MeshGit: Diffing and Merging Polygonal Meshes

Jonathan D. Denning, Fabio Pellacini
Dartmouth Computer Science Technical Report TR2012-722

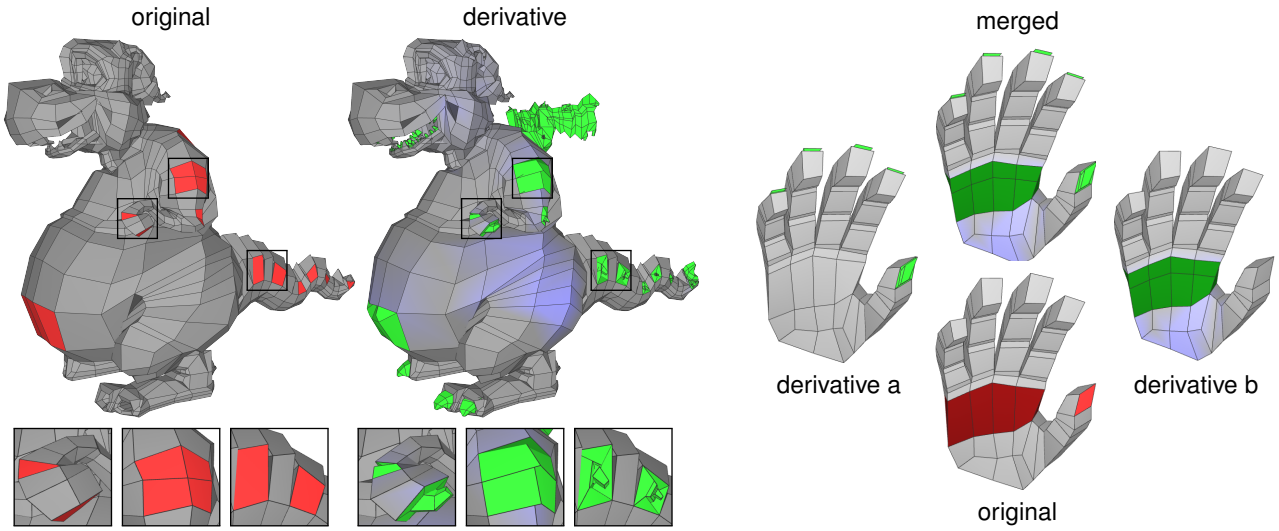

Figure 1: *Example of diffing and merging polygonal meshes done automatically by* MeshGit*. We determine mesh differences by computing the minimal set of operations needed to change one mesh into another.* Left: *Visual difference of two versions of a model in a series. We visualize mesh differences by showing the models side-by-side. On the original, we highlight in red faces that are either deleted or have changed. On the derivative, we highlight in green faces that have been added or have changed, and color face vertices in blue if they have moved.* Right: MeshGit *can automatically merge edits from different derivatives or detect if edits are conflicting. Here we visualize a three-way diff between two derivates and an original, and the merged mesh. Darker and lighter colors indicate from which derivative edits come.*

Abstract

This paper presents *MeshGit*, a practical 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. 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.

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 [Apache 2012] and Git [Torvalds and Hamano 2012] 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 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 the shape of complex meshes [Chang et al. 2011], and smoothly blend portion of them using remeshing techniques [Sharf et al. 2006]. While these algorithms could be adapted to diff and merge complex meshes, they are not directly applicable to our problem since we want precise diffs capturing the differences and robust merges that do not alter the mesh adjacencies.

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.

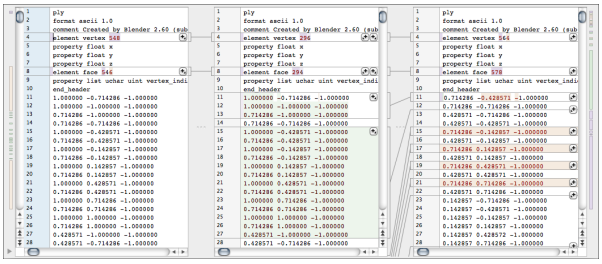

Figure 2: *Three-way diff for text files. The original file is shown in the middle, and two derivatives are shown on the left and right.* MeshGit *follows a similar metaphor for mesh differences.*

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, addition and deletion and face addition 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, as is done explicitly in Git [Torvalds and Hamano 2012] and implicitly in other systems [Apache 2012]. 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.

Contributions. In summary, this paper proposes a practical framework for diffing and merging polygonal meshes typically used in subdivision surface modeling. *MeshGit* does this by (1) defining a mesh edit distance and describing a practical algorithm to compute 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. We evaluate *MeshGit* for a variety of meshes and found it to work well for all. 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 assets by explicitly splitting an asset into components. 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. As a practical demonstration, the open source movie Sintel [Blender Foundation 2011] was produced using similar techniques. The assets of the film, such as the characters, props, and sets, are stored in separate files and linked together in scene files. With the files stored in an Subversion repository, the team of artists are able to easily share edits to individual files. While these types of systems allow for merging changes of different components or files, any change to the same component, regardless of the location of the edit on the component, marks the entire component in conflict. *MeshGit* supports arbitrary edits on meshes without explicitly splitting them in components, allowing the merging of changes onto the same mesh (see Fig. 1).

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]. The graph edit distance is defined as the minimum cost of changing one graph into another, where each change carries a cost. In [Bunke 1998], computing the graph edit distance with a special cost function is shown to be equivalent to the maximum common subgraph-isomorphism problem, the problem of finding the largest subgraph of one graph

that is isomorphic to a subgraph of another. While computing the maximum common subgraph is known to be NP-hard, [Riesen and Bunke 2009] shows that good approximations can be computed using polynomial time algorithms to solve the corresponding graph matching problem.

Several approximation algorithms have been proposed that differ in the expected properties of the input graph. We refer the reader to [Neuhaus and Bunke 2007; 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. 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] and [Robles-Kelly and Hancock 2003] 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. In summary, we found that the current approximation algorithms for computing the graph edit distance either scale poorly with the size of the input meshes or produce poor results since they approximate too heavily the adjacency costs. *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. 4.

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 matches that 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. In this section, we will define the mesh edit distance that we will show how to compute in the following section.

3.1 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 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 define a cost function, the *mesh edit distance*, and pick the matching with minimum cost. The mesh edit distance is comprised of three terms.

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

Geometric Cost. 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 of matching e to e' is given by

$$C(e \leftrightarrow e') = \frac{||\mathbf{x}_e - \mathbf{x}_{e'}||}{||\mathbf{x}_e - \mathbf{x}_{e'}|| + 1} + (1 - \mathbf{n}_e \cdot \mathbf{n}_{e'}), \qquad (1)$$

where $\mathbf{x}_e$ and $\mathbf{n}_e$ are the position and normal of the element e respectively*.

The position term is an increasing, limited function on the Euclidean distance between the elements positions. This formulation 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. The inclusion of the orientation helps in cases where many small elements are located close to one another.

Adjacency Cost. The geometric costs alone are not sufficient to produce intuitive visual differences since it does not take into account changes in the elements adjacencies. For example, Fig. 3 shows two ways to match two meshes. These two matchings differ only by which faces are matched. In one case, the adjacency of the first mesh is maintained, while in the other it is not. In this example, the geometric cost would not distinguish between these

*Note that our implementation assumes that vertices are defined with respect to the same coordinate system during editing, since this is common practice in mesh modeling and since most modeling software stores transformation matrices separately. If necessary, we could run an initial global alignment based on ICP [Brown and Rusinkiewicz 2007]. Furthermore, we normalize the size of both meshes to the average edge length so that vertex distances are normalized to the size of the mesh.

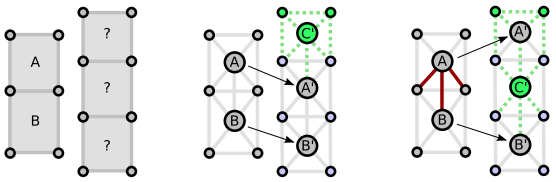

Figure 3: *Left: Two meshes to match. Middle and Right: Two possible matchings of the two meshes illustrated as graphs, with large circles denoting faces, small circles denoting vertices, and edges denoting adjacency. The green dotted edges are unmatched, and the red solid edges are mismatched. In this case, the right matching costs more as the difference between these two matches is solely in the adjacency costs for matching A to A' and B to B'.*

two cases. The top-right subfigure in Fig. 5, 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 in M'. We consider all adjacencies of faces and vertices (i.e., face-to-face, face-to-vertex, and vertex-to-vertex). If either e_1 or e_2 are unmatched, the adjacency cost is a constant β divided by the size of the local neighborhoods $A(e_1, e_2)$ around the pair. The latter is computed as the sum of the number of elements adjacent to e_1 and e_2. If both e_1 and e_2 are matched, but their matching elements are not adjacent, the adjacency cost for the pair is γ divided by size of the local neighborhoods. There is no cost for matched adjacencies. We normalize the constants by the size of the local neighborhoods, so the maximum cost for elements with a large number of adjacencies (such as extraordinary vertices or poles) is the same as those with only a few adjacencies (such as vertices at the edges of the model).

Overall Cost. With the costs defined above, the overall cost of a matching O between meshes M and M' can be as the sum of the unmatched, geometric and adjacency costs, as follows

$$\begin{aligned} C(O) = {} & \alpha(N_u + N'_u) + \\ & + \sum_{\{e \leftrightarrow e'\}} \left[\frac{||\mathbf{x}_e - \mathbf{x}_{e'}||}{||\mathbf{x}_e - \mathbf{x}_{e'}|| + 1} + (1 - \mathbf{n}_e \cdot \mathbf{n}_{e'}) \right] + \\ & + \sum_{(e_1,e_2) \in P_u} \frac{\beta}{A(e_1, e_2)} + \sum_{(e_1,e_2) \in P_m} \frac{\gamma}{A(e_1, e_2)} \end{aligned} \quad (2)$$

where N_u and N'_u are the number of unmatched elements in M and M' respectively, $\{e \leftrightarrow e'\}$ is the set of matched elements, and P_u and P_m are the sets of adjacent element pairs in M and M' that are either unmatched or have different adjacency respectively. All results in this paper are obtained with $\alpha = 2.5$, $\beta = 1.0$, and $\gamma = 3.5$. We determined these constants by matching a large variety of meshes and choosing the values that produce the most informative visualizations.

4 Algorithm

Equivalent Graph Matching Problems. Minimizing the mesh edit distance to determine the optimal mesh matching is equivalent to solving a maximum common subgraph isomorphism problem on appropriately constructed graphs. 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 Equation 2 over the graph. Unfortunately, exact sub-graph matching is known to be NP-Complete in the general case. And, while many polynomial-time graph-matching approximation algorithms have been proposed, 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 the cost function Equation 2.

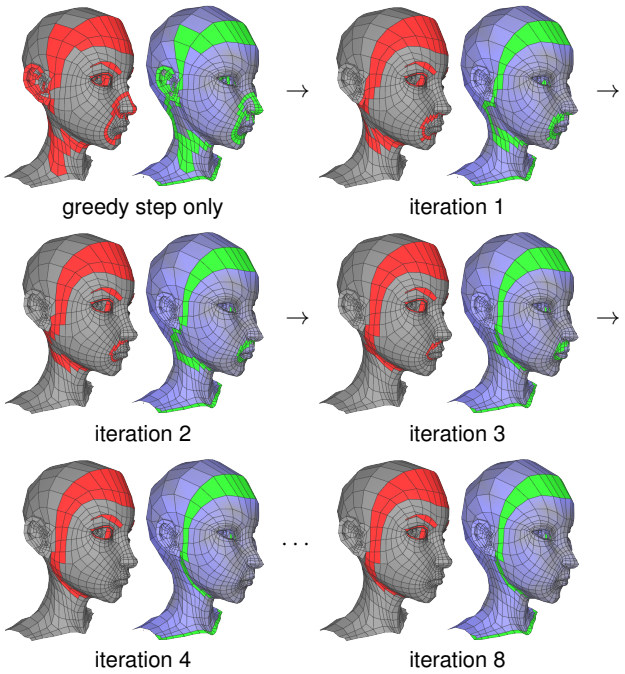

Figure 4: *Two-way diffs taken for subsequent steps of our iterative algorithm, where each iteration refines the differences to become more precise. The top-left subfigure is the matching result after the greedy step but before the backtracking step. Note that these two versions were independently edited, so neither is the derivative of the other. This is the worst case for diffing. Nonethless* MeshGit *handle this case well.*

4.1 Iterative Greedy Algorithm

We initialize the matching O with all elements in M and M' unmatched. This initial matching represents a transformation from M to M' where all elements of M are removed and all elements of M' are added. By construction, this is the highest cost matching. 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 assigning (or removing assignment) 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 until the algorithm converges, where convergence is decided when the current matching is the same as a previous matching to within a $1 - \epsilon$ of the total number of elements. A small, non-zero value for ϵ allows for the detection of small oscillations in the matching of complex meshes, which is present in the meshes for Figs. 6 and 10. In our implementation, we set $\epsilon = 0.002$. Figure 4 illustrates how O evolves for subsequent iterations. Now we describe these two steps in more detail.

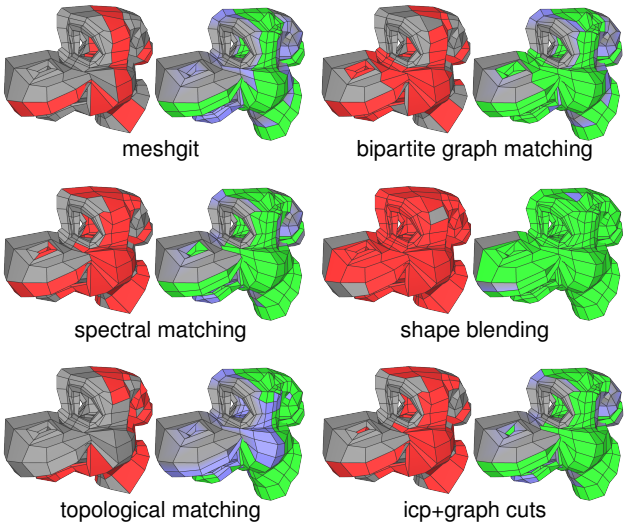

Figure 5: *Two-way diffs from different matching algorithms. Compared to* MeshGit*, the results of the prior methods contain more mismatched adjacencies, because the methods either do not account for adjacencies, do not account for geometry changes, or produce a fuzzy matching.*

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.

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. 4, 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 behavior is not due to the mesh edit distance we introduced, but to the greedy nature of the above algorithm. In our case though, we found that there are two common cases that are responsible for the vast majority of high cost matchings, and that we can easily prevent them by backtracking.

The first case is one of faces that are matched together with their vertices, but where the vertex order is different in the two meshes. 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'. In this example, the matching incurs heavy adjacency costs with adjacent faces over the (b', c') and (c', a') edges. We can easily eliminate these cases by removing assignments of all vertices and faces involved.

The second case is when a group of connected faces that have been matched meets the rest of the mesh over mismatched adjacencies. An example of this behavior is shown in the *greedy step only* subfigure of Fig. 4, where there are small groups of matched faces, such as on the earlobe, that are disconnected from the rest of the mesh due to mismatched adjacencies. These disconnected groups are due to suboptimal initial greedy assignments, favored by the geometric term, in heavily sculpted meshes that also have edits that affect adjacencies. These groups of faces are always quite small relative to the size of the whole connected component upon which they reside. We eliminate small patches of disconnected faces by removing the assignments of all their elements. We choose to backtrack in case the group's size is less than 10% the size of the connected component.

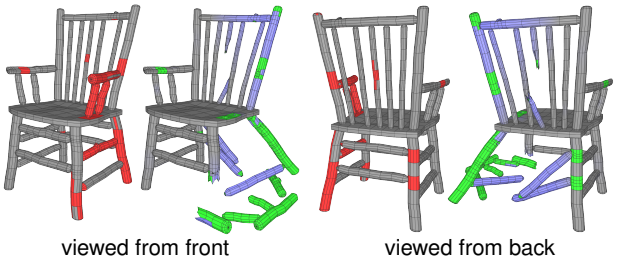

Figure 6: *Two-way diff showing the main limitation of our approach. While* MeshGit *detects most edits correctly, it fails to properly capture edits in the back leg since both geometry and adjacencies change significantly.*

In our backtracking step, we detect and remove assignments of elements as described in both previous cases. Since performing these two adjustments might leave assignments of high costs, we also remove assigment of elements that have mismatched adjacencies and vertices that have unmatched adjacencies. We found that if we run the greedy algorithm after this backtracking step, we obtain improved matchings in the case of heavily sculpted meshes with adjacency changes. We run both steps in an iterative fashion.

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 ($k = 50$ in our tests). This reduces the computation time considerably. Furthermore, we compute the change in the cost function with local updates only, since assigning or removing the assignment of a pair of elements only affects the costs in their local neighborhoods. With these two optimizations, our algorithm has a time complexity of $O(n \log n)$, resulting in a practical solution that we found to perform well even with large meshes of several thousand vertices and faces.

The iterative greedy algorithm repeats until the algorithm converges. For the majority of meshes, the algorithm converged within only a couple of iterations. For meshes with complex edits (e.g., Figs. 4, 6, and 10), the algorithm requires more iterations. Despite the higher number of iterations in these cases, we found the iterative greedy algorithm is a practical solution to approximate the mesh edit distance and produce intuitive visualizations of mesh differences for all the meshes we tested, as seen in Tab. 1.

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 delete 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. Figure 5 shows the results of using different shape matching algorithms to show visual differences. We included our method, bipartite graph matching [Riesen and Bunke 2009], spectral 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; 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[†].

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. 6, 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.

[†]Version 4 in Fig. 5 was modified to contain only the largest connected component, since the shape blending algorithm requires a single connected mesh.

5 Diffing and Merging

In this section, we describe how to visualize the differences between meshes and how to merge two sets of differences into a single mesh.

5.1 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. We have experimented with many color schemes and report here the ones we found the most informative.

Two-way Diff. A two-way diff can illustrate the differences between two versions of a mesh. For example, Fig. 6 shows the computed differences between two chair meshes. In a two-way diff, the original mesh M is displayed on the left and the derivative mesh M' on the right. Deleted faces in M (unmatched or with mismatched adjacencies in M) are indicated by coloring them red. Added faces in M' (unmatched or with mismatched adjacencies in M') are colored green. In our visualizations, we simplify the presentation by not drawing the vertices directly. Instead, the color of a vertex is linearly interpolated across the adjacent faces, unless the face has been colored red or green. A vertex in M' is colored blue if it has moved, where the saturation of blue indicates the strength of the geometric change with gray indicating no change. Unmodified faces and vertices are colored gray.

Three-way Diff. When a mesh M has two derived versions, M^a and M^b, a three-way mesh diff can illustrate the changes between the derivatives and the original, allowing for a comparison of the two sets of edits. The three meshes are shown side-by-side, with the original M in the middle, M^a on the left, and M^b on the right. We use a similar color scheme as with a two-way diff, but the brightness of the color indicates from which derivative the operation comes: light red and green are for M^a, and dark red and green are for M^b. When a face f in M has been modified in both derivatives, this overlap in change is indicated by coloring yellow f in M. An example of a three-way diff is shown in Fig. 1.b.

Series Diff. An artist can also use *MeshGit* to visualize the progression of work on a mesh, as shown in Fig. 7. In this example, twelve snapshots of the model were saved during its construction, starting from the head, then working the body, and then adding the tail, arms, wings, and finally some extra details. Each snapshot is visualized similarly to a three-way diff. For each snapshot, a face f in M is colored green if it was added, red if it is deleted, and orange

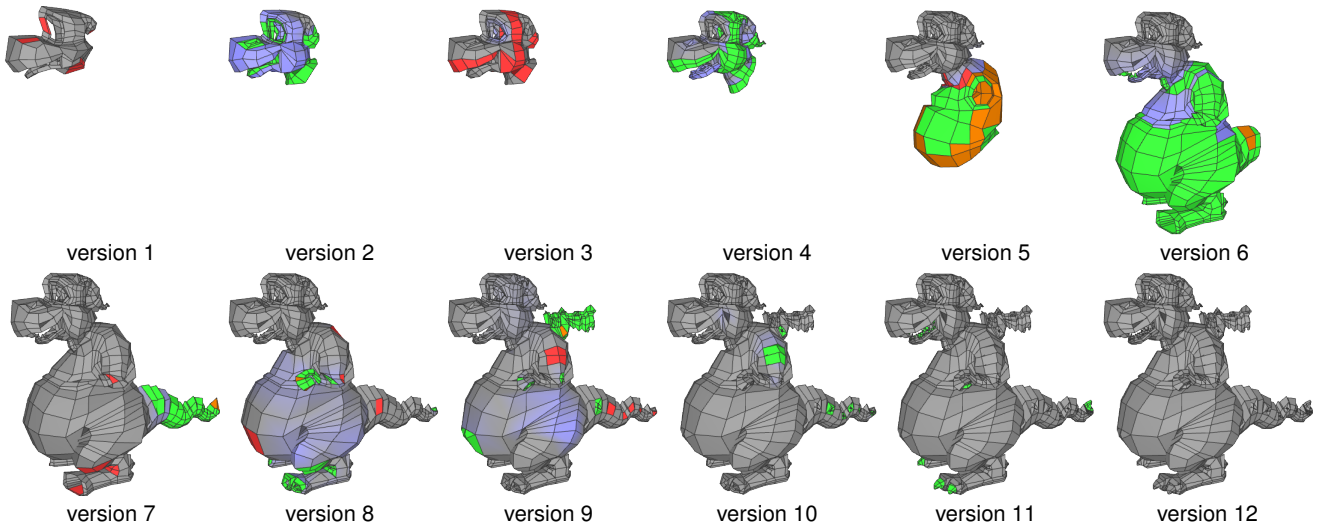

Figure 7: *Twelve snapshots of a model taken during its construction.* MeshGit *can also be used to visualize construction sequences. A face is highlighted in green if it was added to the current snapshot or changed from the previous, red if it is deleted in the next snapshot or changed, and orange if was added and then deleted or changed both times.*

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.

5.2 Mesh Edit Merge

Merging Editing Operations. 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.

Merging Workflow. *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.

Merging Non-Conflicting Edits. 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 demonstrate that adjacencies are well maintained.

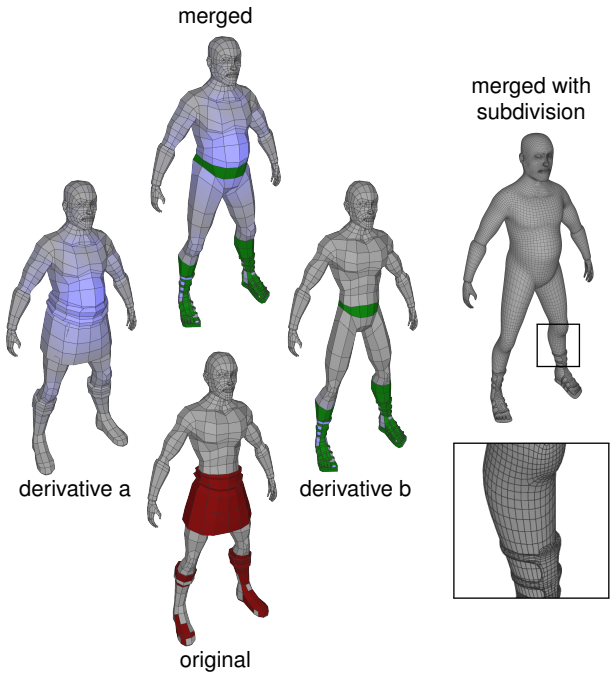

Figure 8: *Example of automatic merging of non-conflicting edits that affect the geometry (derivative a) and adjacencies (derivative b). The original mesh is completely connected. The merged mesh is shown with coloring (top) and after applying Catmull-Clark subdivision rules (far right). The subdivided merged mesh illustrates that* MeshGit *maintains consistent face adjacencies.*

Reducing Conflicts. In our previous definition, if even a single mesh transformation is in conflict, none of the edits can be safely applied. We could ask the user to resolve the conflict by picking single transformations from each set, a situation that would be obviously too cumbersome. To reduce the number of conflicts and reduce the granularity of users' decisions, we partition mesh transformations into groups that can be safely applied individually. This is akin to grouping text characters into lines in text merging.

The key observation is that edits that cause adjacency changes

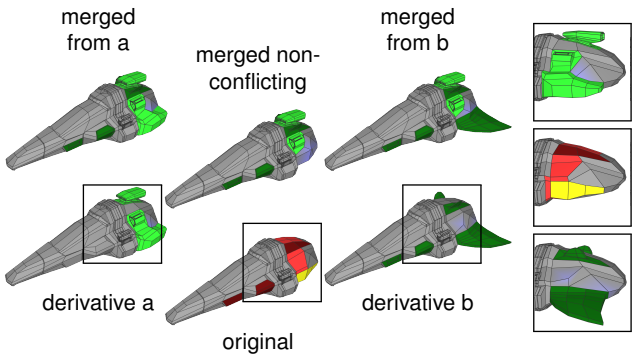

Figure 9: MeshGit *detects conflicting mesh differences, visualized in yellow, between the derivatives, and partitions the changes into groups that can be applied individually. In this set of meshes, the expanded base of* derivative a *and added wings of* derivative b *are conflicting edits. All non-conflicting changes are applied automatically. On the left, the bottom row shows a three-way diff of the meshes, and the top row shows three possible ways of resolving the conflicting mesh edits. The insets on the right show the input meshes from a different angle.*

can be bounded by the outer edges of the involved faces. For example, an artist may inset a patch of faces on a mesh. One way to represent this inset operation is to delete the patch from the original mesh and replace it with an inset version of the patch, connecting it to the rest of the mesh in exactly the same way the original patch was. The vertices and edges surrounding the original patch remain present during the entire process. We use this observation to guide our edit partitioning rule. An edge between two vertices is on the boundary if the adjacent faces are unmatched (added or deleted) but the vertices of the edge are matched. These boundary edges form a boundary loop. We then partition the sets of editing operations to within these boundary loops, essentially grouping edits in small non-intersecting patches. We can detect conflicts between the revisions at the granularity of these patches and ask users to resolve the conflicts at the same granularity.

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 edits. *MeshGit* successfully detected the conflicts to the body and merged all other changes automatically. 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, shown throughout the paper, running our implementation on a quad-core 2.93GHz Intel Core i7 with 16GB RAM and an ATI Radeon HD 5750 graphics card. All of the meshes and source code are available as supplemental material.

Models. Our implementation takes as input meshes containing vertex positions and the vertex indices for the faces. We chose meshes from different sources and when available included all versions. The *creature* (Fig. 10; [Goralczyk 2008]) and *durano* (Figs. 1,7; [Vazquez 2009]) meshes are from two series of saved

Model	Fig.	Number of Faces			Orig. → Ver. 1	
		original	ver. 1	ver. 2	time,	iters
chairs	6	3290	3951	—	35.1s,	10
creature	10	11475	17433	—	79.4s,	3
durano 1	7	276	520	520	0.8s,	2
durano 4	7	786	906	1716	1.3s,	1
durano 7	7	1930	2186	2772	3.2s,	1
durano 10	7	3078	3722	3722	5.4s,	1
hand	1	199	209	209	0.3s,	1
shaolin	8	1850	1850	2158	2.7s,	1
sintel	4	1810	1712	—	11.3s,	8
spaceship	9	1827	2173	2031	3.0s,	1

Table 1: *Statistics of input meshes and computing the mesh edit distance. The right two columns show the total time and the number of iterations of the iterative greedy algorithm to compute the mesh edit distance between the original mesh and a derivative.*

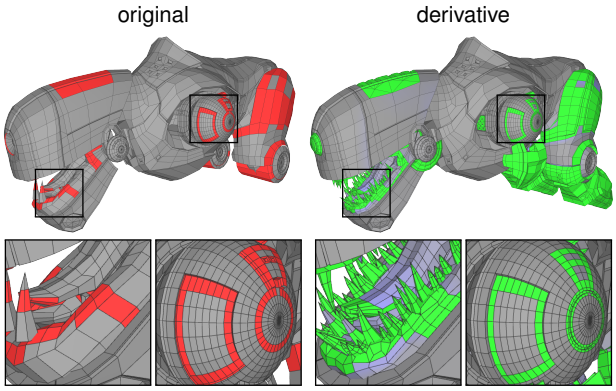

Figure 10: *Mesh diff of two snapshots in a construction sequence. The original mesh has 42 components and over 11k faces, and the derivative has 122 components and over 17k faces. Along with the large number of additional components,* MeshGit *automatically detects and highlights the many refining and reshaping changes such as on the gumline and in the insets on shoulder ball.*

snapshots taken through the mesh construction history. The *sintel* (Fig. 4; [Blender Foundation 2011]) meshes are a set of mesh variations, where there is no clear original mesh. The *chair* (Fig. 6; ["Lumpycow" 2010]) meshes are an ancestor mesh and one derivative mesh. For the *hand* (Fig. 1; [Williamson 2009]), *shaolin* (Fig. 8; [Silva 2011b; Silva 2011a]), and *spaceship* (Fig. 9; [Grassard 2011]) meshes, we model two derivative meshes from the original one to demonstrate merging. These models span a variety of objects, from organic to man-made, and they were taken from different artists that likely have different styles of modeling.

Timing. As summarized in Tab. 1, the number of faces of the tested meshes vary widely from hundreds to over seventeen-thousand. *MeshGit* took from a few seconds to less than two minutes to compute the mesh edit distance between two meshes. This provides a practical solution for typical modeling workflows. We further expect that these timings to be significantly improved by a more optimized implementation of our code.

For the larger, more complex meshes, most of the time is spent iteratively refining the mesh edit distance. We found that the iterative greedy algorithm tended to iterate more in cases where the derivatives had strong sculpting edits along with changes to adjacencies. Such edits can be seen in Figs. 4 and 6.

7 Conclusion and Future Work

This paper presented *MeshGit*, an algorithm that supports 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 compute it. This distance is defined as the minimum cost of matching the vertices and faces of one mesh to those of another. 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 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 is a mostly trivial addition to *MeshGit* that requires changing our mesh elements to allow for arbitrary data to be attached; diffing and merging would follow the same extract algorithms. In the future, we 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. This would be helpful in creating arrays of secondary characters.

8 Acknoledgements

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

References

APACHE, 2012. Apache subversion. subversion.apache.org.

BLENDER FOUNDATION, 2011. Sintel model. 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. Revision control database for 3d assets. Tech. rep., University College London.

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.

GORALCZYK, A., 2008. Creature model. Creature Factory Blender Open Movie Workshop, vol. 2.

GRASSARD, F., 2011. Small spaceship (low poly). www.blendswap.com/blends/vehicles/small-spaceship-low-poly/.

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.

"LUMPYCOW", 2010. Chair model. www.blendswap.com/3D-models/furniture/lumpycow_household_brokenchair/.

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.

ROBLES-KELLY, A., AND HANCOCK, E. 2003. Edit distance from graph spectra. In *International Conference on Computer Vision*, 234–241.

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.

SILVA, E. D. R., 2011. Lost angel model. http://www.blendswap.com/blends/characters/lost-angel.

SILVA, E. D. R., 2011. Shaolin model. www.blendswap.com/3D-models/characters/shaolin.

TORVALDS, L., AND HAMANO, J., 2012. Git. git.scm.com.

VAZQUEZ, P., 2009. Durano model. Venom's Lab Blender Open Movie Workshop, vol. 4.

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

WILLIAMSON, J., 2009. Hand model. cgcookie.com/blender/2009/11/14/model-male-base-mesh/.

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.