Intelligent Scissoring for Interactive Segmentation of 3D Meshes

William Kiefer[1]
Princeton University

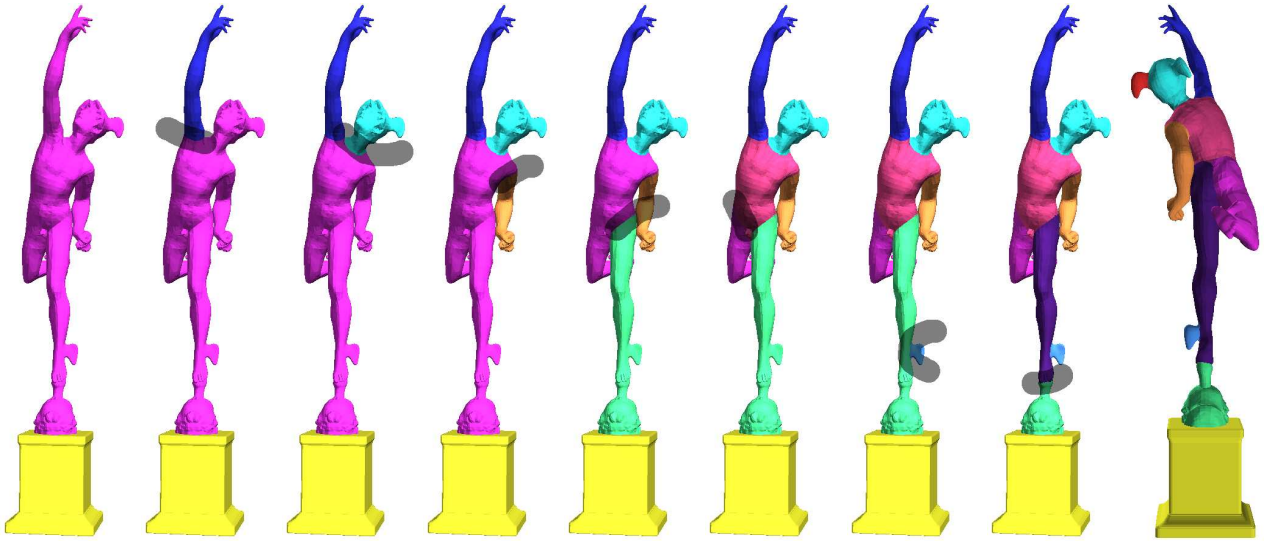


Figure 1: *Example session in which a user segmented a model with intelligent scissoring. Strokes are shown in transparent black. Different colored surfaces indicate separate parts computed by the intelligent scissoring algorithm. Note how the cuts between parts are smooth and well-placed, even though the user drew wide, inaccurate strokes. This sequence took under a minute with the system presented in this paper and the viewpoint never changed. The image on the far right shows the cuts as seen from on the backside of the mesh.*

Abstract

Many algorithms and tools exist for the segmentation of 3D meshes. However, they are labor intensive and lack the simplicity of 2D image segmentation systems, limiting them to a small set of expert users. This paper introduces a new segmentation tool which aims for accurate and flexible interactive mesh segmentation, yet maintains an easy-to-use interface suitable for novices and experienced users alike. Extending the stroke based interface of 2D intelligent scissoring, 3D intelligent scissoring tackles the difficulties that arise from three dimensionality while at the same time gives the user more freedom than in previous systems. Intelligent scissoring obtains desirable segmentations easily and interactively, and presents many possibilities for future work within the field.

Keywords: Segmentation, Interactive modeling tools

1 Introduction

The decomposition of an object into a set of meaningful parts is an important operation in many areas of computer graphics. In two dimensions, a variety of easy-to-use tools already exist for feature extraction and image segmentation. However, expanding these tools to three dimensional meshes introduces numerous new problems. Current research has developed a number of new tools for this task, yet because they can be labor intensive and difficult to use, their use is limited to a small set of experienced users. Thus, there is a demand in 3D interactive modeling for an easy-to-use tool that combines the simplicity of two dimensional image segmentation with the accuracy of many current three dimensional segmentation tools.

This paper introduces just such an intuitive and flexible tool for the interactive segmentation of 3D meshes. The user paints some number of strokes on the mesh surface to signify seams of desired segmentation. The system then finds the optimal closed contour through the user's strokes, and segments the mesh accordingly. This intelligent scissors system is a true extension to previous work on two dimensional image intelligent scissoring; the underlying algorithm is enhanced to adapt to the problems of three dimensionality and the simple stroke based interface is modified to allow for greater user flexibility. These two extensions, an optimization algorithm that allows for iterated refinement and a flexible interface for interactive segmentation, are the main contributions of this work.

Motivation for this work comes from the aforementioned gap within the current selection of interactive segmentation tools. Average users do not have user-friendly tools to create meshes of their own design. Additionally, the effort involved to use these tools increases as mesh resolution increases. As three dimensional models becomes more widespread for the everyday computer user, it is important to research and develop new easy-to-use tools that work for both the professional and the novice.

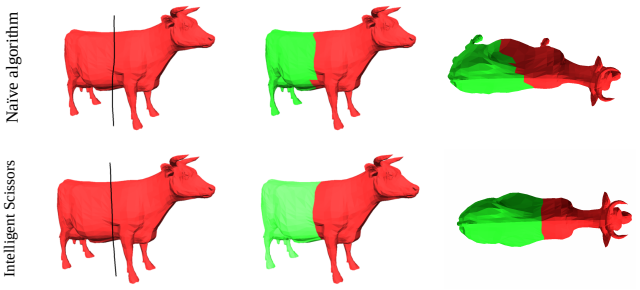
The following sections address the major issues in the development, use, and testing of 3D intelligent scissors. Section 2 explores current research and systems that relate to intelligent scissoring. Section 3 provides a broad overview of the system, while sections 4 through 6 examine the algorithms and implementation of intelligent scissors. Section 7 provides results from the system and section 8 concludes.

2 Related Work

Intelligent scissoring of 3D meshes builds off the research and work of numerous other systems. The following section describes what work has been done with relation to intelligent scissoring and how intelligent scissoring expands the existing field of interactive modeling.

2D Intelligent Scissoring: There has been extensive study in the

[1]Senior thesis, April 30th 2004. *Advised by* Thomas Funkhouser.



(a) User stroke (b) Front view (c) Top view

Figure 2: *A screen-space "lasso" (top row) produces an unexpected segmentation when the camera view is not perfectly aligned with the desired cut. In contrast, our intelligent scissors (bottom row) finds the optimal cut through the stroke, which may or may not be orthogonal to the view direction.*

field of 2D feature extraction. Systems like [Mortensen and Barrett 1995] have been incorporated into almost every commercial graphics editor today. Adobe Photoshop ([Incorporated 2004]) has both a "magnetic lasso" and an "extract" option to aid users in segmenting features within the image. Essentially, our system aims to transfer these intuitive and effective tools into the 3D modeling world. However, there are two main difficulties that arise during this progression into three dimensionality. First, there are no occlusions in 2D images. The entire image can always be seen from a single viewpoint. Thus, it is simple for a user to trace the contour of the feature they would like to extract. In three dimensions, the entire mesh cannot usually be seen from a single viewpoint, making it difficult and, for the most part, impossible to draw a complete contour in a single stroke from a single viewpoint. Second, the main assumption for 2D intelligent scissors, stemming from the lack of occlusions, is that the user will draw a fully closed contour. Both the "magnetic lasso" and "extract" option in Adobe Photoshop require such a contour. Because one of the goals for 3D intelligent scissors is simplicity, and since drawing a closed contour on a mesh in a single stroke is usually impossible, this assumption will not be made. In the end, our system will have to connect the various user strokes into a single closed contour as well as determine the best cut within the specified strokes.

Screen Space Algorithms: Almost every existing modeling system today (e.g. [Foundation 2004], [Wavefront 2004]) contains some variation on a simple screen space algorithm for segmenting parts of a mesh. The most common tool is a "select box," analogous to 2D image select boxes, where the user clicks and drags a point on the screen to define a rectangular region on the screen. All vertices and faces that project onto the screen within this region are then selected and segmented. This type of algorithm constrains the desired cut contour to align exactly with the view direction. As seen in Figure 2, poor results are obtained when this constraint is not met. In fact, it is often impossible to align the the best cut for a given mesh feature with the viewpoint direction. Also, surfaces in the mesh are cut even if they are not visible, so other parts of the model that are occluded by the selected region will also be segmented. In all instances, significant clean up by the user is required to create the desired segmentation. The time spent cleaning up (i.e. selecting individual faces for addition or removal from the segmented part) is proportional to the number of faces in the mesh, thus making finely sampled meshes labor intensive. Many modeling systems have extended the select box to a screen space "lasso." However, this lasso suffers from the same problems as mentioned above and additionally requires the user to draw a very precise contour on the screen.

Geometric Snakes: Geometric Snakes, introduced by [Lee and Lee 2002], is an energy minimization method for feature extraction on meshes. This method is effective in finding both protrusions and depressions in a mesh, and avoids the difficult clean up problems of screen space algorithms. However, in order to specify the region of the desired segmentation, individual vertices must be selected by the user. This forces the user, after specifying some vertices on one side of the mesh, to rotate the mesh and specify at least one vertex on the other side if a full encircling contour is desired. On large finely sampled meshes, it can be tedious to specify specific vertices and difficult to determine the optimal vertices to choose. Intelligent scissors aims to give the task of picking vertices to the system and allow the user to simply specify general areas of desired segmentation without necessarily changing the view direction.

Shortest Path Algorithms: In addition to geometric snakes, there are systems that find the shortest path between user specified vertices ([Gregory et al. 1999; Wong et al. 1998; Zöckler et al. 2000]). These systems fall subject to the same difficulties mentioned above and require the user to specify the vertices in consecutive order. Intelligent scissors builds off of this approach yet alleviates the user from the responsibility of specifying vertices and allows for the painting of strokes in arbitrary order and direction.

Min Cut Algorithms: As an extension to shortest path algorithms, a prototype intelligent scissoring system found the local graph around the shortest path using a breadth first search. Faces along the opposite edges of the connected graph were marked as sources and sinks, and a minimum cut algorithm was applied to find the optimal cut contour along the initial shortest path stroke contour. This method worked well for low resolution meshes. Yet since the running time to find a minimum cut is $O(n^2)$ where n is the number of vertices, this method was too slow to be interactive for large meshes. Additionally, it is difficult to form a minimum cut graph for an arbitrary number of user strokes in arbitrary locations, so more restrictions have to be placed upon the user. Local optimizations are also difficult since every refining stroke requires global minimum cut computation. This can cause the cut contour to change upon refinement in areas that were not local to the refining stroke.

Automatic Segmentation: Automatic segmentation methods ([Katz and Tal 2003]) produce desirable results for many meshes. There are obvious advantages to these automatic methods, yet oftentimes a user can immediately visualize a desired segmentation that could be difficult for automatic methods to determine. Also, in the context of an interactive modeling tool, the user would want to specify exactly which part should be segmented rather than automatically segment the mesh into many parts.

CAD Systems: CAD systems ([PTC 2004]) are not much different than the other commercial modeling systems mentioned above. The main difference lies in the type of meshes created, rather than in the system's tools. The meshes created in CAD systems tend to have strongly defined features with many sharp angles and flat planes. These situations are ideal for intelligent scissors because almost any sharp feature could be segmented by a single stroke. Thus, intelligent scissors could become an important tool in these systems.

Cut and Paste Methods: Cut and paste based modeling systems like [Biermann et al. 2002] allow for feature extraction from a source mesh before transferring that feature to some destination mesh. These systems focus mostly on the warping of the given feature to the destination mesh rather than the feature extraction itself. Typically, a general region around the desired feature must be specified by the user via a spline or set of faces. Yet this region need not be exact because the system will eventually blend the mesh feature with the destination mesh. Thus, the feature extraction tools of these systems are not exact enough for pure mesh segmentation and require the user to specify a full contour. Their user specified contour is generally not optimized by the system given that their

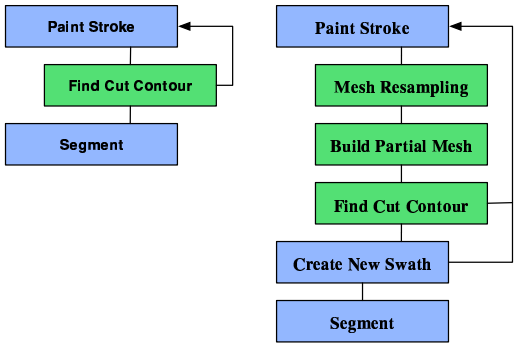


Figure 3: Left: *The basic flow of a user session.* ***Right:*** *An expanded view of a user session. Blue rectangles represent user controlled actions while green rectangles represent automated work on the part of the system (Creating new swaths can be done automatically by the system or by the user).*

algorithm blends the feature in the end.

This paper is an expansion of the intelligent scissoring described in [Funkhouser et al. 2004]. Certain figures and portions of text from this earlier work have been incorporated into this paper.

3 Overview

As seen in related work, current methods for interactive segmentation can be labor intensive and tedious. Many of the methods become more difficult as the resolution of the mesh increases. Thus, intelligent scissors aims to fill a current gap in interactive modeling tools and presents a simple and intuitive method to segment meshes into parts, regardless of the resolution of the initial mesh. Specifically, intelligent scissors frees the user from specifying specific vertices, allows the user to produce cuts that encircle the mesh from a single viewpoint, and gives the user the freedom to specify portions of the cut contour in any arbitrary order while the system gives progressive feedback.

The left image in Figure 3 shows the most basic flow of the intelligent scissors system. The user simply paints a stroke on the surface of the mesh indicating the desired area of segmentation. The system then finds the optimal closed cut contour defined by that region. Finally, the mesh is segmented along the cut contour. To allow for greater flexibility, the system can show the proposed cut contour to the user before segmentation and the user can paint more strokes to achieve further refinement.

The right image in Figure 3 shows a more detailed view of what actually happens in the intelligent scissors system. There are three main extensions to the basic system that the user may or may not use. (1) When the user finishes a stroke, the painted region of the mesh can optionally be resampled according to a desired edge length (section 6). This allows for better cut contours in coarse regions of the mesh where there are few initial edges and large triangles. (2) The system can optionally build a partial mesh, explained in section 5, which ensures optimization along the full length of self-intersecting user strokes. (3) In order to cut meshes with a genus other than zero, multiple cut contours may be needed. Swaths, explained in detail in section 4.4, are collections of user strokes that maintain separate cut contours. The system allows for creation of multiple swaths to handle high genus meshes.

These extensions do not complicate the basic flow of the intelligent scissors system. Instead, they extend the flexibility of the system as a whole while maintaining a basic interface. Mesh resampling and partial mesh creation are both handled automatically by the system. The creation of new swaths can also be handled automatically by the system, but may also be controlled by the user. In the common segmentation case, the user will simply draw a stroke and the mesh will be segmented immediately. Further refinement can be achieved through additional paint strokes on any of the existing swaths.

4 Intelligent Scissoring

Intelligent scissoring expands upon previous methods in four important ways. These are described in detail in the following four subsections.

4.1 Stroke Specification

Using a brush metaphor, the user paints “strokes" on the mesh surface to specify where cuts should be made (Figure 4a). Each stroke has a user-specified width, r, representing a region of uncertainty within which the computer should construct the cut to follow the natural seams of the mesh. From the user’s perspective, the meaning of each paint stroke is “I want to cut the surface along the best seam within here.” From the system’s perspective, it specifies a constraint that the cut contour *must* pass within r pixels of every point on the stroke, and it provides parameters for computing the cost of cutting along every edge, e, in the mesh:

$$cost(e) = c_{len}(e)^{\alpha} \times c_{ang}(e)^{\beta} \times c_{dist}(e)^{\delta} \times c_{vis}(e)^{\gamma} \times c_{dot}(e)^{\lambda} \times c_{curv}(e)^{\eta}$$

Edge costs are used to determine the optimal path along the mesh. Each edge cost parameter is described in depth below. The default values for the parameter weighting terms, α, β, δ, γ, λ and η are all one, but may be tweaked by the user for various cut behaviors.

Edge Length: $c_{len}(e)$ is simply the length of e, and ensures that short cut contours have low cost.

Dihedral Angle: $c_{ang}(e) = \theta_e/2\pi$ where θ_e is the angle between the two adjacent faces of e, giving cuts along concave edges less cost. This parameter dominates cut contour decisions on coarse meshes where single edges typically define important topological features such as creases. However, in finely sampled meshes, $c_{curv}(e)$ provides better optimization in troughs and valleys where the dihedral angle does not provide enough local information.

Visibility: $c_{vis}(e)$ gives less cost to edges that are not visible. This parameter is motivated by the observation that the user would have painted on a visible edge if a cut were desired there. In other words, not painting visible edges signifies that the user does not desire a cut in that region. Combined with $c_{dot}(e)$, $c_{vis}(e)$ encourages the least cost cut contour to traverse the “back-side” of the mesh. Without these terms, the least cost path would most likely traverse through the user stroke and then back upon itself. In general, when the user stroke is more than half of the width of the object being cut, the least cost path will traverse the “back-side” of the mesh

Normal Orientation: $c_{dot}(e)$ as mentioned above encourages the least cost cut contour to traverse the back side of the mesh. Whereas $c_{vis}(e)$ gives all non-visible edges less cost, $c_{dot}(e)$ gives edges whose adjacent face normals are aligned with the viewing direction less cost. Without this parameter, the least cost cut contour would have the tendency to follow the non-visible edges along the silhouette boundary back to the beginning of the stroke. Thus, $c_{dot}(e)$ allows the least cost contour to traverse the actual “back-side” of the mesh, rather than traverse the edges just beyond the silhouette boundary.

Stroke Distance: $c_{dist}(e) = \frac{r-d}{r}$ where d is the maximum distance from the centerline of the stroke to the screen space projection of

the edge. This parameter reflects the desire of the user to cut edges that lie close to the center of the stroke.

Curvature: Considering curvature when calculating the edge cost allows for better cut decisions in finely sampled troughs and creases of a mesh where the dihedral angle does not provide enough information about local topology. $c_{curv}(e)$ represents the curvature of the mesh perpendicular to the edge direction. When a mesh is loaded, curvatures and principal direction vectors are computed for every vertex according to [Rusinkiewicz 2004]. These values are interpolated to compute curvatures for every edge. Depending on the mesh topology, these edge curvatures can vary greatly over a wide range of real numbers. In order to map this variance to a meaningful range of [0, 1], statistical techniques are applied. Specifically, after the initial mean and standard deviation are calculated, curvature values that lie more than three standard deviations above or below the mean are temporarily ignored while a second mean and standard deviation are calculated. This second mean and the standard deviation define the floor and ceiling that map curvature values to the range [0, 1]. The default range, which can be adjusted by the user, clamps curvature values to within three standard deviations above and below the second calculated mean.

The $c_{ang}(e)$ and $c_{curv}(e)$ parameters are by default set up to encourage cuts along edges with negative perpendicular curvatures (protrusions and creases in the mesh). However, by inverting these terms it is possible to segment along very convex portions of the mesh (depressions and rounded corners).

4.2 Finding the Cut Contour

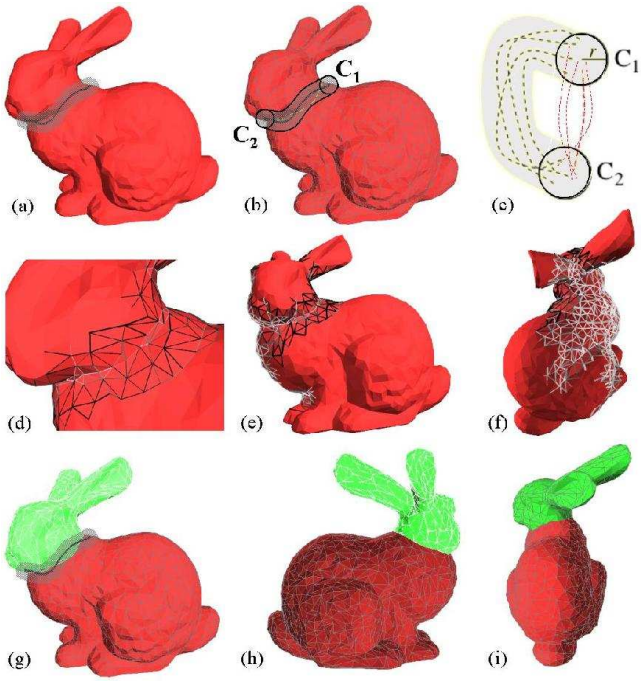


Figure 4: *Cutting the bunny with intelligent scissoring: (a) the user draws a wide paint stroke; (b) the system identifies all vertices in the* caps *of the stroke, C_1 and C_2; (c) it then finds the least cost paths from every vertex in C_1 to every vertex in C_2 twice, once constrained to lie within the stroke (yellow dotted lines) and once without any constraints (red dotted lines), and forms the proposed cut out of the the pair of paths with the least total cost. (d-f) Since the edges traversed by the algorithm (wireframe gray) have less cost (lighter gray values) in concave seams and on the back-side of the mesh, (g-f) the least cost cut partitions the mesh along a natural seam of the mesh.*

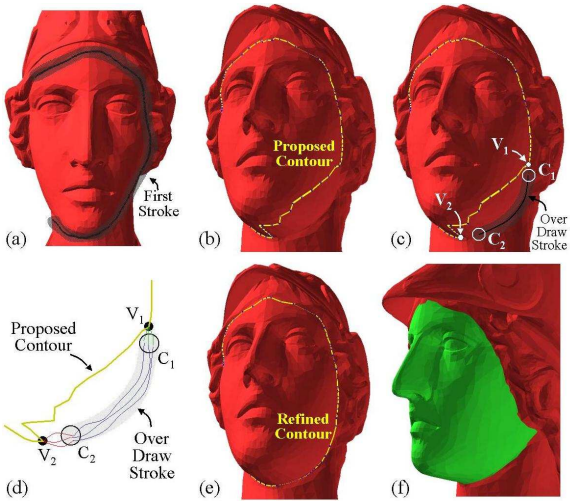


Figure 5: *Cutting the face of Athena with intelligent scissoring: (a) the user draws an imprecise first stroke (gray); (b) the system proposes a cut (yellow curve); (c) an overdraw stroke (gray) is drawn to refine the cut; (d) the system splices in the least cost path traveling from V_1 first to C_1 (red), then to C_2 within the stroke (blue), and finally to V_2 (green); (e) the proposed cut contour is updated; (f) the final result is a segmentation of the mesh into two parts (green and red) separated by natural seams of the mesh.*

As mentioned in the previous section, the challenge is to find the least cost closed sequence of edges that passes within r pixels of every point on the user's stroke in sequence. Because the user can draw an open contour, two distinct least cost sub-problems must be solved: the optimal path within the stroke and the optimal path connecting the end points of the stroke back together. These two paths together create a closed cut contour. The key to solving these two sub-problems efficiently is observing that the cut must pass through at least one vertex in the "cap" at each end of the stroke. The caps of the stroke, C_1 and C_2 (Figure 4b), are defined as the sets of vertices within screen space radius r of the first and last points on the stroke drawn over the model itself. Using Dijkstra's shortest path algorithm ([Dijkstra 1959]), modified with our edge cost function described above, the system solves the two sub-problems: (1) find the least cost path constrained within the boundaries of the user's stroke between all vertices in C1 and all vertices in C2, and (2) find the least cost path, unconstrained by the stroke boundaries, between all vertices in C1 and all vertices in C2 (Figure 4c). The optimal cut contour is the pair of paths, one from each sub-problem, that forms a closed contour with least total cost.

The computational complexity of the intelligent scissoring algorithm for a single stroke is $O(k\,n\log n)$, where n is the number of edges in the mesh, and k is the number of vertices in $min\{C_1, C_2\}$. k is typically small and the constrained least cost path problem only considers a small subset of the mesh. Therefore, the upper bound on computation time is determined by the unconstrained least cost path search. In general, these least cost path searches only cover some subset of the mesh, allowing running times to be interactive in practice.

4.3 Refining the Cut Contour

By default, the system will partition the mesh immediately after the first stroke, according to the computed optimal cut. However, the user is provided the option of refining the cut interactively with "over-draw" strokes. In this case, the system displays a "proposed cut" for user verification. If unsatisfied, the user can draw new strokes that refine the cut incrementally. This feature encourages the user to draw broad strokes quickly, in any order, and then iteratively refine the details only where necessary.

For each over-draw stroke, S, the system automatically determines the portion of the proposed cut that should be replaced by the over-draw stroke and splices in a new, locally optimal path through the new stroke. The system starts by finding the points, V_1 and V_2, on the proposed contour closest to the stroke's endpoints on the mesh (Figure 5c). If they lie within a previously painted region, the system simply removes the shorter of the two cut contours between them. Otherwise, V_1 and V_2 are moved away from each other along the proposed cut until they both reside in previously painted regions.

To compute the new path from V_1 to V_2, a divide and conquer approach is used again. We first compute the least cost paths from V_1 and V_2 to all vertices in their corresponding stroke caps, C_1 and C_2. Then, we compute the least cost paths within the stroke from all vertices in C_1 to all vertices C_2 as before (Figure 5d). Finally, we find the triplet with least total cost forming a connecting path from V_1 to V_2 through C_1 and C_2 and splice it into the proposed cut. This algorithm also runs in $O(k\,n\log n)$.

This incremental refinement approach has several desirable properties. First, it provides local control, guaranteeing that previously drawn strokes will not be overridden by new strokes unless they are in close proximity. Second, it is fast to compute, since all but two of the least cost path searches are constrained to lie within the stroke. Finally, this method allows the user to specify precisely where the splice should be made by simply starting and stopping the over-draw stroke with the cursor near the proposed contour.

4.4 Handling High Genus

Meshes with a genus above zero cannot necessarily be segmented into parts using a single closed cut contour. To handle these cases, we define a new term, a *swath*, to refer to a collection of one or more user strokes that together create a single closed cut contour. Each swath on the mesh can be refined independently of the others. The mesh always has one active swath, which is the target of any new user refinement, and zero or more inactive swaths. At any point, the user can create a new swath or switch which swath is currently active.

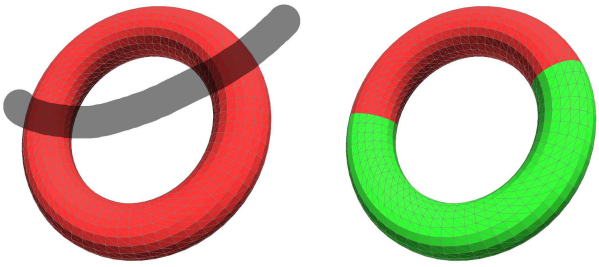


Figure 6: Left: *A single user stroke that crosses two sections of the torus, causing the system to automatically generate two separate swaths.* ***Right:*** *Resulting segmentation.*

Additionally, new swaths are automatically created when the user stroke crosses multiple portions of the mesh on a single stroke, i.e. crosses a silhouette boundary onto the background and then crosses a silhouette boundary back onto the mesh (Figure 6, left). Without this feature, the user would be required to draw one stroke, create a new swath, draw the second stroke and finally segment. Automatic swath generation allows the user to simplify this procedure into a single stroke.

5 Partial Mesh Creation

The following subsections provide the motivation, definition, and implementation of the partial mesh extension to the intelligent scissoring system.

5.1 Motivation and Definition

Given the algorithm described in section 4, the least cost cut contour is still not guaranteed to satisfy the original problem description in certain cases. Specifically, when the user stroke is self-intersecting, the least cost cut contour will not necessarily pass within r pixels of every point on the user stroke. Because there are no negative edge weights, the least cost cut contour will always bypass any cycles within the user stroke, thereby neglecting to pass within r pixels of all points on that cycle.

An initial solution might be to restrict users from drawing cycles within their strokes. This might be feasible when dealing with extraneous cycles that do not make intuitive sense when segmenting meshes. However, one common case is for the user to slightly overlap the start and end of the stroke (Figure 7, left), in essence making the whole stroke path a cycle. In these cases the system would propose the cut in the middle image of Figure 7, whereas the correct least cost contour would look like the right image of Figure 7.

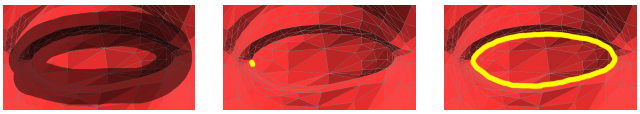


Figure 7: Left: *The user stroke, black, shown around the eye of Athena. Notice that the start and end of the stroke overlap each other.* ***Center:*** *The proposed contour without the use of a partial mesh. The least cost contour between cap vertices is a single edge.* ***Right:*** *The proposed cut contour using partial mesh construction. The least cost path satisfies the requirement that it passes within r pixels of every point on the original user stroke. Proposed contours in this figure are shown thicker for clarity.*

In order to overcome this case of self-intersecting user strokes we define a *partial mesh*. A partial mesh is an exact copy of the original mesh under the user stroke with the added property that it does not self-intersect (Figure 8). The least cost paths between cap vertices constrained within the stroke are then calculated on the partial mesh, while the unconstrained least cost paths are still computed on the original mesh. Optimizing the cut contour over the partial mesh ensures that the least cost cut contour will traverse any cycles within the user stroke, thus avoiding the situation in Figure 7, middle.

The creation of partial meshes is presented as an option to the user and may be turned off. Although the creation of partial meshes does not add significantly to the running time (it runs in $O(m)$ where m is the number of vertices within the stroke), turning it off saves computation in cases where the user does not plan to draw self-intersecting strokes.

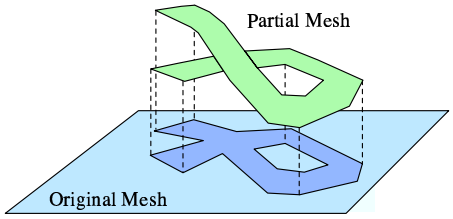


Figure 8: *Visualization of a partial mesh. Notice that the partial mesh is a duplicate of the original mesh except that it does not intersect itself.*

5.2 Implementation

The implementation of partial meshes is straightforward. After the user stroke is drawn, a non-optimized open contour is found along the center of the stroke. Every edge along this contour is created within the initially empty partial mesh. Doing so "unfolds" the stroke: intersecting edges along the stroke are no longer adjacent on the partial mesh. New edges, faces, and vertices are added via a breadth first search whose terminating case is a vertex that lies outside the stroke radius r. Each edge, face, and vertex on the partial mesh stores a reference to the original mesh, allowing easy translation back to the original mesh after the least cost path is found.

6 Mesh Resampling

The following subsections describe the motivation and implementation of mesh resampling with respect to intelligent scissors.

6.1 Motivation

Cut contours described in the previous sections were all limited to preexisting vertices and edges. This limitation of cut contours, although satisfactory in many cases, cannot handle many possible segmentations. Planar regions of a coarsely or finely sampled mesh may only contain a small number of triangles whose edges allow for few segmentation options. For example, if a user wished to cut a tabletop comprised of two large triangles, the only possible cut would be the single edge across the diagonal. With mesh resampling, the tabletop could be cut into any arbitrary shape (Figure 10).

Segmenting through planar regions with low tessellation is one practical motivation for mesh resampling within the intelligent scissoring system. However, many other scenarios can be ameliorated with this extension: a user may have specific and fine segmentation requirements, the initial mesh could be poorly tessellated (i.e. many long thin triangles), or the segmented region may need to be edited after the segmentation in a way that requires additional samples. Mainly, mesh resampling allows for more segmentation flexibility regardless of the initial mesh.

6.2 Implementation

Mesh resampling is achieved through the iteration of three basic edge operations as in [Hoppe et al. 1993]. These three edge operations collapse, split, or swap a single edge as seen in the diagram in Figure 9. To determine which edges need to be collapsed, split, and swapped, we use a method similar to that of [Markosian et al. 1999] and [Lawrence and Funkhouser 2003] where each edge keeps track of its "desired length" as well as its actual length. After the user paints a stroke, the "desired length" of every visible edge underneath the stroke is updated to the length specified by the user before the stroke. The ratio of desired length to actual length for each edge determines which operations, if any, are to be performed for the edge.

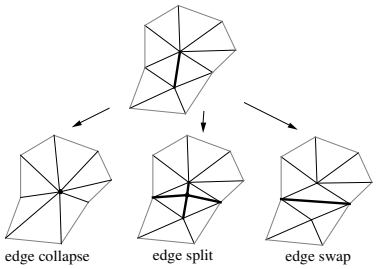


Figure 9: *The three basic edge operations used during mesh resampling.*

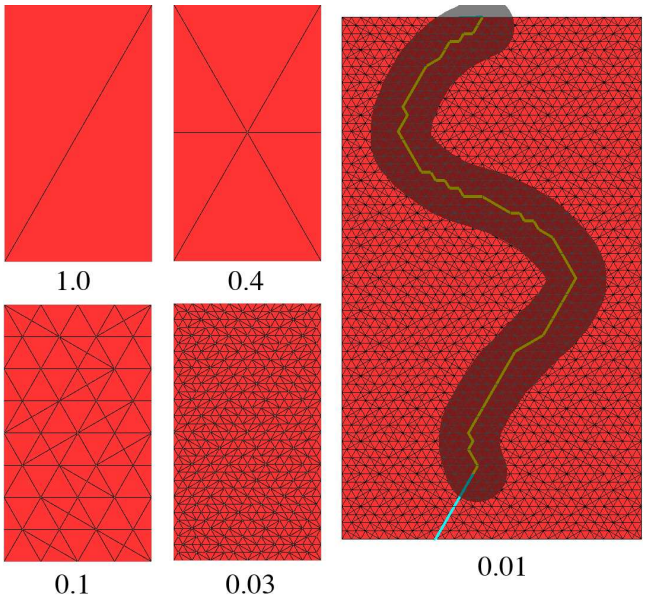


Figure 10: *An overhead view of a table top that has been resampled incrementally using the indicated values as desired edge lengths. The zoomed image on the right shows a user stroke along with the proposed cut contour. This arbitrary cut was impossible on the original tabletop, shown top left, that only consisted of two triangles.*

Specifically, three heaps maintain each of three values for every edge: the minimum dot product of all angles on the adjacent faces for edge swaps, the ratio of desired length to actual length for edge splits, and the ratio of desired length to actual length for edge collapses. Although both are sorted using the same value, the edge collapse heap is distinct from the edge split heap because they are sorted in opposite orders. The system iterates through each heap in order (split, swap, collapse), performing that heap's respective edge operation on the minimum value edge until no more legal operations can be performed in that heap. Legal edge operations are defined as follows:

Edge Split: Edge splits are legal when the ratio of desired length to current length falls below a given threshold. The default threshold for the system is 0.5. Thus, when the desired length of an edge is half the current length of the edge, that edge is split, and the two resulting edges will better approximate the desired length of an edge. Because splitting an edge does not introduce any error into the mesh topology, there are no additional restrictions on edge splits.

Edge Swap: Edge swaps are legal if two conditions are met: (1) the current minimum dot product of all the angles of the edge's adjacent faces falls below a threshold and (2) if the swap will increase the minimum dot product of all the angles of the resulting adjacent faces. Condition one ensures that only adjacent triangles with one large angle are considered, and condition two ensures that the swap will be productive. Large angles are characteristic in long thin triangles and thus, attempting swap edges in these situations promotes the creation of roughly uniform equilateral triangles.

Edge Collapse: Edge collapses are legal when the ratio of desired length to current length is larger than a given threshold. The default for the system is 1.5. Using logic similar to that of legal edge splits, edges are collapsed when the current length is less then half of the desired length. Collapsing these short edges aids in the removal of skinny triangles. Edge collapses have the ability to introduce resampling error because they can significantly change local topology. Currently, the system does not impose any additional restrains

on the legality of edge collapses because most of the edges that fall above the threshold are small enough that the error produced is not significant. However, more sophisticated methods such as error quadrics ([Garland and Heckbert 1997]) or energy minimization ([Hoppe et al. 1993]) could be applied to diminish the introduction of error.

One proposed method for mesh resampling involves creating vertices along the center of the user stroke, and then triangulating the surrounding area appropriately. However, this method suffers from the major setback that it does not sample uniformly. Irregularly shaped triangles, especially long thin ones, have the severe disadvantage that any future segmentation in the area produces jagged, irregular cuts. Additionally, within the context of interactive modeling, uniform sampling improves results for many other mesh operations ([Hoppe et al. 1993]). Therefore, the implemented method of mesh resampling described above is used primarily because it ensures that affected triangles are roughly equilateral and that vertices are evenly sampled. A uniform resampling of the mesh allows greater flexibility for future segmentation and other modeling operations. Furthermore, laying down vertices along the user stroke tends to lock the cut contour into the exact path of the user. Rather than create a path of edges along the user stroke and adjust the surrounding topology, intelligent scissors adopts the opposite view: resample the local topology and then optimize under the stroke accordingly.

7 Results

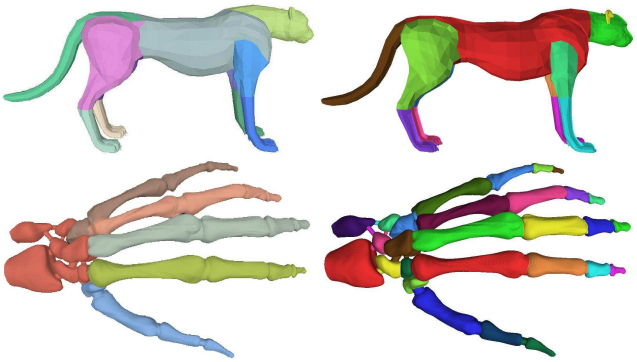
The following subsections present results obtained from the intelligent scissors system described above. All tests were performed on a 2.8 GHz Pentium IV processor running Windows XP with 1 GB of memory and a GeForce4 graphics card.

7.1 Scissoring

Figure 11 compares our intelligent scissoring results for a cheetah and a hand with results reported by [Katz and Tal 2003]. While this comparison is not altogether fair, since our algorithm is interactive and theirs is automatic, it highlights the main disadvantage of automatic approaches: they do not understand semantics. For instance, the cheetah segmentation produced by [Katz and Tal 2003] includes portions of the animal's back with the tail and neck and contains an unnatural boundary between the right-hind leg and body (Figure 11, top left). As a result, the parts cannot be simply pasted into another model without re-cutting them. Similarly, their hand segmentation does not separate all the bones. Our segmentation of the hand (Figure 11, bottom right) took 13 minutes (of user time), while the automatic segmentation (Figure 11, bottom left) took 28 minutes (of computer time) for the same model [Katz and Tal 2003]. Our segmentation of the cheetah took under thirty seconds.

Figure 1 and Figure 17 show two example user sessions where a complicated model can be segmented into many parts from a single viewpoint using single stokes. Both sessions took under a minute of user time and highlight the simplicity of intelligent scissoring. Figure 16 shows an additional session more typical of a CAD system. This series shows that on a mesh with sharp features, intelligent scissoring can provide a quick accurate segmentation with a single stroke.

Figure 15, on the other hand, shows a more involved user session on a mesh where the object being segmented could not be cut with a single stroke. Instead, the user draws three separate strokes (Figure 15b, d, f) to achieve the desired segmentation of the dragon limb (Figure 15h), using the proposed cut contours (Figure 15c, e, g) as iterative feedback. This series, completed in under three minutes, demonstrates the advantage of refinement strokes on a complicated mesh.



[Katz and Tal 2003] Intelligent scissors

Figure 11: *Comparison of segmentations produced by the automatic algorithm of [Katz et al, 2003] (left) and the interactive intelligent scissors algorithm described in this paper (right) for a model of a cheetah and a hand (654,666 polygons). Note that the cuts are better aligned with the semantic seams of the object with intelligent scissors.*

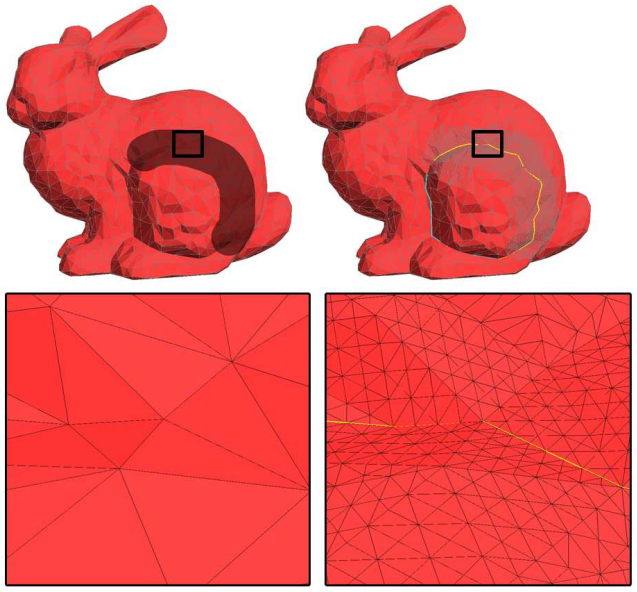


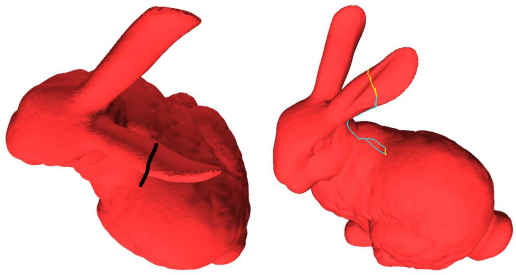
Figure 12: Left: *The top image shows a user stroke over the bunny. The portion of the mesh indicated is shown in detail below.* ***Right:*** *The proposed cut contour after resampling. Again, the indicated region of the mesh is shown in detail below. Notice the even resampling of the area under the stroke.*

Mesh resampling (Figure 12) successfully resamples the mesh with even triangulation under the stroke. Although in this example the resampling does not provide a path of less cost, it does allow for more detailed editing of the segment. Because only areas under the stroke are reampled, there is a tendency for uneven triangulation along the border of the resampled area and the rest of the original mesh. Other limitations to mesh resampling are discussed in the section 7.2.

7.2 Limitations

The intelligent scissoring algorithm cannot perform all types of splits (e.g. Figure 14). For instance, if a surface is occluded from all viewpoints, the user cannot paint on it. Other user interface metaphors, such as a "laser" mode in which all surfaces under the mouse get cut, would be better for such a situation (e.g., [Owada et al. 2003]). Similarly, the painting interface metaphor can produce

unexpected results when the user paints over a silhouette boundary. The problem is that the system makes sure to cut through every part of the user's stroke, which may connect points adjacent in screen space but distant on the surface (Figure 13).



(a) User Stroke (b) Proposed Cut

Figure 13: *The intelligent scissors algoritm ensures that the cut contour (blue and yellow line on right) visits all regions painted by the user (left), which may be problematic when the stroke crosses an interior silhouette boundary. Yellow portions of the cut are painted, and blue ones are not.*

If the user brush width, r, is large enough, desired cuts across thin parts of the mesh don't necessarily traverse the backside of the mesh. Since r is large, the least cost path will, in these situations, be the folded contour between the two closest cap vertices within the stroke. Take, for instance, the case where r is greater than half of the width of the object being segmented. The C_1 and C_2 vertices will overlap and the resulting contour will be a single vertex or single edge. To combat this situation, a heuristic was used to limit the actual set of cap vertices within each cap to those whose projection onto the estimated stroke vector fell below a certain threshold. This heuristic effectively removed cap vertices that were located close to the center of the stroke and close to the other end cap. While this solved the problem and forced cuts with large caps to traverse the backside of the mesh, it sometimes eliminated the vertex that would have produced the least cost path. Further work could be done to investigate the minimum set of cap vertices to use that would still produce the optimal least cost cut contour and avoid the above large r situation.

The mesh resampling algorithm presented in section 6 is a prototype and is limited in two ways: (1) the algorithm does not attempt to minimize error and (2) the algorithm only resamples the mesh under the user stroke and not along the rest of the cut contour. Fortunately, as mentioned previously, the first limitation is minimal because the mesh resampling algorithm used does not produce significant error most of the time. Only occasionally does the local topology change noticeably. Yet resampling itself was not the focus of this paper and other more sophisticated resampling methods could easily substitute for the current method.

The second limitation, on the other hand, is somewhat significant. While resampling under the user stroke is effective and produces the desired results (Figure 14c), the areas of the mesh between strokes are not resampled (Figure 14d). Thus, if the user attempts to segment a fairly simple object that consists of a few large triangles, new resampled edges are created along the user stroke on the front side but edges on the backside of the mesh remain coarse. Thus, the proposed cut contour will be desirable on the front yet jagged on the back side where the number of edges is low. Figure 14 demonstrates this situation. Further work must be done to determine where else the mesh should be resampled after such a user stroke. A screen space solution to this question fails in many of the same cases that a screen space segmentation algorithm fails and resampling the entire mesh is unnecessary and computationally expensive for most situations. Another possible solution would be to find the initial jagged contour, resample around it, and then find a new least cost cut contour. Nevertheless, the resampling of the mesh brings to light this new limitation and spawns a new area of research.

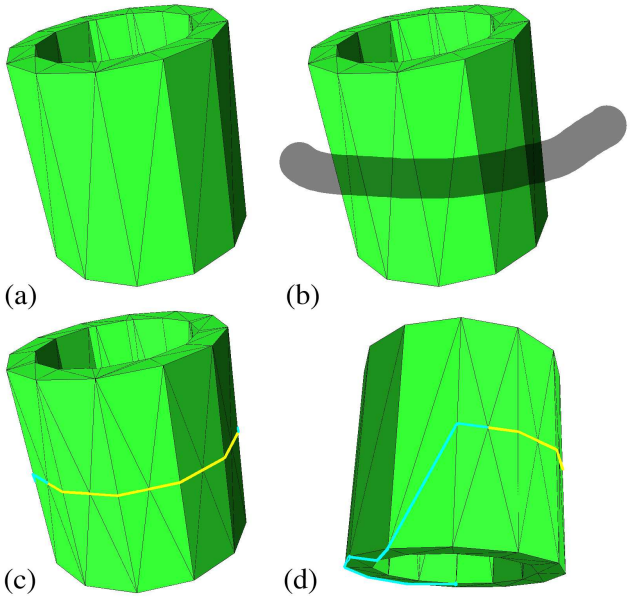


Figure 14: *A user stroke with mesh resampling turned on. The initial mesh (a) has no horizontal edges along the surface of the tube. The user stroke (b) results in the creation of new edges in the desired area and what looks like a desirable cut (c). However, the backside of the model has not been resampled and the cut contour is forced to go along the bottom of the tube (d). Resampling the entire mesh for this situation would be a probable solution, yet would be inefficient for many other situations. Interestingly, this mesh also demonstrates a limiting case because the only possible segmentation requires painting on the inside of the tube.*

8 Conclusions and Future Work

This paper proposes an intuitive and easy-to-use tool for interactive segmentation of 3D meshes. This intelligent scissoring system is accurate and maintains its simplicity despite the resolution of the target mesh. Although this segmentation system can be adopted into a variety of disciplines, it is aimed primarily for incorporation within interactive modeling tools. An earlier version of 3D intelligent scissors was implemented for Modeling by Example ([Funkhouser et al. 2004]), a system designed to create models from parts of existing models within a database.

In addition to the future work discussed in section 7.2, there are various avenues of future research for intelligent scissors. Since intelligent scissors combines both a user interface and an algorithm in a modular way, various other algorithms can be incorporated underneath the interface. For instance, Geometric Snakes ([Lee and Lee 2002]) could be applied to the proposed cut contour for further refinement. Work could also be conducted to develop variations to the stroke based interface that would allow different stroke specifications for different situations.

Overall, this paper presents a new tool in an unexplored area of interactive modeling, focusing on achieving the simplicity of two dimensional image segmentation within the three dimensional world. As a new tool in an array of existing modeling operations, intelligent scissors will hopefully open the door to new easy-to-use modeling tools that will help make 3D meshes as common and accessible as 2D clip art is today.

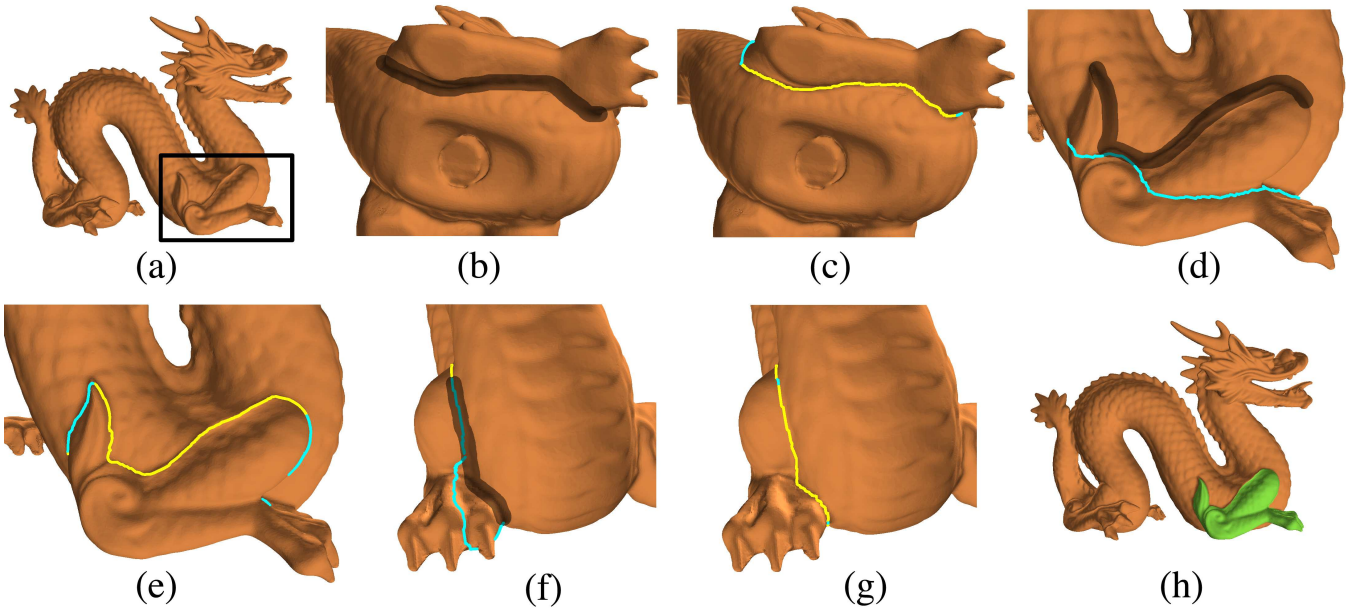


Figure 15: *Example session in which the user segments the front right limb of a dragon mesh (a) consisting of 1,132,830 faces. This session demonstrates intelligent scissors for a segmentation that requires more than one stroke. The user draws an initial stroke (b) on the underside of the mesh. The system proposes a cut (c), however the user rotates the model and sees that the rest of the contour is not desirable. (d). An additional stroke is drawn (d) and the system refines the contour (e). The figure is rotated again, and the user draws a third refining stroke (f). The proposed contour (g) is what the user wants, and the mesh is segmented. The resulting segmentation is shown in green (h).*

Figure 16: *A user stroke and resulting segmentation of the fandisk model (12,946 faces). Segmentation is shown from various viewpoints.*

Figure 17: *A sequence of user strokes on the armadillo model (345,944 faces). From left to right, the user drawn series of approximate strokes with the resulting segmentation.*

References

BIERMANN, H., MARTIN, I., BERNARDINI, F., AND ZORIN, D. 2002. Cut-and-paste editing of multiresolution surfaces. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, ACM Press, ACM, 312–321.

DIJKSTRA, E. W. 1959. A note on two problems in connection with graphs. *Numerical Mathematics 1*, 269–271.

FOUNDATION, B., 2004. Blender. http://www.blender3d.com.

FUNKHOUSER, T., KAZHDAN, M., SHILANE, P., MIN, P., KIEFER, W., TAL, A., RUSINKIEWICZ, S., AND DOBKIN, D. 2004. Modeling by example. *ACM Transactions on Graphics (SIGGRAPH 2004)* (Aug.).

GARLAND, M., AND HECKBERT, P. S. 1997. Surface simplification using quadric error metrics. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., ACM, 209–216.

GREGORY, A., STATE, A., LIN, M., MANOCHA, D., AND LIVINGSTON, M. 1999. Interactive surface decomposition for polyhedral morphing. *Visual Comp 15*, 453–470.

HOPPE, H., DEROSE, T., DUCHAMP, T., MCDONALD, J., AND STUETZLE, W. 1993. Mesh optimization. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, ACM, 19–26.

IGARASHI, T., MATSUOKA, S., AND TANAKA, H. 1999. Teddy: a sketching interface for 3D freeform design. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., ACM, 409–416.

INCORPORATED, A. S., 2004. Adobe photoshop CS. http://www.adobe.com.

KATZ, S., AND TAL, A. 2003. Hierarchical mesh decomposition using fuzzy clustering and cuts. *ACM Trans. Graph. 22*, 3, 954–961.

LAWRENCE, J., AND FUNKHOUSER, T. 2003. A painting interface for interactive surface deformations. *Pacific Graphics* (Oct.).

LEE, Y., AND LEE, S. 2002. Geometric snakes for triangluar meshes. *Computer Graphics Forum (Eurographics 2002) 21*, 3, 229–238.

MARKOSIAN, L., COHEN, J. M., CRULLI, T., AND HUGHES, J. 1999. Skin: a constructive approach to modeling free-form shapes. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., ACM, 393–400.

MORTENSEN, E. N., AND BARRETT, W. A. 1995. Intelligent scissors for image composition. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, ACM Press, ACM, 191–198.

OWADA, S., NIELSEN, F., NAKAZAWA, K., AND IGARASHI, T. 2003. A sketching interface for modeling the internal structures of 3d shapes. In *3rd International Symposium on Smart Graphics*, Lecture Notes in Computer Science, Springer, 49–57.

PTC, 2004. Pro engineer. http://www.ptc.com.

RUSINKIEWICZ, S. 2004. Estimating curvatures and their derivatives on triangle meshes.

WAVEFRONT, A., 2004. Maya. http://www.aliaswavefront.com.

WONG, K. C.-H., SIU, Y.-H. S., HENG, P.-A., AND SUN, H. 1998. Interactive volume cutting. In *Graphics Interface*.

ZELEZNIK, R. C., HERNDON, K. P., AND HUGHES, J. F. 1996. Sketch: an interface for sketching 3D scenes. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, ACM Press, ACM, 163–170.

ZÖCKLER, M., STALLING, D., AND HEGE, H. 2000. Fast and intuitive generation of geometric shape transitions. *Visual Comp 16*, 5, 241–253.