unclear how to merge the changes automatically while respecting artists intentions. For this reason, we follow text editing workflows, and ask the user to either choose which set of operations to apply or to merge the conflict by hand. We reduce the number of conflicts, thus the granularity of users' decisions, by partitioning the mesh transformations into groups that can be safely applied individually. This is akin to grouping text characters into lines in text merging.

An example of our automatic merging is shown in Figure 4.1.b, where the changes do not overlap in the original mesh. In this case, MeshGit merges the changes automatically. Another example is shown in Figure 4.7. In one version the body is sculpted by moving vertices, while in the other the skirt is removed and the boots are replaced with sandals, thus also changing the face adjacencies. These two sets of differences do not affect the same elements on the original since sculpting affects only the geometric properties of the vertices. MeshGit can safely merge these edits. The top subfigure of Figure 4.7 show the resulting merged mesh with colors indicating the applied transformations. On the right we show recursively applying Catmull-Clark subdivision rules twice to demonstrate that adjacencies are well maintained.

To handle conflicts gracefully, we make the observation that edits that change adjacencies will partition the mesh into regions, such that each region contains faces that are all added, deleted, have some geometric changes, or are unchanged. If we apply all edits of one region, we obtain a resulting merge that is valid and respects the artists changes to adjacencies. Therefore, we partition the edits by finding connected regions of matched elements (similar to the backtracking step) that have adjacency changes on the boundaries, and detect conflicts between the revisions at the granularity of these regions. This is akin to grouping text changes into line, rather than applying them as individual characters.

Figure 4.8 shows an example with a conflicting edit on a spaceship model. In one version, features are added to the spaceship's body and the base of the body has been enlarged. In the other, the cockpit exterior is detailed and wings are added to the

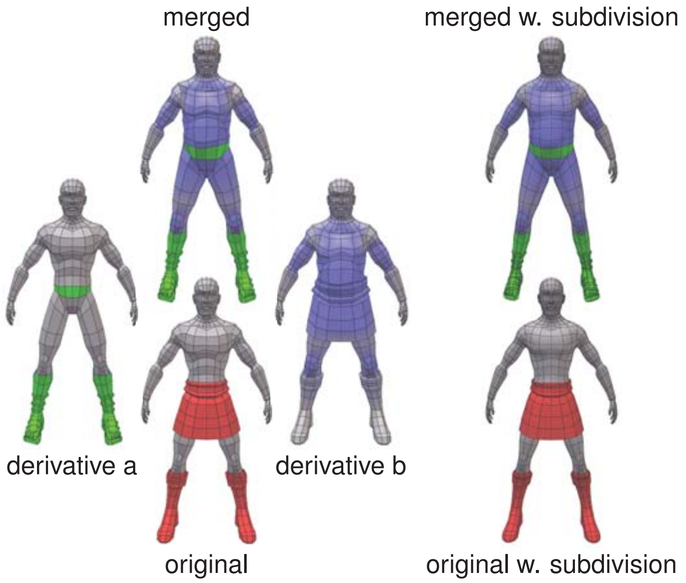


Figure 4.7: *Automatic merge of non-conflicting edits that affect the adjacencies (derivative a) and geometry (derivative b). We show both the original and merge after applying Catmull-Clark subdivision to show that MeshGit maintains consistent face adjacencies.*

base and top of the body. In this case, the extended base in the first version and the added lower wings in the second version are conflicting edits. MeshGit successfully detected the conflicts to the body and merged all other changes automatically (top center). To resolve the conflicts, the user can pick which version of edits to apply and use MeshGit to properly apply the edits, as shown in the figure, or simply resolve the conflict manually. The top three subfigures show three possible ways to resolve the conflicted merge.

4.7 Results

We tested MeshGit on a variety of meshes whose statistics are collected in Table 4.1 by running our algorithm on a quad-core 2.93GHz Intel Core i7 with 16GB RAM. All meshes and source code are available as supplemental material.

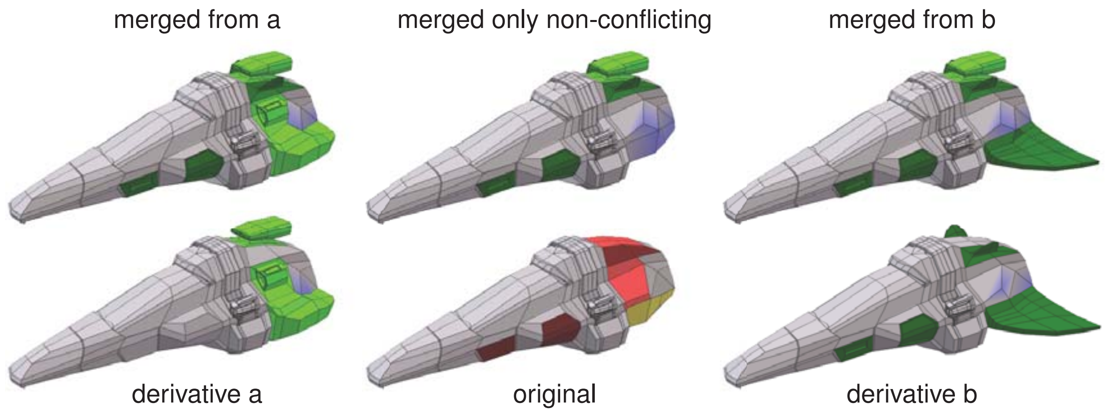


Figure 4.8: *MeshGit detects conflicting mesh differences, visualized in yellow, between the derivatives, and partitions the changes into groups that can be applied individually. In this case, the expanded base of* derivative a *and added wings of* derivative b *are conflicting. All non-conflicting changes are applied automatically, while the user can choose from which version to include the conflicted ones. The top row shows three possible ways of resolving the conflict.*

Model Selection. We chose meshes from different artists that likely have different styles of modeling. The *creature* and *durano* meshes are from two series of saved snapshots taken through the mesh construction history. The *sintel*, *keys*, and *dragon* models are mesh variations where there is no clear original and derivative. The *chair*, *shuttle*, and *woman* pairs contains an ancestor and a derivative mesh. For the *hand*, *shaolin*, and *spaceship*, we model two derivative meshes from the original one to demonstrate merging. See supplemental materials for full reference for meshes. These models span a variety of shape types, including characters to man-made objects, and are made of a mix of triangles and quads. MeshGit worked well regardless of the mesh author and whether their adjacencies were highly regular or irregular. Furthermore, while we expect that MeshGit will be mostly useful when a mesh is derived from an ancestor, we have shown that it works well also when two meshes do not have a clear ancestor. This is significant benefit over instrumentation-based systems that would not be able to compare these cases.

Model	**Reference**	**Fig.**	**Number of Faces**			**Time**
			original	ver. 1	ver. 2	
chairs	[Lumpycow]	4.4	3290	3951	—	4.7s
creature	[Goralczyk]	4.1	11475	17433	—	14.5s
dragon	[Böhler]	4.5	—	88028	96616	307.9s
durano 1	[Vazquez]	4.6	276	520	520	0.5s
durano 4		4.6	786	906	1716	0.4s
durano 7		4.6	1930	2186	2772	1.5s
durano 10		4.6	3078	3722	—	1.2s
hand	[Williamson]	4.1	199	209	209	0.1s
keys	[Thomas]	4.9	—	1652	1854	6.7s
shaolin	[Silva]	4.7	1850	1850	2158	2.4s
sintel	[Blender]	4.2	—	1810	1712	2.7s
spaceship	[Grassard]	4.8	1827	2173	2031	0.9s
shuttle	[Kuhn]	4.9	166974	193970	—	585.3s
woman orig.	[Williamson]	4.9	13984	—	—	—
woman deriv	[Nyman]	4.9	—	8616	—	33.7s

Table 4.1: *Statistics for the meshes used in our tests and the timings to computate of the mesh edit distance between the versions. Full reference for meshes available in supplemental material.*

Timing. As summarized in Table 4.1, the number of faces of the meshes in this paper vary widely from hundreds to over hundreds of thousand. Meshes typically used in subdivision modeling have tens of thousand of faces. In these cases, MeshGit takes at most tens of seconds to compute the mesh edit distance, showing that it can be trivially integrated in a design workflow. We also include significantly larger meshes used in high-polygon modeling. MeshGit scales very well also in these cases, taking only hundreds of seconds. Note that many of the other algorithms we compared with were not only less precise, but would simply have not run on these cases. We further expect that these timings to be significantly improved by a more optimized implementation of our code.

Challenging Models. As seen already throughout out the paper, MeshGit worked well in our tests for both diffing and merging. Figure 4.9 shows a few challenging cases. The *keys* dataset is a mix of triangles and quads with adjacencies that are less regular than meshes used for subdivision. MeshGit can handle these irregular cases just as well. The *woman* pair shows such significant amount of sculpting and adjacency changes, that at a cursory look it is not easy to tell that these meshes are related. MeshGit works well also in this extreme case and clearly highlights the changes that turned a mesh into the other. Finally, The *shuttle* model is a large modeled modeled with thousands of individual components, whose provenance was not known, that are heavily modified and sometimes welded together. Even if this model was built as components, the system presented by Dobos and Steed [26] could not handle it since the provenance in not known and the components themselves are sometimes merged. MeshGit simply treats each mesh as a whole and finds meaningful differences without the need to properly manage components manually.

4.8 Conclusion and Future Work

This chapter presented MeshGit, an algorithm for diffing and merging polygonal meshes. Inspired by version control for text editing, we introduce the mesh edit distance as a measure of the dissimilarity between meshes and an iterative greedy algorithm to approximate it. We transform the matching computed from the mesh edit distance into a set of mesh editing operations that will transform the first mesh into the second. These operations can then be used directly to visualize the difference between meshes and to merge edits. In the future, we would like to extend our implementation to support diffing and merging of other geometric attributes (e.g. UV, bone weights, etc.). This should be an easy extension to MeshGit that would requires us to change our mesh elements

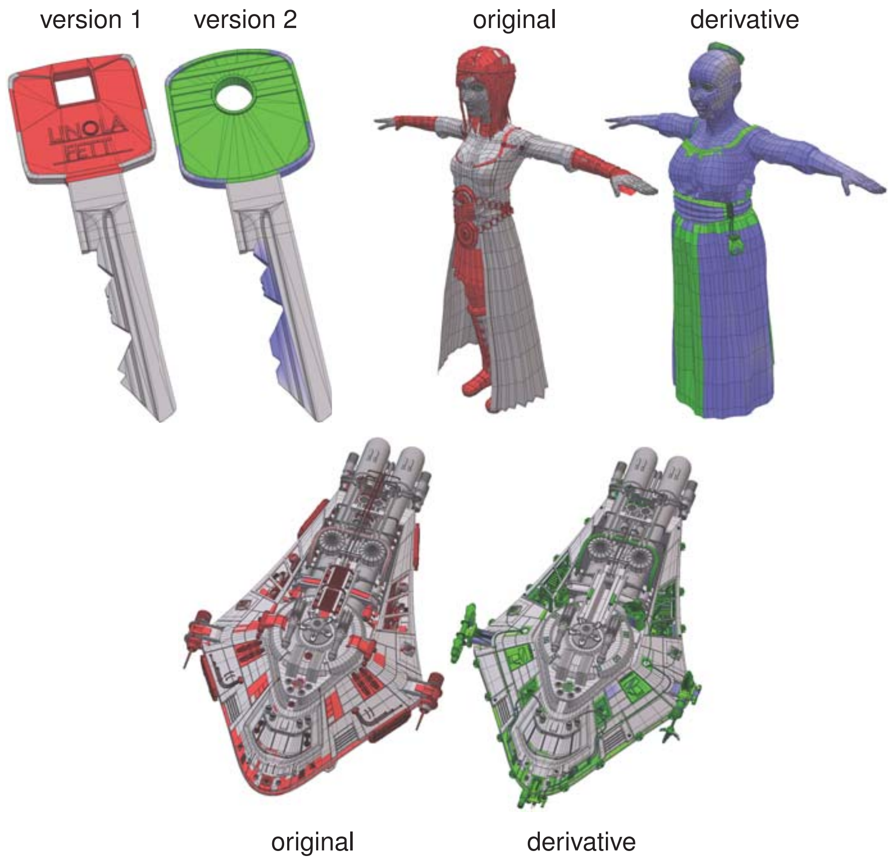


Figure 4.9: *MeshGit handles well cases with irregular adjacencies (top-left), with significant geometric and adjacency changes (top-right), and with high vertex and face counts (bottom; 167k and 194k polygons from 2254 and 3352 original components respectively). All six of these meshes are composed of both triangles and quads.*

to allow for arbitrary data to be attached with diffing and merging following similar algorithms. We also plan to explore other uses of our mesh edit distance in editing workflows. For example, we believe it would allow “spatial undos”, where all operations related to a part of the mesh could be removed regardless of the order they were executed in. Finally, we could use MeshGitto automatically generate mesh variations from only a few models by automatically applying different edits combination.

Chapter 5

CrossComp: Comparing Multiple Artists Performing Similar Tasks

This chapter describes how to extend the work of the previous chapters to visualize and compare multiple artists performing similar mesh editing workflows.

5.1 Overview

In the previous chapters, we have focused on summarizing and visualizing the edits of a single workflow and visualizing and merging the edits of two independent workflows. In this chapter, we focus on visualizing the similarities and dissimilarities of many workflows where digital artists perform similar tasks. The tasks have been chosen so each artist starts and ends with a common state. We show how to leverage the previous work to produce a visualization tool that allows for easy scanning through the workflows.

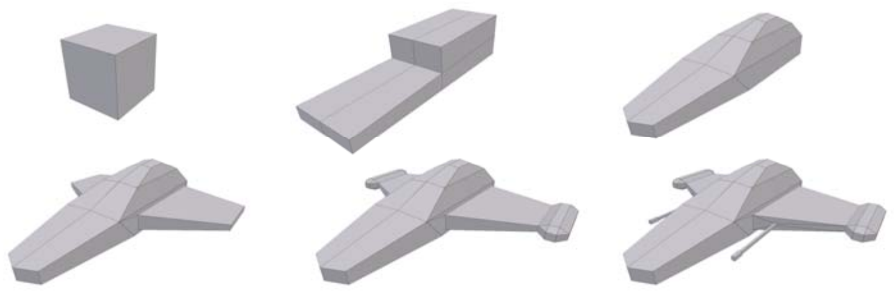


Figure 5.1: *A subset of snapshots from Scout sequence by Author.*

5.2 Introduction

Let us consider the following scenario as a motivating example. Suppose that a digital arts instructor assigns to the students the task of creating a particular 3D model. The assignment could be used to assess the students' ability or technique or to teach the student a new technique. For the former use-case, the instructor might choose to give to the students a target model to recreate. For the latter, the instructor might present the instructions in the form of a video tutorial. The scenario illustrated above is a common practice especially for web-based mentoring, such as with CG Cookie.

When the assigned task involves many components, the instructor may ask the students to periodically save a snapshot of their model as they work and then submit their workflow as a work-in-progress sequence. When the task is a single piece, the students may only report the final state of their model.

In chapters 2 and 3, we demonstrated two systems that summarize and visualize a single artist working on a single task. Clearly, the instructor could use one of these tools to review the workflow of each student. These tools and techniques, however, do not help with determining how closely the student followed the tutorial, with identifying effective or efficient workflow patterns, or with finding poor techniques or common modeling problems.

In this chapter, we present CrossComp, a system designed to help compare multiple artists performing similar mesh editing tasks. We focus on task-based polygonal modeling workflows, where the start and ending conditions are highly defined but the workflows from start to finish may differ. We demonstrate CrossComp by analyzing four subjects performing four tasks, where three tasks use a video tutorial and the fourth uses a target 3D model. We remark on some observations on the workflows that are clear in CrossComp but might have been missed with manual inspection or with inspecting only snapshots. Finally, we conclude with reporting on open-ended feedback from a professional digital artist and instructor and with discussing limitations and future research directions for this work.

5.3 Related Work

The works by Kong et al. [41] and Pavel et al. [61] are closely related to the work presented in this chapter. The goal of their work is to help users identify the trade-offs between many possible workflows that perform the same image-editing task, such as "Find Edges" or "Sketch Effect". They present and evaluate different workflow visualizations for displaying and comparing image-editing workflows. One visualization, called *union graph*, compares two sequences of commands by showing each sequence as a directed graph with a node for each operation and directed edges to indicate temporal order. The similarity and dissimilarity is indicated by overlapping nodes of the two graphs if the corresponding operations are sufficiently similar in terms of operation name or type and parameter settings. Another, called *alignment view*, compares two or more sequences of commands by arranging the workflows according to similarity in operation usage. The operations of each workflow is drawn as a list, and edges are drawn between neighboring operations that are similar.

While their data included short, highly-polished tasks scraped from photo-editing tutorials, our work focuses on much longer workflows that can contain errors and undone work. Furthermore, although they provided step-by-step visualizations of the workflow allowing for manual inspection and comparison, their automated methods rely solely on the operation type and parameter settings. Typically mesh editing software has far fewer number of operations that can be performed with many operations able to perform several different types of manipulations. In other words, when compared to image editing workflows, the differences in mesh editing workflows depend more on the effect of the operation or the combination of operations than the actual operation name, parameter setting, or order of operations.

Lafrenier et al. [44] describe a system, FollowUs, where a user can view a tutorial submitted by the original author or by other users performing their version of the tutorial. Matejka et al. [52] describe a recommender system, CommunityCommands, that collects usage data from a user community and then displays to each user a set of commands the user may not be familiar with. These two systems enhance a user's understanding of the tutorial or software system by presenting how other users of the community perform the task or use the software. The focus of our work is to provide the user a tool to compare the workflows of the community, not just to review.

5.4 Data Collection

Our experiments consisted of four relatively short tasks, involving roughly 20 to 60 minutes of modeling, of moderately increasing difficulty. The first three tasks we presented to the subject in video tutorial format, and the final task was given as a target model. We asked the subjects to follow as closely as possible the steps in the three video tutorials and to recreate as precisely as they could the target mesh of the fourth task. For

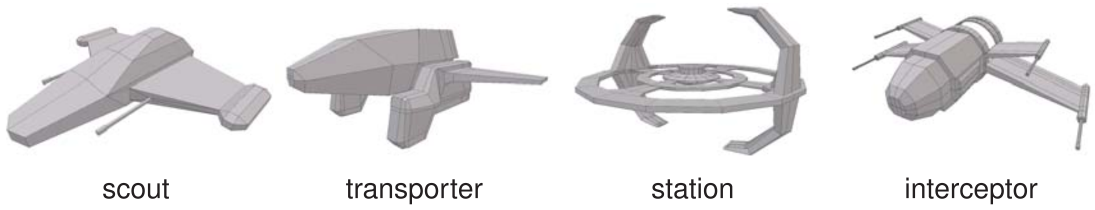


Figure 5.2: *Final meshes for each task.*

the final task, the subject could use any modeling technique to replicate the model.

Although all of the subjects reported having some modeling experience, some did not have any experience using the chosen modeling software prior to starting the experiment. Therefore, we designed the video tutorials to provide software usage instruction in addition to high-level explanations of the mesh construction via an overlaid audio track. The video of each tutorial is a screen-capture time-lapse of the construction played back in an interactive video player at real time with a few pauses to point out features. The mesh of the final task is viewed within an interactive 3D viewer to allow the subject to inspect and interact with the mesh.

We chose for all four tasks the theme of spaceships. Although these goal-based tasks would limit the exploration and variability of the workflows, we felt that open-ended tasks or goals that were open to interpretation would inject a subjectivity and aesthetic component into the workflow that would make objective analysis significantly more difficult. Figure 5.2 shows the final mesh for each of the four tasks.

We used an instrumented version of Blender to record the workflows, both for the author and for the subjects. The starting condition for all tasks contains a single unit cube. Every action that modified the state of the modeling software was recorded, including the undoing of work. The entire recording system for the subject was a self-contained executable with a simple interface, which simplified the process for the subject, and allowed the subject to work at their own pace.

Model	Type	Author	Subj. 1	Subj. 2	Subj. 3	Subj. 4
scout	video	100	125	298	144	217
transporter	video	171	197	238	164	311
station	video	244	—	160	306	377
interceptor	mesh	195	507	283	230	465

Table 5.1: *Statistics for workflow comparison data. The numerical values indicate number of mesh changing edits (no selections, view changes, etc.). The author created the video tutorials (scout, transporter, station) and mesh target goal (interceptor) that the other subjects followed and tried to reproduce. Note: Subject 1 did not finish the station task.*

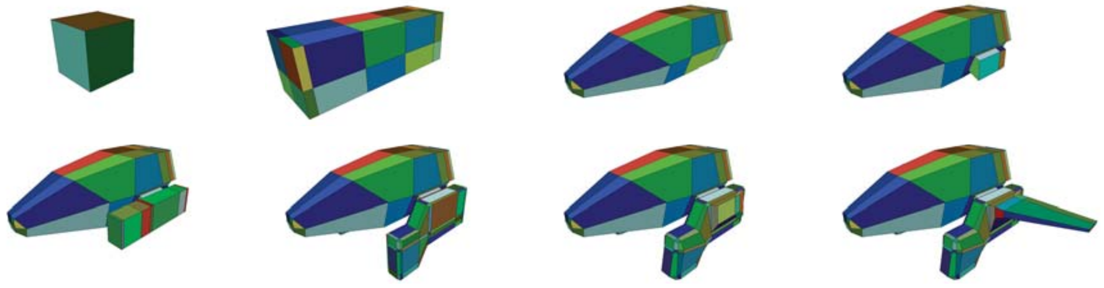


Figure 5.3: *A subset of snapshots from Transporter sequence by Author. Corresponding faces are colored similarly.*

Four subjects participated in the study, but one subject did not submit one of the tasks. See Table 5.1 for statistics on the recorded workflows.

5.5 Correspondence and Distance

CrossComp takes as input the recorded snapshots of the corresponding workflows. In order to compare, CrossComp must build an intra-correspondence of elements along each workflow and an inter-correspondence between the workflows. The intra-correspondences is constructed similarly to MeshFlow, where each face is uniquely labeled (locally) upon creation and tracked throughout the workflow.

While we cannot make any assumptions about the state of the mesh in the middle of the workflow, because the mesh can be arbitrarily changed, we can assume that the beginning and ending states of two workflows are similar to known states. As the beginning state for each task is a unit cube and therefore not very informative in terms of inter-correspondences, we use the final state of each workflow to build inter-correspondences. We use a slightly modified MeshGit[1] to build inter-correspondences between the ending state of the meshes and to uniquely label (globally) the faces. See Figure 5.4 for results of building inter-correspondences.

Snapshot Edit Distance. One way to compare two meshes to find how similar or dissimilar they are is to compute an edit distance between the pair. The edit distance between two meshes is defined as the minimal amount of change required to turn one mesh into the other. If the edit distance is small, then the two meshes are quite similar; if the distance is large, then the two meshes are quite dissimilar.

In Chapter 4 we defined a mesh edit distance which we used to build a correspondence between meshes. CrossComp uses a modified version of the mesh edit distance[2] along with the already established intra-correspondences and inter-correspondences to compute an edit distance between any two pairs of snapshots.

5.6 Visualization

The basic user interface for CrossComp is shown in Figure 5.5. The left column shows a 3D embedding of the snapshots after performing a nonlinear dimensionality reduction of the pairwise edit distances, the center column visualizes a heat map of the pairwise

[1]The MeshGit modifications include: the the dot product of the elements' normals in geometric cost are made absolute, and the greedy step is performed one additional time at end without removing twisted faces or faces with mismatched adjacencies. The first modification accounts for flipped normals, and the second modification allows MeshGit to match as many faces as possible by ignoring mismatched adjacencies.

[2]The snapshot edit distance considers only the face elements of MeshGit's mesh edit distance.

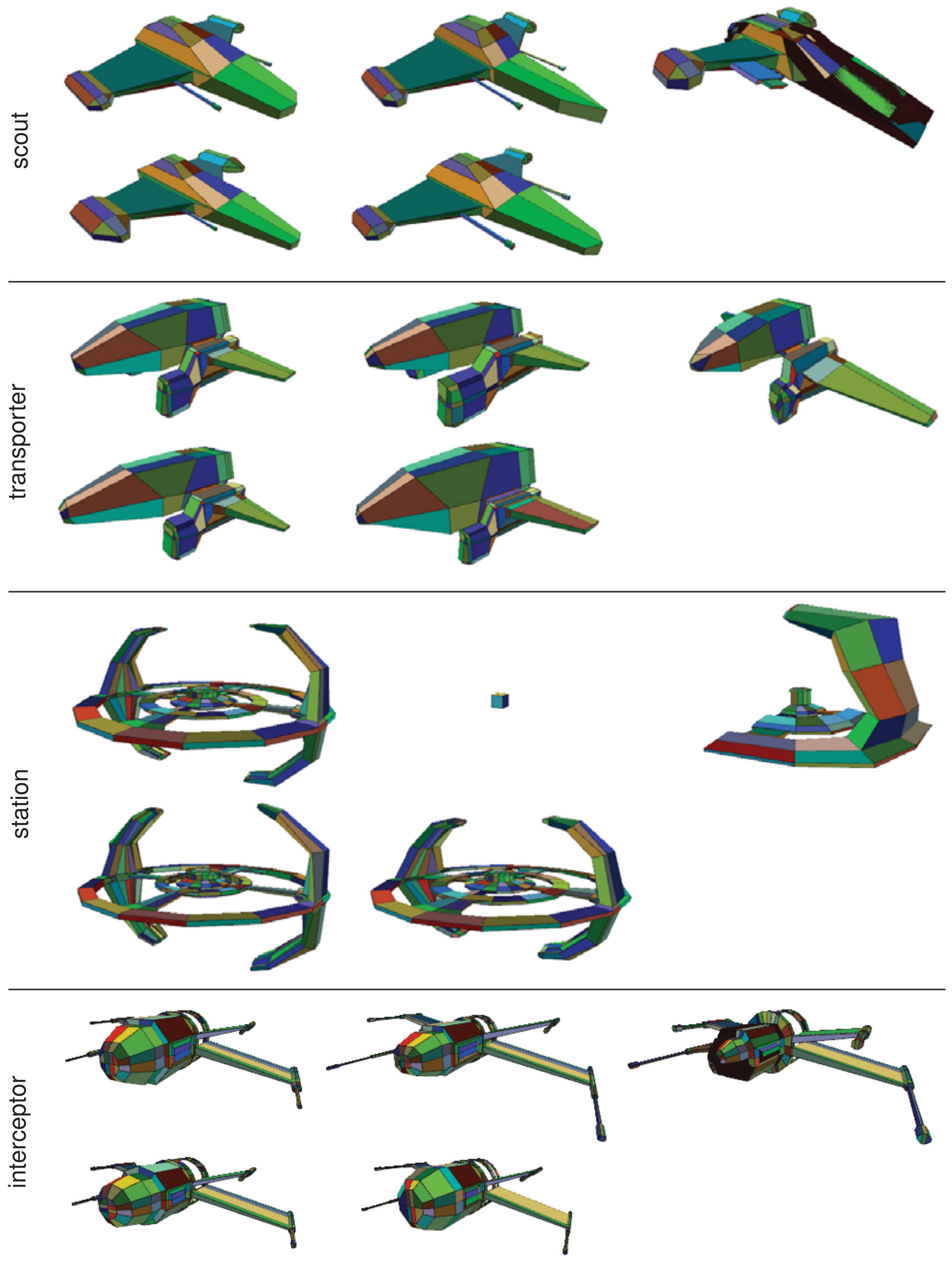


Figure 5.4: *Final meshes for each of the tasks with inter-correspondences illustrated by matching face colors. The top-left subfigure for each workflow was constructed by Author, and all other subfigures are for the modeling subjects. The faces of top-left subfigure are randomly colored, and the faces for other workflows are colored to indicate inter-correspondences. If the face does not have an inter-correspondence, it is colored dark red.*

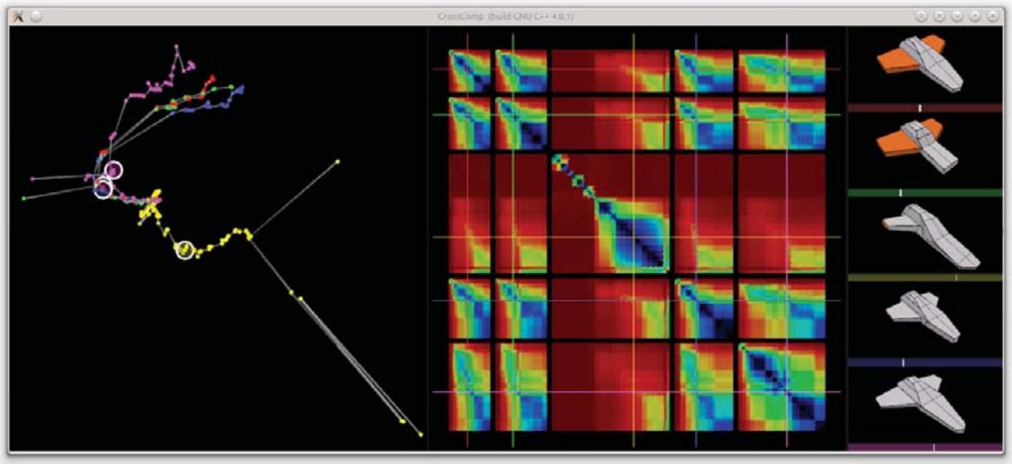


Figure 5.5: *Basic user interface. The left column visualizes the snapshots of workflows in low-dimensional space. The center column shows a pairwise edit distance heat map. The right column contains interactive views of each workflow.*

edit distances, and the right column consists of interactive views of the snapshots for each workflow. While each column visualizes different features of the workflows, they are synced over the time dimension for each workflow. This syncing means, for example, that adjusting the current time of a workflow in one column will automatically update the corresponding visualizations in the other columns. The first column indicates currently viewed time with a white circle; the second with horizontal and vertical lines; the third with white ticks on the colored bars below the model. Each workflow has an associated color (red, green, yellow, blue, purple, resp.). Changes to the mesh are indicated in the third column by coloring the modified faces orange.

In all of the figures, the original tutorial author workflow is the first workflow (red), and the subjects' workflows are compared to the author.

Edit Distance Coordinates. The left side of Figure 5.6 shows a 3D embedding of the Scout workflows according to their pairwise snapshot edit distance. Each snapshot of the workflow is indicated by a dot, colored corresponding to the workflow. The edges

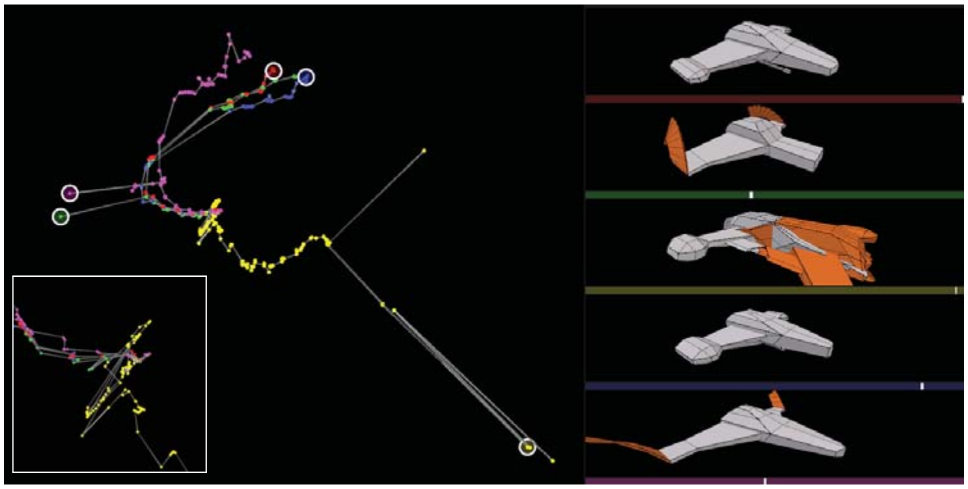


Figure 5.6: *Outliers in Scout task. Two of the workflows (2,5) used the wrong operation or and one the wrong parameter setting (3), causing a very large change that differed greatly from the other workflows. Inset zooms into the initial state of the workflows.*

between dots indicate temporal order of edits. We performed a nonlinear dimensionality reduction on the pairwise edit distances by using Isomap [74] with a k-nn search to find the local neighborhood. We used a value of $k = 10$, but forced at least one mesh from each workflow to be included (the mesh with smallest edit distance) so the embedding would take all workflows into account.

The dots corresponding to two similar snapshots will appear close in this space, while the dots of two quite different snapshots will be far apart. Referring back to Figure 5.6, note the inset figure which zooms into the large cluster of dots near the center of the column. These dots correspond to the early snapshots of the workflows, where the meshes were very similar in shape (the initial cube mesh). From these dots, all the workflows except the third (yellow) follow very closely to each other with just a few outliers. The outliers, selected in the figure, were caused by the artists performing an incorrect operation (here, the *spin* operation instead of *subdivide*). The artist quickly corrected the error by undoing the work and then continued following closely the tutorial. The third workflow, however, diverged from the other workflows

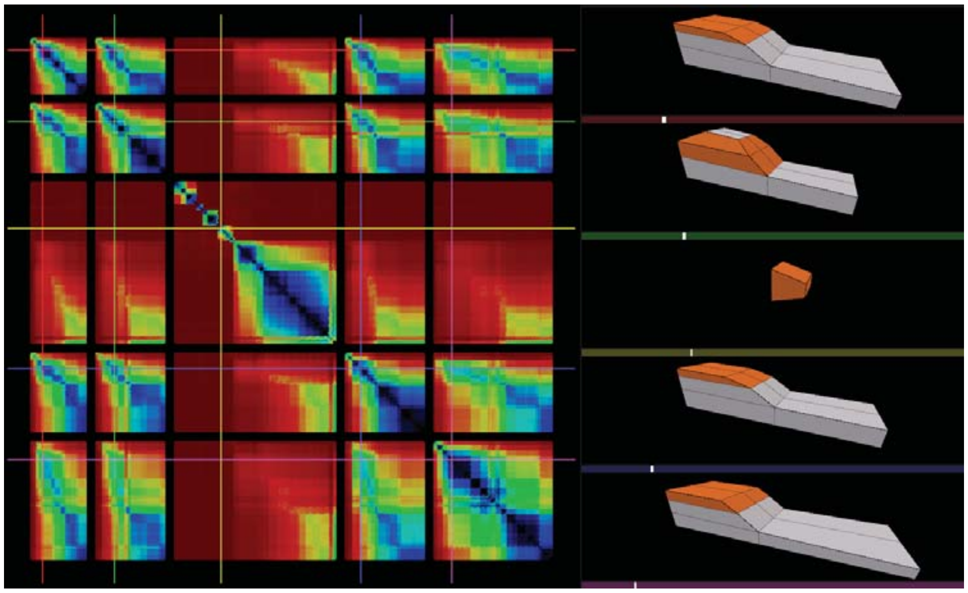


Figure 5.7: *Heat map of Scout task.*

after performing a large number of incorrect operations, seen as the numerous back and forth edges near the center of the inset. Close to the end of the third workflow, we see some additional outliers where the artist attempts to choose the correct parameter settings for the *mirror modifier*. We discuss this more below.

Edit Distance Heat Map. Figure 5.7 shows a heat map visualization of the pairwise edit distances of the Scout task workflows. The topmost row and leftmost column of the heatmap correspond to the first workflow, followed by the second workflow moving down and right, etc. The color in the intersection of a specific row and column indicates the edit distance between the mesh snapshots corresponding to the specific row and column. The color is determined by linearly mapping the regularized distance to a color gradient that runs from black to blue, green, red, and dark red, where a black color indicates no edit distance (exactly the same mesh), and a dark red color indicates a large distance (very different meshes). Extra space is added between rows and columns to distinguish the workflows. The horizontal and vertical lines running

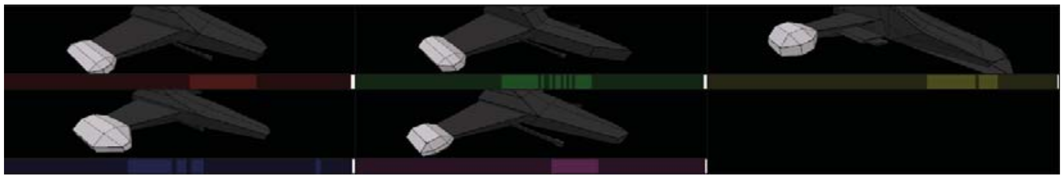


Figure 5.8: *Filtering to spatial selection.*

across the heat map indicate the currently viewed time for the corresponding workflow. We regularize the edit distances by dividing by the total number of faces. We found that edit distance regularization helps filter accumulated change and generates more intuitive heat maps.

One observation to note about the figure is the wide band of dark red rows and columns in the early parts (top-left corner) of the third workflow, where the artist made and corrected several mistakes. Finally, after nearly a third of workflow, the artist was able to follow along with the tutorial, although with some errors which is seen with green color (moderate distance) in bottom-right corner of each block of third row or column.

Cross-Workflow Scrubbing. While the user scrubs through the timeline of one workflow, CrossComp can automatically snap the other workflows to the closest snapshot in terms of the edit distance. This cross-workflow scrubbing allows the user to inspect how all of the workflows progressed, even though the artists may have worked at a different pace. We define the closest snapshot in a specific workflow to a given snapshot as the snapshot with the lowest regularized edit distance from the given.

Spatial Filtering. Similarly to MeshFlow and 3DFlow, the user can perform spatial filtering on the workflows to find when the artists modified a region of interest. When the user selects a face in one workflow, the corresponding faces in the other workflows are selected, too. The timeline (colorbar below the model) is darkened to indicate the edits that do not modify the selected faces. See Figure 5.8 for an example.

5.7 Results

Figure 5.9 displays the results of the Transporter, Station, and Interceptor workflows. Below we will discuss briefly some observations for these workflows.

Transporter. Generally, all four subjects followed the Transporter tutorial relatively closely. The fifth workflow contained a few corrected errors (visualized as the purple outlier runs in the first column.) The first and fourth workflows were the closest pair of workflows. While all the final meshes were similar in shape, the differences of proportions and fine details of the engines caused a divergence of the workflows in the 3D embedded view.

Station. In the Station task, one of the subjects did not submit the completed task, so the second workflow remains as a cube. Also, the third workflow only loosely followed the tutorial and involved fewer edits than the video tutorial, and the subject did not have the mesh positioned correctly for the mirror modifier to duplicate the other three quadrants properly, resulting in an outlier in the first column. The first, fourth, and fifth workflows followed each other closely.

Interceptor. Where the previous tasks were presented as a video tutorial, the interceptor task was presented to the subjects as a final target mesh. The subjects were free to construct the mesh using any techniques and in any order. One important observation to note is that while the artists can construct the mesh in any order, the majority of divergence was due to differences in adjacencies. For example, the first and fourth

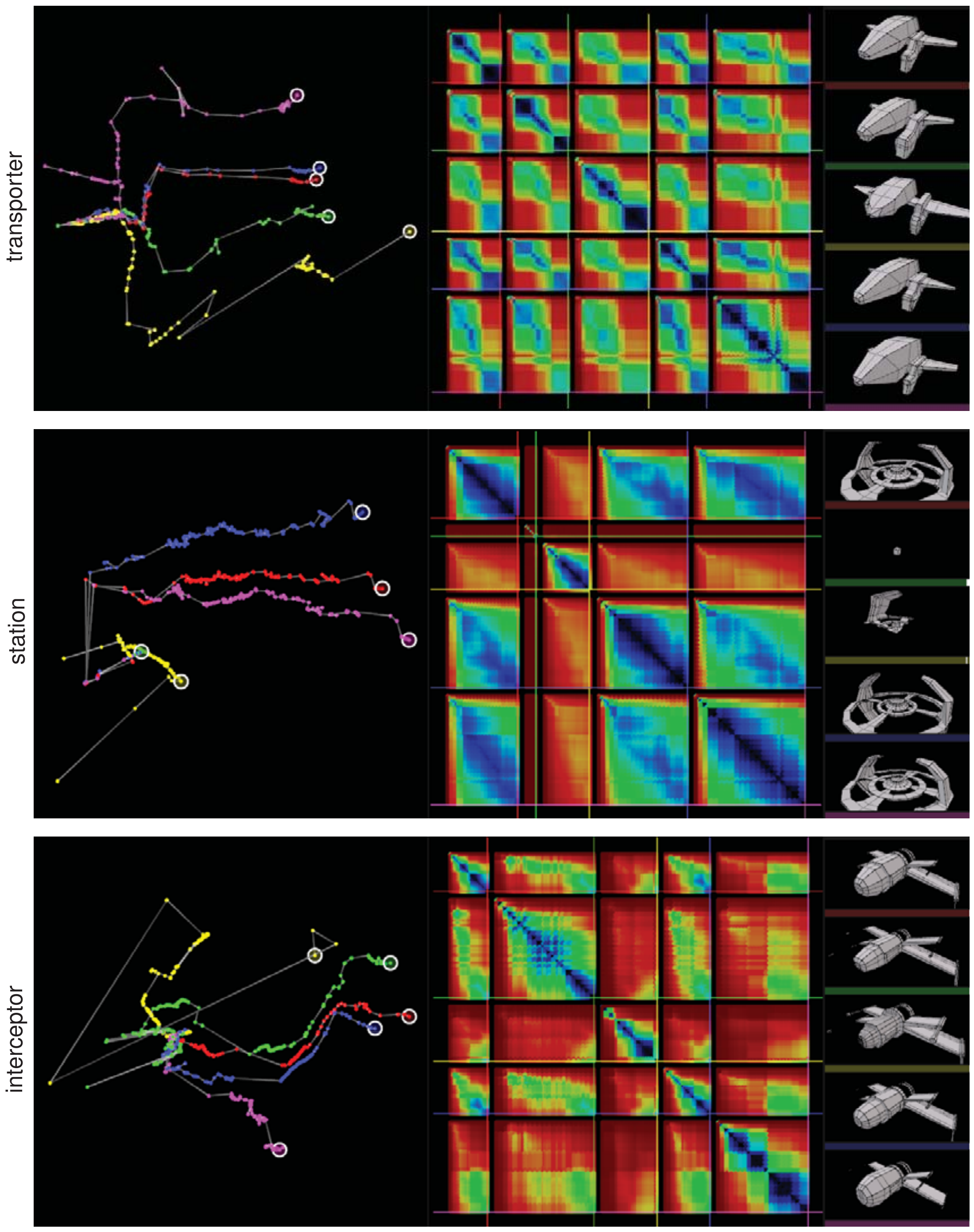


Figure 5.9: *Results of Transporter, Station, and Interceptor.*

workflows are relatively close in the first column, because their meshes are topologically quite similar. However, the second, third, and fifth workflows contained many changes in adjacency (missing features, extra faces, incorrectly connected faces, etc.) and therefore appear to diverge from the first and fourth workflows. The extremely large distances seen in the third workflow are due to setting incorrectly the mirror modifier parameters.

5.7.1 Feedback

We presented our findings to Jonathan Williamson, a professional digital modeling artist and instructor for CG Cookie, in order to gather some open-ended feedback. Williamson stated that the embedded view made it clear when the artists made and then corrected a mistake and that the curves hinted at the similarities of the workflows. When shown the Interceptor dataset, he remarked about how the subjects took a similar approach to constructing the spaceship despite not having step-by-step instructions, which was an unexpected observation.

Williamson said that he is quite excited about the results and interested in finding ways to use CrossComp to instruct. One usage scenario he proposed centers on an assignment he has given before, which follows closely the Interceptor workflow, where he asks the students to create a challenging mesh. CG Cookie has created four exercises of this type in the past, and Williamson states that while they receive many more requests to do more, they have not been able to due to the time involved in reviewing the workflows. After looking over the submitted final versions, he would create a video tutorial on constructing the model while pointing out common mistakes and pitfalls seen in the students' results. He believed that CrossComp would help him in finding, analyzing, and pointing out these situations.

5.7.2 Limitations

There are a few limitations to our input data and approach to analyzing. We discuss some of these limitations in this subsection.

Input Data. We designed our experiments to include instructions for using Blender and to be relatively short and simple. This decision was motivated by some of our subjects may have no experience using Blender and possibly only little experience modeling. Furthermore, despite walking the subjects step-by-step through first three tasks, one workflow was submitted incomplete, and two submitted with gross errors. Although these issues limit the scope of our experiments to novices and amateurs, we found that CrossComp was able to produce intuitive results that helped with making key observations about individual workflows and with comparing the workflows with one another. We leave for future work the study of more experienced subjects performing longer and more advanced tasks.

Correspondences. MeshGit builds a one-to-one correspondence between two meshes. A discrete correspondence works well when the two meshes are very similar in terms of face adjacency. However, when only a fuzzy correspondence is necessary or computable, such as when the models use the mesh to provide a relatively loose representation the surface, other surface correspondence methods might be more appropriate. We chose to use MeshGit's correspondence building method and designed our experiments to fit in these limitations, because MeshGit computes a mesh edit distance which we use directly. We leave the exploration of other correspondence building and distance computing methods for future work.

Edit Distances. Computing a full pairwise edit distance can become quite expensive, growing polynomially in the lengths of workflows and number of subjects. It should be noted that the pairwise distances needs to be computed only once and then cached, is a highly parallel operation, is symmetric, and can be only sparsely computed.

5.8 Conclusion

In this chapter, we presented CrossComp, a method for comparing multiple artists performing similar tasks. Motivated by real-world digital modeling exercises, we demonstrated how to use intra- and inter-correspondences within a set of workflows to compute a pairwise snapshot edit distance. CrossComp can visualize these edit distances as a heat map, where similar and dissimilar snapshots are identified using cool and hot colors, respectively. CrossComp can also perform nonlinear dimensionality reduction on the distances to embed the workflows in a 3D space, where curves and distances indicate similar editing patterns or mistakes and errors. Open-ended feedback from a professional artist and instructor indicate that a system like CrossComp could strongly benefit the instruction community.

Chapter 6

Future Work

This chapter covers possible future directions for this research. We start by describing a possible way to visualize and compare mesh editing workflows of multiple artists performing similar tasks. Then we discuss workflows outside mesh editing to which we believe our methods could extend. Finally we discuss another way to combine the research presented here to create an interactive tutorial system.

6.1 Extending Beyond Mesh Editing Workflows

In this thesis, we have focused on only one segment of the 3D production pipeline, mesh creation and editing. We believe that the methods presented here could be extended to other segments of the pipeline, such as texturing, rigging, or lighting. Furthermore, we believe that the methods presented in this thesis can be extended to any dataset where a workflow can be recorded.

High-level visualizations share insight into the structure and editing patterns of particular users. For example, the version control visualization system Gource [15] visualizes the changes committed to a repository. See Figure 6.1 for a visualization

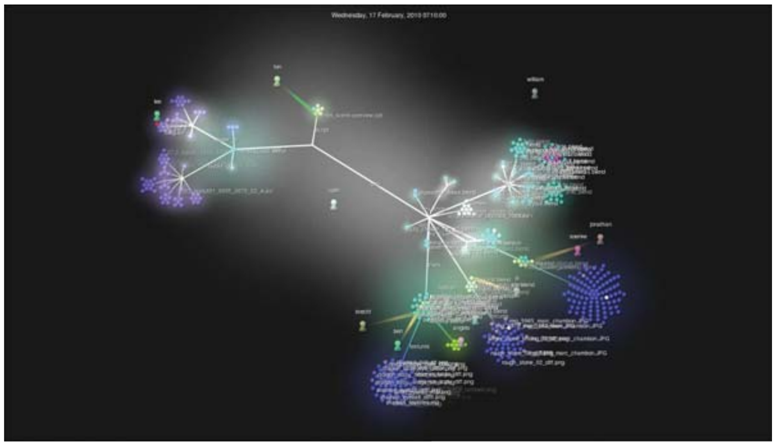


Figure 6.1: *Single frame of Sintel production repository visualization. The software Gource visualizes the production repository for Sintel as a graph, with nodes corresponding to files in the repository and edges indicating the file structure. Actors move about the graph and highlight nodes when files are added, modified, or deleted.*

of the Sintel production repository. While these visualizations are informative from a high-level, they do not visualize the details of what changed. In other words, these visualizations do not distinguish between trivial or major modifications to a file. We believe that a balanced combination of high-level visualizations, from systems such as Gource, and lower-level visualizations, as presented in this thesis, can provide highly informative visualizations.

6.2 Interactive Tutorials

In chapters 2 and 3, we focused on fully automated methods of summarizing mesh editing workflows as an alternative to video- or image-based tutorials. However, we believe that these works could be combined with MeshGit in chapter 4 to create an interactive tutorial with live feedback. In such a system, the student's current work is compared to the tutorial author's work to determine when to automatically advance to the next step. Furthermore, such a system could also evaluate the student's work in progress, providing a score that corresponds to the mesh edit distance which could guide the student's work toward convergence on the target mesh.

6.3 Workflow Comparison

In chapter 5, we demonstrated a method for comparing multiple artists performing similar mesh editing tasks. Recently researchers have begun studying methods for comparing how artists perform short tasks differently and for discovering different short operation sequences artists use. Although comparing workflows is quite subjective in nature, we believe that these objective methods can provide deeper insights into artists workflows, assisting the subjective comparison. Workflow analysis appears to be a fruitful direction.

Chapter 7

Conclusion

In this thesis, we have described methods for studying and managing mesh editing workflows. We focused on automated methods that work well for polygonal meshes constructed by polygonal modeling or digital sculpting techniques.

First we discuss two approaches to summarizing long mesh editing workflows, one relies on editing patterns (MeshFlow, chapter 2) and the other on a change metric (3DFlow, chapter 3). The advantage of the former approach is that the n-gram analysis and levels of detail highlight patterns in the artist's workflow. The advantage of the latter is that it does not rely on tight instrumentation of the editing software and that it works well even when editing patterns are much harder to discern. A small case study with digital arts students indicates that MeshFlow is an improvement over traditional media of showing mesh editing workflows, and digital arts teachers report 3DFlow would greatly assist them in producing tutorials by simplifying their tutorial creation workflow.

Following summarization, we discuss a method for diffing and merging meshes. The key to determining differences between two meshes is building a correspondence between them. We evaluate MeshGit by testing it with a wide range of meshes that have undergone various types of edits.

Building off of the results of MeshFlow, 3DFlow, and MeshGit, we demonstrate a method for comparing multiple artists performing similar mesh editing tasks. Again the key to performing this comparison is in building correspondences, intra-correspondences along a single workflow and inter-correspondences between different workflows. We visualize the pairwise edit distances computed from the correspondences as a heat map and by embedding into 3D space. Open-ended feedback from a professional artist and instructor indicates that these visualizations would greatly improve the workflow evaluation process.

We conclude with a number of potentially fruitful directions in which this research could extend.

Bibliography

[1] Apache. Apache subversion. http://subversion.apache.org, 2013.

[2] Assa, J., Caspi, Y., and Cohen-Or, D. Action synopsis: pose selection and illustration. *ACM Trans. Graphics* (2005), 667–676.

[3] Autodesk. 3ds Max. http://www.autodesk.com/products/autodesk-3ds-max, 2014.

[4] Autodesk. Maya. http://www.autodesk.com/products/autodesk-maya, 2014.

[5] Barnes, C., Goldman, D. B., Shechtman, E., and Finkelstein, A. Video tapestries with continuous temporal zoom. *ACM Trans. Graphics* (2010), 89:1–89:9.

[6] Bergman, L., Castelli, V., Lau, T., and Oblinger, D. DocWizards: a system for authoring follow-me documentation wizards. In *Proc. ACM UIST* (2005), pp. 191–200.

[7] Berlage, T. A selective undo mechanism for graphical user interfaces based on command objects. *ACM Trans. CHI* (1994), 269–294.

[8] Blender Foundation. Sintel. http://www.sintel.org, 2011.

[9] Blender Foundation. Blender, 2013.

[10] Blender Institute. Sintel, 2010.

[11] Blender Institute. Tears of steel, 2012.

[12] Böhler, A. Pet monster Valentine - Treasure Edition. http://www.thingiverse.com/thing:17331, 2012.

[13] Brown, B. J., and Rusinkiewicz, S. Global non-rigid alignment of 3-d scans. *ACM Transactions on Graphics 26*, 3 (July 2007), 21:1–21:9.

[14] Bunke, H. On a relation between graph edit distance and maximum common subgraph. *Pattern Recognition Letters 18* (1998), 689–694.

[15] Caudwell, A. gource. http://code.google.com/p/gource, 2009.

[16] Chang, W., Li, H., Mitra, N., Pauly, M., Rusinkiewicz, S., and Wand, M. Computing correspondences in geometric data sets. In *Eurographics Tutorial Notes* (2011).

[17] Chang, W., and Zwicker, M. Automatic registration for articulated shapes. *Computer Graphics Forum 27*, 5 (2008), 1459–1468.

[18] Chaudhuri, S., Kalogerakis, E., Guibas, L., and Koltun, V. Probabilistic reasoning for assembly-based 3D modeling. *ACM Transactions on Graphics 30*, 4 (2011), 35:1–35:10.

[19] Chaudhuri, S., and Koltun, V. Data-driven suggestions for creativity support in 3d modeling. *ACM Transactions on Graphics 26*, 6 (2010), 183:1–183:10.

[20] Chen, H.-T., Grossman, T., Wei, L.-Y., Schmidt, R., Hartmann, B., Fitzmaurice, G., and Agrawala, M. History assisted view authoring for 3D models. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2014), CHI '14, ACM.

[21] Chen, H.-T., Wei, L.-Y., and Chang, C.-F. Nonlinear revision control for images. *ACM Transaction on Graphics 30*, 4 (2011), 105:1–105:10.

[22] Christel, M. G., Smith, M. A., Taylor, C. R., and Winkler, D. B. Evolving video skims into useful multimedia abstractions. In *Proc. SIGCHI* (1998), pp. 171–178.

[23] Cour, T., Srinivasan, P., and Shi, J. Balanced graph matching. In *NIPS* (2006), pp. 313–320.

[24] Culum, A. Hailfire droid. http://www.cgwhat.com/hailfire-droid/, 2009. [Online; accessed 2011, Jon. 13].

[25] Doboš, J., Mitra, N. J., and Steed, A. 3D Timeline: Reverse engineering of a part-based provenance from consecutive 3d models. *Eurographics Symposium on Rendering 33*, 2 (2014).

[26] Doboš, J., and Steed, A. 3D Diff: an interactive approach to mesh differencing and conflict resolution. In *SIGGRAPH Asia 2012 Technical Briefs* (New York, NY, USA, 2012), SA '12, ACM, pp. 20:1–20:4.

[27] Drincic, N. [Shark] Modeling Process. http://www.3dm3.com/tutorials/shark/, 2004. [Online; accessed 2011, Jan. 13].

[28] Dubrovina, A., and Kimmel, R. Matching shapes by eigendecomposition of the laplace-beltrami operator. In *Proc. 3DPVT* (2010).

[29] Eppstein, D., Goodrich, M. T., Kim, E., and Tamstorf, R. Approximate topological matching of quad meshes. *The Visual Computer* (2009), 771–783.

[30] Gao, X., Xiao, B., Tao, D., and Li, X. A survey of graph edit distance. *Pattern Analysis and Applications 13* (2010), 113–129.

[31] Goralczyk, A. Creature. Creature Factory Blender Open Movie Workshop, vol. 2, 2008.

[32] Grabler, F., Agrawala, M., Li, W., Dontcheva, M., and Igarashi, T. Generating photo manipulation tutorials by demonstration. *ACM Trans. Graphics* (2009), 66:1–66:9.

[33] Grassard, F. Small Spaceship (low poly). http://www.blendswap.com/ /blends/vehicles/small-spaceship-low-poly/, 2011.

[34] Grossman, T., Matejka, J., and Fitzmaurice, G. Chronicle: Capture, exploration, and playback of document workflow histories. In *UIST* (2010).

[35] Harrison, S. M. A comparison of still, animated, or nonillustrated on-line help with written or spoken instructions in a graphical user interface. In *Proc. SIGCHI* (1995), pp. 82–89.

[36] Holten, D., and Van Wijk, J. J. Force-directed edge bundling for graph visualization. *Computer Graphics Forum 28*, 3 (2009), 983–990.

[37] Jack, B. Helmet modeling. http://www.bracercom.com/tutorial/content/ /Ironman_Helmet_Modeling/ironman_helmet_modeling.html, 2011. [Online; accessed 2011, Jan. 13].

[38] Kang, H.-W., Chen, X.-Q., Matsushita, Y., and Tang, X. Space-time video montage. In *Proc. IEEE Computer Society Conference on CVPR* (2006), pp. 1331–1338.

[39] Kelleher, C., and Pausch, R. Stencils-based tutorials: design and evaluation. In *Proc. SIGCHI* (2005), pp. 541–550.

[40] Kim, V. G., Lipman, Y., and Funkhouser, T. Blended intrinsic maps. *SIGGRAPH* (2011), 79:1–79:12.

[41] Kong, N., Grossman, T., Hartmann, B., Agrawala, M., and Fitzmaurice, G. Delta: a tool for representing and comparing workflows. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2012), ACM, pp. 1027–1036.

[42] Kuhn, C. Hyperspace Shuttle 1.0 and 2.0. http://www.blendswap.com/blends/vehicles/hyperspace-shuttle, http://www.blendswap.com/blends/vehicles/hyperspace-shuttle-2-0, 2012.

[43] Kurlander, D., and Feiner, S. A visual language for browsing, undoing, and redoing graphical interface commands. In *Visual Languages and Visual Programming*, S. K. Chang, Ed. Plenum Press, 1989, pp. 257–275.

[44] Lafreniere, B., Grossman, T., and Fitzmaurice, G. Community enhanced tutorials: improving tutorials with multiple demonstrations. In *Proceedings of the 2013 ACM annual conference on Human factors in computing systems* (New York, NY, USA, 2013), CHI '13, ACM, pp. 1779–1788.

[45] Leordeanu, M., and Hebert, M. A spectral technique for correspondence problems using pairwise constraints. In *International Conference on Computer Vision* (2005), pp. 1482–1489.

[46] Levenshtein, V. I. Binary codes capable of correcting spurious insertions and deletions of ones. *Probl. Inf. Transmission 1* (1965), 8–17.

[47] Li, W., Agrawala, M., and Salesin, D. Interactive image-based exploded view diagrams. In *Proc. Graphics Interface* (2004), pp. 203–212.

[48] Li, W., Grossman, T., and Fitzmaurice, G. GamiCAD: A gamified tutorial system for first time autocad users. In *UIST* (2012).

[49] Li, W., Ritter, L., Agrawala, M., Curless, B., and Salesin, D. Interactive cutaway illustrations of complex 3d models. *ACM Trans. Graphics* (2007), 31:1–31:11.

[50] "Lumpycow". Broken Chair. http://www.blendswap.com/3D-models/ /furniture/lumpycow_household_brokenchair, 2010.

[51] MakerBot Industries, LLC. Thingiverse. http://thingiverse.com, 2013.

[52] Matejka, J., Li, W., Grossman, T., and Fitzmaurice, G. CommunityCommands: Command recommendations for software applications. In *UIST* (2009).

[53] Mitra, N. J., Yang, Y.-L., Yan, D.-M., Li, W., and Agrawala, M. Illustrating how mechanical assemblies work. *ACM Trans. Graphics* (2010), 58:1–58:11.

[54] Nakamura, T., and Igarashi, T. An application-independent system for visualizing user operation history. In *UIST* (2008).

[55] Nakamura, T., and Igarashi, T. An application-independent system for visualizing user operation history. In *Proceedings of the 21st annual ACM symposium on User interface software and technology* (New York, NY, USA, 2008), UIST '08, ACM, pp. 23–32.

[56] Narayanan, N. H., and Hegarty, M. Multimedia design for communication of dynamic information. *Int. J. Hum.-Comput. Stud.* (2002), 279–315.

[57] Neuhaus, M., and Bunke, H. *Bridging the gap between graph edit distance and kernel machines*. World Scientific, 2007.

[58] Nyman, K. Ishtarian Matron Karl G Nyman. http://www.blendswap.com/blends/ /characters/ishtarian_matron_karl_g_nyman, 2010.

[59] Orbay, G., and Kara, L. B. Beautification of design sketches using trainable stroke clustering and curve fitting. *IEEE Transactions on Visualization and Computer Graphics 17*, 5 (May 2011), 694–708.

[60] Palmiter, S., and Elkerton, J. An evaluation of animated demonstrations of learning computer-based tasks. In *Proc. SIGCHI* (1991), pp. 257–263.

[61] Pavel, A., Berthouzoz, F., Hartmann, B., and Agrawala, M. Browsing and analyzing the command-level structure of large collections of image manipulation tutorials. Tech. rep., Electrical Engineering and Computer Sciences, University of California at Berkeley, October 2013.

[62] Pixologic. ZBrush. http://www.pixologic.com/zbrush, 2013.

[63] Pottmann, H., Wallner, J., Huang, Q.-X., and Yang, Y.-L. Integral invariants for robust geometry processing. *Comput. Aided Geom. Des. 26*, 1 (Jan. 2009), 37–60.

[64] Riesen, K., and Bunke, H. Approximate graph edit distance computation by means of bipartite graph matching. *Image and Vision Computing 27* (2009), 950–959.

[65] Roosendaal, T. Durian open movie project : Sintel full studio svn online. http://www.sintel.org/news/sintel-full-studio-svn-online, 2011.

[66] Rusinkiewicz, S., and Levoy, M. Efficient variants of the icp algorithm. *International Conference on 3D Digital Imaging and Modeling* (2001).

[67] Sharf, A., Blumenkrants, M., Shamir, A., and Cohen-Or, D. Snappaste: an interactive technique for easy mesh composition. *The Visual Computer 22* (2006), 835–844.

[68] Sharma, A., Horaud, R. P., Cech, J., and Boyer, E. Topologically-robust 3d shape matching based on diffusion geometry and seed growing. In *Computer Vision and Pattern Recognition* (2011), pp. 2481–2488.

[69] Sharma, A., von Lavante, E., and Horaud, R. P. Learning shape segmentation using constrained spectral clustering and probabilistic label transfer. In *European Conference on Computer Vision* (2010), pp. 743–756.

[70] Silva, E. D. R. Shaolin. http://www.blendswap.com/3D-models/characters/ /shaolin, 2011.

[71] Silva, S., Madeira, J., and Santos, B. S. Polymeco—an integrated environment for polygonal mesh analysis and comparison. *Computers & Graphics 33*, 2 (2009), 181 – 191.

[72] Su, S. L., Paris, S., Aliaga, F., Scull, C., Johnson, S., and Durand, F. Interactive visual histories for vector graphics. Tech. rep., Massachusetts Institute of Technology, 2009.

[73] Tate, B. Model a detailed high poly fire hydrant in 3ds max. http://cg.tutsplus.com/tutorials/autodesk-3ds-max/model-a-detailed-high-poly-fire-hydrant-in-3ds-max/, 2009. [Online; accessed 2011, Jan. 13].

[74] Tenenbaum, J. B., de Silva, V., and Langford, J. C. A global geometric framework for nonlinear dimensionality reduction. *Science 290* (2000).

[75] Terry, M., Kay, M., Van Vugt, B., Slack, B., and Park, T. Ingimp: introducing instrumentation to an end-user open source application. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2008), CHI '08, ACM, pp. 607–616.

[76] Thomas, L. Bunch of Keys. http://www.blendswap.com/blends/misc-objects/ /bunch-of-keys/, 2012.

[77] Torvalds, L., and Hamano, J. Git. http://git.scm.com, 2013.

[78] Vazquez, P. Durano model. Venom's Lab Blender Open Movie Workshop, vol. 4, 2009.

[79] VisTrails. VisTrails Provenance Explorer for Maya. http://www.vistrails.com/ /maya.html, 2010.

[80] Williamson, J. Character modeling in blender. http://cg.tutsplus.com/ /tutorials/blender/character-modeling-in-blender-basix/, 2010. [Online; accessed 2011, Jan. 13].

[81] Williamson, J. Ishtarian Woman. http://www.sintel.org/category/concept-art/ /feed/, 2010.

[82] Zeng, Y., Wang, C., Wang, Y., Gu, X., Samaras, D., and Paragios, N. Dense non-rigid surface registration using high-order graph matching. In *Computer Vision and Pattern Recognition* (2010), pp. 382–389.