MeshGit: Diffing and Merging Meshes for Polygonal Modeling

Jonathan D. Denning* Fabio Pellacini*†
*Dartmouth College †Sapienza University of Rome

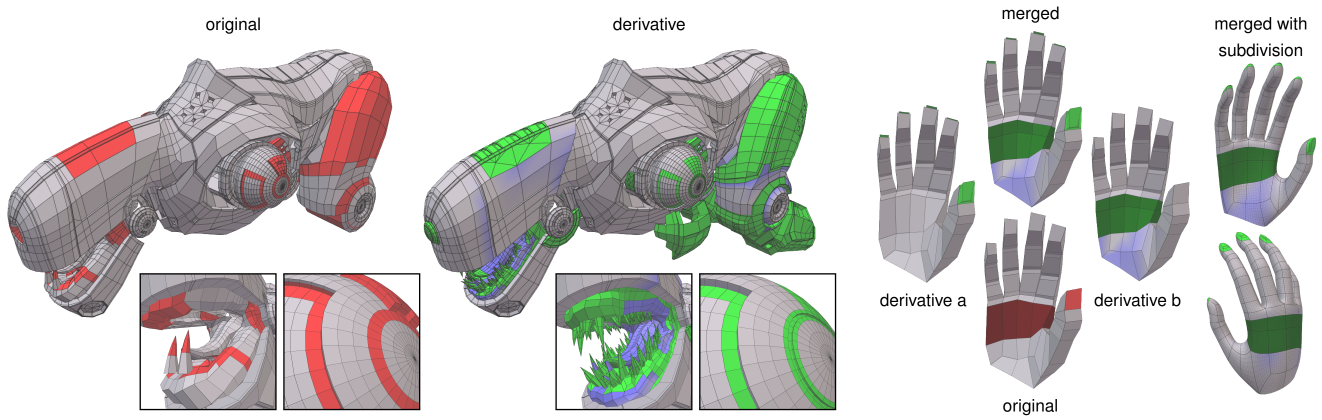


Figure 1: *Examples of diffing and merging polygonal meshes done automatically by* MeshGit*.* Left*: We visualize changes between two snapshots of the creation of a creature mesh as a* two-way diff*. The* derivative *mesh contains many changes, including significant changes in adjacency (red/green) and 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).* Right*: 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.*

Abstract

This paper presents *MeshGit*, a practical algorithm for diffing and merging polygonal meshes typically used in subdivision modeling workflows. Inspired by version control for text editing, we introduce the *mesh edit distance* as a measure of the dissimilarity between meshes. This distance is defined as the minimum cost of matching the vertices and faces of one mesh to those of another. We propose an iterative greedy algorithm to approximate the mesh edit distance, which scales well with model complexity, providing a practical solution to our problem. We translate the mesh correspondence into a set of mesh editing operations that transforms the first mesh into the second. The editing operations can be displayed directly to provide a meaningful visual difference between meshes. For merging, we compute the difference between two versions and their common ancestor, as sets of editing operations. We robustly detect conflicting operations, automatically apply non-conflicting edits, and allow the user to choose how to merge the conflicting edits. We evaluate *MeshGit* by diffing and merging a variety of meshes and find it to work well for all.

Keywords: polygonal modeling, geometry, diff and merge, visualization

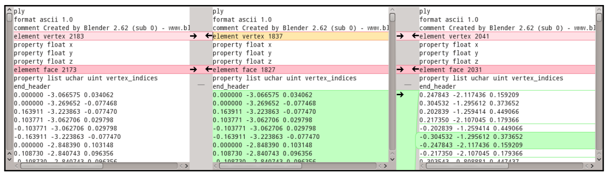
Links: DL PDF

ACM Reference Format
Denning, J., Pellacini, F. 2013. MeshGit: Diffing and Merging Meshes for Polygonal Modeling. ACM Trans. Graph. 32, 4, Article 35 (July 2013), 9 pages. DOI = 10.1145/2461912.2461942 http://doi.acm.org/10.1145/2461912.2461942.

1 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 [Chang et al. 2011], and smoothly blend portion of them using remeshing techniques [Sharf et al. 2006]. 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. [Doboš and Steed 2012] 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.



```
ply
format ascii 1.0
comment Created by Blender 2.62 (sub 0) - www.bl
element vertex 2183
property float x
property float y
property float z
element face 2173
property list uchar uint vertex_indices
end_header
0.000000 -3.066575 0.034062
0.000000 -3.269652 -0.077468
0.163911 -3.223863 -0.077470
0.103771 -3.062706 0.029798
-0.103771 -3.062706 0.029798
-0.163911 -3.223863 -0.077470
0.000000 -2.848390 0.103148
0.108730 -2.840743 0.096356
```

```
ply
format ascii 1.0
comment Created by Blender 2.62 (sub 0) - www.bl
element vertex 1837
property float x
property float y
property float z
element face 1827
property list uchar uint vertex_indices
end_header
0.000000 -3.066575 0.034062
0.000000 -3.269652 -0.077468
0.163911 -3.223863 -0.077470
0.103771 -3.062706 0.029798
-0.103771 -3.062706 0.029798
-0.163911 -3.223863 -0.077470
0.000000 -2.848390 0.103148
0.108730 -2.840743 0.096356
```

```
ply
format ascii 1.0
comment Created by Blender 2.62 (sub 0) - www.bl
element vertex 2041
property float x
property float y
property float z
element face 2031
property list uchar uint vertex_indices
end_header
0.247843 -2.117436 0.159209
0.304532 -1.295612 0.373652
0.202839 -1.259414 0.449066
0.217350 -2.107045 0.179366
-0.202839 -1.259414 0.449066
-0.304532 -1.295612 0.373652
-0.247843 -2.117436 0.159209
-0.217350 -2.107045 0.179366
```

Figure 2: *Three-way diff for text files. The original file is shown in the middle, and two derivatives are shown on the left and right.* MeshGit *follows a similar metaphor for mesh differences.*

***MeshGit*.** We present *MeshGit*, an algorithm that supports diffing and merging polygonal meshes. Figure 1 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 Fig. 2). Inspired by the string edit distance [Levenshtein 1965], 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, Fig. 6 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.

2 Related Work

Revision Control. Recent work by [Doboš and Steed 2012] 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 [Blender Foundation 2011]. *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 Fig. 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 in [Chang et al. 2011]. Some of these methods [Chang and Zwicker 2008; Brown and Rusinkiewicz 2007] are variants of iterative closest point [Rusinkiewicz and Levoy 2001] 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 [Leordeanu and Hebert 2005; Sharma et al. 2010] to determine a sparse correspondence between two shapes. [Sharma et al. 2011] 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. 2010] 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. 2011] 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. 2009] 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 [Neuhaus and Bunke 2007]. [Bunke 1998] 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 [Gao et al. 2010] 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 2009] 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. 2006] 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. [Sharf et al. 2006] 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 2010] and [Chaudhuri et al. 2011] 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 2010] let the users explore their undos histories. [Denning et al. 2011] shows rich visual histories of mesh construction by highlighting and visually annotating changes to the mesh. [Chen et al. 2011] 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 Fig. 3.

3 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 [Levenshtein 1965] 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 [Brown and Rusinkiewicz 2007] or a shape-based alignment [Dubrovina and Kimmel 2010] 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 Fig. 4, 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

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

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

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 adjacencies 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.

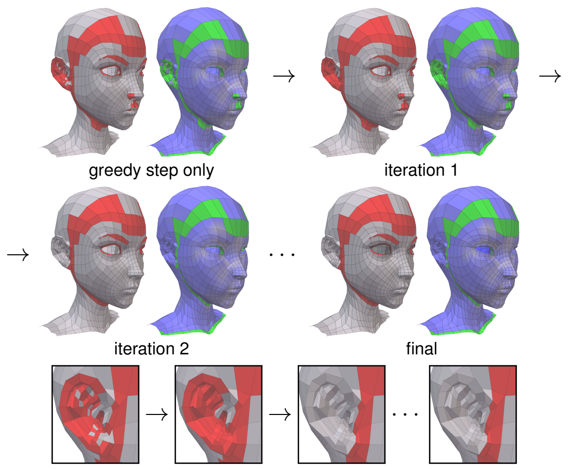


Figure 3: *Two-way diffs taken for subsequent steps of our iterative algorithm, where each iteration refines the differences to become more precise. These two versions were independently edited, so neither is the derivative of the other. This is the worst case for diffing. Nonethless* MeshGit *handles this case well.*

4 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 [Neuhaus and Bunke 2007; Bunke 1998], that is known to be NP-Hard in the general case . And, while many polynomial-time graph-matching approximation algorithms have been proposed [Gao et al. 2010], 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.

4.1 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 3 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 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 Fig. 3, 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.

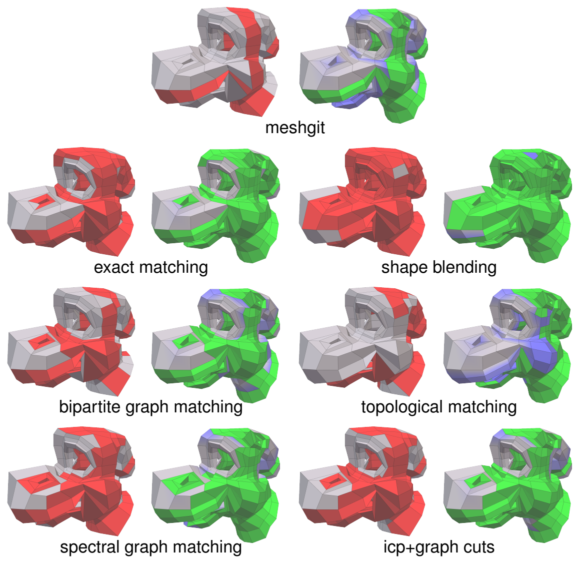


Figure 4: *Two-way diffs from different matching algorithms. Compared to* MeshGit*, the results of the prior methods contain more mismatched adjacencies, because the methods either do not account for adjacencies, do not account for geometry changes, or produce a fuzzy matching.*

4.2 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 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.

4.3 Discussion

Comparison. Fig. 4 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

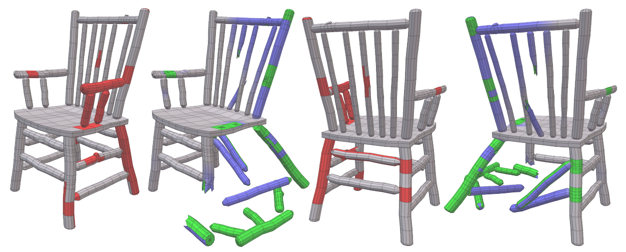


Figure 5: *Two-way diff showing the main limitation of our approach. While* MeshGit *detects most edits correctly, it fails to properly capture edits in the back leg since both geometry and adjacencies change significantly.*

matching [Riesen and Bunke 2009], spectral graph matching [Cour et al. 2006], shape blending [Kim et al. 2011], topological matching [Eppstein et al. 2009], and iterative closest point with graph cuts [Chang and Zwicker 2008]. 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 Fig. 7[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 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 Fig. 5, 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 [Brown and Rusinkiewicz 2007] 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.

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

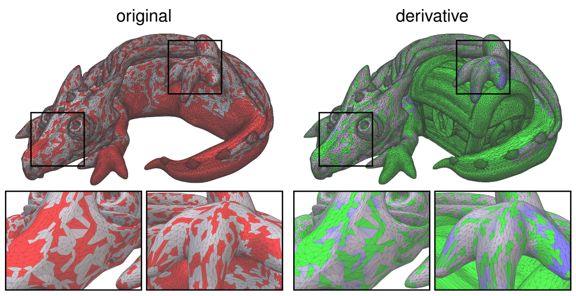


Figure 6: *Two-way diff of meshes with similar shape but different adjacencies due to remeshing. While* MeshGit *computes the diff correctly, the resulting visualization might not be as informative since in this workflow artists focus only on geometry changes.*

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 6 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.

5 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 Fig. 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 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 Fig. 1.b. We use a color scheme similar to the above, but the brightness of the color indicates from which

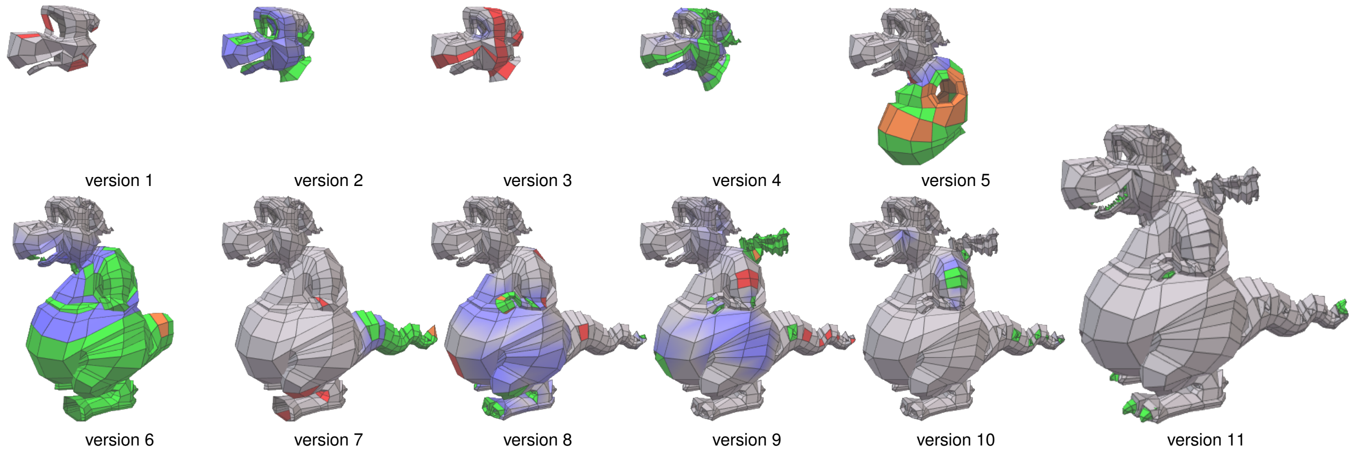


Figure 7: MeshGit *can be used to visualize construction sequences, here shown on twelve snapshots. Faces are green if added to the current snapshot or changed from the previous, red if deleted in the next or changed, and orange if added and then deleted or changed both times.* Version 11 *is enlarged to show better the fine features added, namely the teeth, claws on hand and feet, and the horn at tip of tail.*

derivative the operation comes. When a face has been modified in both derivatives it is indicated in yellow (Fig. 9).

An artist can also use *MeshGit* to visualize the *progression* of work on a mesh, as shown in Fig. 7. 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* [Denning et al. 2011], 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 Fig. 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 Fig. 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 Fig. 8. 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 Fig. 8 show the resulting merged mesh with colors indicating the applied transformations. On the right we show recursively applying Catmull-Clark subdivision rules twice to

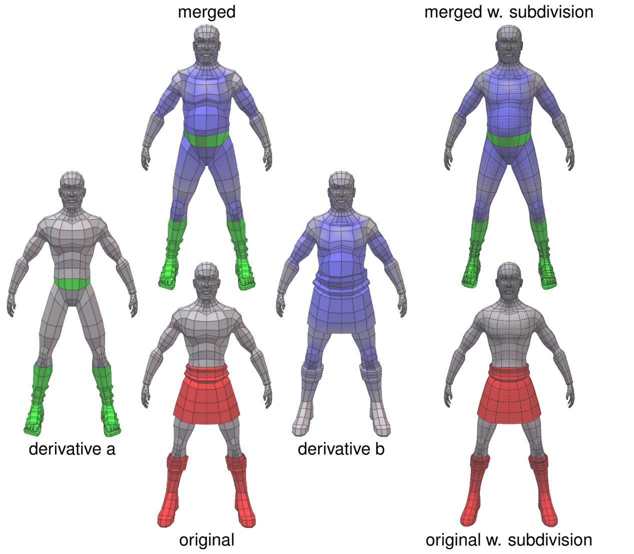


Figure 8: *Automatic merge of non-conflicting edits that affect the adjacencies (derivative a) and geometry (derivative b). We show both the original and merge after applying Catmull-Clark subdivision to show that* MeshGit *maintains consistent face adjacencies.*

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 9 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 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

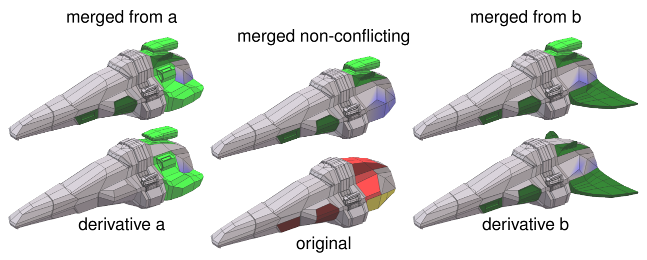


Figure 9: MeshGit *detects conflicting mesh differences, visualized in yellow, between the derivatives, and partitions the changes into groups that can be applied individually. In this case, the expanded base of* derivative a *and added wings of* derivative b *are conflicting. All non-conflicting changes 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.*

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.

6 Results

We tested *MeshGit* on a variety of meshes whose statistics are collected in Tab. 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.

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.

Timing. As summarized in Tab. 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.

Model	Reference	Fig.	Number of Faces			Time
			original	ver. 1	ver. 2	
chairs	[Lumpycow]	5	3290	3951	—	4.7s
creature	[Goralczyk]	1	11475	17433	—	14.5s
dragon	[Böhler]	6	—	88028	96616	307.9s
durano 1	[Vazquez]	7	276	520	520	0.5s
durano 4		7	786	906	1716	0.4s
durano 7		7	1930	2186	2772	1.5s
durano 10		7	3078	3722	—	1.2s
hand	[Williamson]	1	199	209	209	0.1s
keys	[Thomas]	10	—	1652	1854	6.7s
shaolin	[Silva]	8	1850	1850	2158	2.4s
sintel	[Blender]	3	—	1810	1712	2.7s
spaceship	[Grassard]	9	1827	2173	2031	0.9s
shuttle	[Kuhn]	10	166974	193970	—	585.3s
woman orig.	[Williamson]	10	13984	—	—	—
woman deriv	[Nyman]	10	—	8616	—	33.7s

Table 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.*

Figure 10 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, [Doboš and Steed 2012] 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.

7 Conclusion and Future Work

This paper 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 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 *MeshGit* to automatically generate mesh variations from only a few models by automatically applying different edits combination.

8 Acknowledgements

We would like to thank the authors of the meshes used and the authors of the matching algorithms for providing source code and support. This work was partially supported by NSF (CCF-0746117), Intel, and the Sloan Foundation.

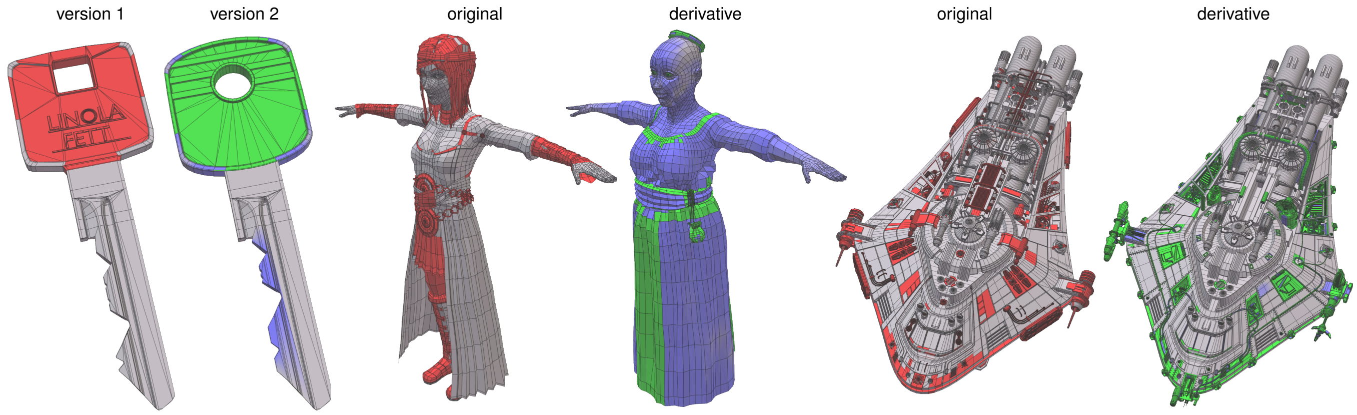


Figure 10: MeshGit *handles well cases with irregular adjacencies (left), with significant geometric and adjacency changes (middle), and with high vertex and face counts (167k and 194k polygons from 2254 and 3352 original components respectively). All six of these meshes are composed of both triangles and quads.*

References

BLENDER FOUNDATION, 2011. Sintel. www.sintel.org.

BROWN, B. J., AND RUSINKIEWICZ, S. 2007. Global non-rigid alignment of 3-d scans. *ACM Transactions on Graphics 26*, 3 (July), 21:1–21:9.

BUNKE, H. 1998. On a relation between graph edit distance and maximum common subgraph. *Pattern Recognition Letters 18*, 689–694.

CHANG, W., AND ZWICKER, M. 2008. Automatic registration for articulated shapes. *Computer Graphics Forum 27*, 5, 1459–1468.

CHANG, W., LI, H., MITRA, N., PAULY, M., RUSINKIEWICZ, S., AND WAND, M. 2011. Computing correspondences in geometric data sets. In *Eurographics Tutorial Notes*.

CHAUDHURI, S., AND KOLTUN, V. 2010. Data-driven suggestions for creativity support in 3d modeling. *ACM Transactions on Graphics 26*, 6, 183:1–183:10.

CHAUDHURI, S., KALOGERAKIS, E., GUIBAS, L., AND KOLTUN, V. 2011. Probabilistic reasoning for assembly-based 3D modeling. *ACM Transactions on Graphics 30*, 4, 35:1–35:10.

CHEN, H.-T., WEI, L.-Y., AND CHANG, C.-F. 2011. Nonlinear revision control for images. *ACM Transaction on Graphics 30*, 4, 105:1–105:10.

COUR, T., SRINIVASAN, P., AND SHI, J. 2006. Balanced graph matching. In *NIPS*, 313–320.

DENNING, J. D., KERR, W. B., AND PELLACINI, F. 2011. Meshflow: interactive visualization of mesh construction sequences. *ACM Transaction on Graphics 30*, 4, 66:1–66:8.

DOBOŠ, J., AND STEED, A. 2012. 3D Diff: an interactive approach to mesh differencing and conflict resolution. In *SIGGRAPH Asia 2012 Technical Briefs*, ACM, New York, NY, USA, SA '12, 20:1–20:4.

DUBROVINA, A., AND KIMMEL, R. 2010. Matching shapes by eigendecomposition of the laplace-beltrami operator. In *Proc. 3DPVT*.

EPPSTEIN, D., GOODRICH, M. T., KIM, E., AND TAMSTORF, R. 2009. Approximate topological matching of quad meshes. *The Visual Computer*, 771–783.

GAO, X., XIAO, B., TAO, D., AND LI, X. 2010. A survey of graph edit distance. *Pattern Analysis and Applications 13*, 113–129.

KIM, V. G., LIPMAN, Y., AND FUNKHOUSER, T. 2011. Blended intrinsic maps. *SIGGRAPH*, 79:1–79:12.

LEORDEANU, M., AND HEBERT, M. 2005. A spectral technique for correspondence problems using pairwise constraints. In *International Conference on Computer Vision*, 1482–1489.

LEVENSHTEIN, V. I. 1965. Binary codes capable of correcting spurious insertions and deletions of ones. *Probl. Inf. Transmission 1*, 8–17.

NEUHAUS, M., AND BUNKE, H. 2007. *Bridging the gap between graph edit distance and kernel machines*. World Scientific.

RIESEN, K., AND BUNKE, H. 2009. Approximate graph edit distance computation by means of bipartite graph matching. *Image and Vision Computing 27*, 950–959.

RUSINKIEWICZ, S., AND LEVOY, M. 2001. Efficient variants of the icp algorithm. *International Conference on 3D Digital Imaging and Modeling*.

SHARF, A., BLUMENKRANTS, M., SHAMIR, A., AND COHEN-OR, D. 2006. Snappaste: an interactive technique for easy mesh composition. *The Visual Computer 22*, 835–844.

SHARMA, A., VON LAVANTE, E., AND HORAUD, R. P. 2010. Learning shape segmentation using constrained spectral clustering and probabilistic label transfer. In *European Conference on Computer Vision*, 743–756.

SHARMA, A., HORAUD, R. P., CECH, J., AND BOYER, E. 2011. Topologically-robust 3d shape matching based on diffusion geometry and seed growing. In *Computer Vision and Pattern Recognition*, 2481–2488.

VISTRAILS, 2010. VisTrails Provenance Explorer for Maya. www.vistrails.com/maya.html.

ZENG, Y., WANG, C., WANG, Y., GU, X., SAMARAS, D., AND PARAGIOS, N. 2010. Dense non-rigid surface registration using high-order graph matching. In *Computer Vision and Pattern Recognition*, 382–389.