PERGAMON

Computers & Graphics 25 (2001) 635–642

COMPUTERS & GRAPHICS

www.elsevier.com/locate/cag

Technical Section

Interactive cutting of 3D surface meshes

Cynthia D. Bruyns[a,*], Steven Senger[b]

[a] Center for Bioinformatics, NASA Ames Research Center, Mail Stop 239-11, Moffett Field, CA 94035-0168, USA
[b] University of Wisconsin - La Crosse, Department of Computer Science, 1725 State St., La Crosse, WI 54601, USA

Abstract

Engineers and scientists from many fields are using three-dimensional reconstruction for visualization and analysis of physical and abstract data. Beyond observing the recreated objects in artificial space, it is desirable to develop methods that allow one to interactively manipulate and alter the geometry in an intuitive and efficient manner. In the case of medical data, a baseline interactive task would be to simulate cutting, removal and realignment of tissue. In order to provide for operational realism, the underlying mathematical calculations and topological changes need to be invisible to the user so that the user feels as though they are performing the task just as they would on the real world counterpart of the reconstructed object. We have developed a method that allows the user to directly sketch the desired cut contour on a three-dimensional surface in a manner that simulates dragging scissors through fabric. This new method can be employed interactively, allowing the user to perform the task in an intuitive, natural manner. © 2001 Elsevier Science Ltd. All rights reserved.

Keywords: Graphics data structures and data types; Interaction techniques; Medicine and science

1. Introduction

A number of techniques have been developed to simulate cutting surface and volumetric meshes. Our method focuses on creating a system that can use various forms of user interfaces and can be interactive on very large scientific and medical datasets. Our tool can be used for modeling (i.e. creating new surfaces) or simulation (i.e. simulating cutting operations) and has been tested on models up to 900,000 polygons in size using both 2D (mouse) and 3D (tracked instrument) interfaces. The new cutting tool system, described here, is built on the OpenInventor framework and works in conjunction with many commonly used viewing and editing tools.

One way to classify different cutting approaches is by the interface used to translate user input into operations on the underlying mesh of an object. Common interface devices such as a 2D mouse or tablet, for example require mapping the user's 2D movement in user space into 3D coordinates in the virtual space. 3D input devices, such as a tracked instruments (i.e. glove, stylus, or other instruments with a sensor attached) allow the user greater freedom but with the added expense of performing intersection tests over some representation of the mesh. Devices which also convey the reaction forces of a given cutting operation, such as a haptic feedback devices (i.e. with a stylus tip, scissors or endoscopic handle attachment) reduce the degrees of freedom allowed to the user but require computation of the reaction forces at kilohertz speeds. Additionally, cutting tools can also be classified according to the dynamic state of the mesh. Systems that simulate dynamic deformations of a mesh either before or as a result of cutting operations have additional computational requirements. Cutting tools can also be categorized by the nature and type of intermediate steps required to simulate an otherwise continuous cutting procedure. These steps, such as picking surface points, or placing a series of planes, often limit the level of realism afforded by these methods, but have historically been necessitated by the inability to interactively update

*Corresponding author. Tel.: +1-650-604-0375; fax: +1-650-604-3954.
E-mail address: cbruyns@mail.arc.nasa.gov (C.D. Bruyns).

the underlying topological changes on large surface meshes.

In one implementation, the user defines the outline of the cut path by picking points on the surface of the object. The vertices closest to the pick location are treated as seed points and are connected by a shortest path algorithm [1]. The shortest path between these points determines the connected set of vertices that define the cut contour. The complete contour defines the boundary of a mesh subregion, which can be separated from the original mesh by making a copy of each of the vertices and updating vertex connectivity of the polygons along the boundary.

A more natural means of cutting allows for tracing the contour directly on the 3D surface of the object. This requires mapping the user's position to a corresponding position on the 3D object. Mapping the user's position onto an object is commonly performed either by casting a ray into a scene or by using z-buffering techniques and performing intersection tests with the object. In order to trace a contour many of these tests are necessary in order to fit a curve to the sampled points. Even then, there is no guarantee that the sampled points land on topologically significant locations, that is, points that can be used in retriangulation, such as an intersection with an edge or vertex.

A more useful procedure would create new vertices at the intersections of the path with existing polygons. These kinds of cutting algorithms are implemented by placing a single plane (fixed or adaptive in size) or a series of planes linked by successive surface markers [2–5]. When using a single plane, positioning it into the desired configuration may require repeated attempts by the user in order to obtain the desired position and orientation. When linking planes, the number of markers used to define the cut contour controls the resulting curvature of the cut path. These methods are usually semi-interactive, necessitating some lag in time between placement of the plane and mesh retriangulation.

Cutting tools that incorporate interactive intersection tests and mesh retriangulation typically use 3D interface devices and collision detection algorithms. One approach has been to simply remove the intersected elements creating a tear or hole within the mesh [6]. Another method is to duplicate the existing vertices closest to the intersection and update vertex connectivity of the polygons along the boundary in order to produce a separation along the cut path [7]. One could also create new vertices at the intersections and retriangulate the intersected polygons based on cutting cases or templates [8–10].

Our method takes a similar approach, retriangulating the mesh based on cut cases. However, we bind an object to the surface of the mesh to represent the cutting instrument and allow the user to directly manipulate this object, tracking its location and performing interactive polygon retriangulation as the object moves along an arbitrary cut path. Moreover, since the tool creates and responds to changes in the underlying mesh data structures as it moves along the surface of an object, it can be used in conjunction with deformable meshes as well. Such deformations might occur as a result of modeling the force with which the user is moving the cutting object through a mesh, or dynamic reaction of the mesh after a cutting operation.

2. Methods

We have chosen to implement the cutting tool by incorporating its functionality in OpenInventor framework as a subclass of SoDragger, which is the base class of objects that respond to the user click/drag/release event paradigm [11]. The physical cutting tool is an object that is attached to the surface of a mesh which is then moved by the user to create cuts in real-time. Interactive cutting is made possible by locally updating the associated mesh data structures. The result of these cut operations is a modified mesh that represents the new topology. The entire cut procedure occurs in four phases, placement, activation, cutting and deactivation:

2.1. Placement

Placement of the *cutter* occurs when the user visually inspects the model and decides where the cut should begin. When using a 2D input device, a single mouse click at this location triggers a ray pick callback that determines the face and location the user has chosen. When using a 3D input device, the user's position is mapped onto the object by projection onto the closest face of the mesh. The chosen face is recorded as the *currentFace* and the location is recorded as the *startPt*. An instance of the *cutter* is then created and its geometry is deposited into the scene at the *startPt* and oriented in the direction of the *currentFace's* normal. We have chosen to use a red sphere as the dragger geometry to indicate the advancing point of the cut.

2.2. Activation

This method is similar to the SoDragger::dragStart() routine. Activating the *cutter* triggers a callback that defines a local *tangentPlane* defined by the *startPt* and *currentFace's* normal. Using a 2D device, activation can be signaled by selecting the *cutter* object. Using a 3D device, toggling a switch can signal the user's intention to move the *cutter*.

2.3. Cutting

The cutting callback, which is recursively called to advance the *cutter* to the user's current position, can be further divided into five phases: movement; tracking; testing; retriangulation; and update. Fig. 1 outlines the cutting loop and the data obtained at each phase:

Movement: While active, user motion triggers a callback similar to the SoDragger::drag( ) routine that projects the user position (M) onto the local *tangentPlane* on the surface of the object at $M_{Projected}$. The *cutter*'s position is then set to $M_{Projected}$ allowing the *cutter* to move along the plane of the *currentFace*. Given the updated *cutter* position, we track *dragVec* the vector from $M_{Projected}$ to the last edge intersection point (*entryPt*). If there has not yet been an intersection, we use the original selected location (*startPt*).

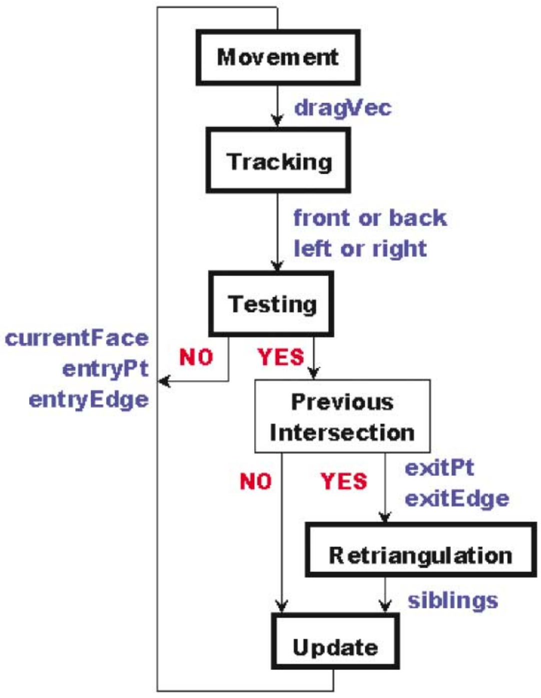

Fig. 1. Flow diagram of cutting callback and associated data at each step.

Tracking: In an effort to reduce the time needed to test the edges of the next face for intersection, we can immediately access and test the next edge that would be crossed. By projecting *dragVec* onto the last edge crossed (*entryEdge*), we can predict if the *cutter* will be dragged to the edge sharing the left or right vertex of *entryEdge*. We can also project *dragVec* onto the vector formed by crossing the *entryEdge* with the *currentFace's* normal to determine if the *cutter* is being moved forwards or backwards through the *currentFace*. Knowing whether the *cutter* is moving to the right or left, forwards or backwards direction allows us to immediately obtain the next edge that should be tested for intersection.

Testing: A test is performed to see if the *cutter* has been dragged beyond the edge boundaries of the *currentFace*. Dragging beyond these boundaries would mean that there is an intersection between an edge and the *dragVec*. The location of this intersection on the edge is called the *exitPt* and the edge that the *cutter* crossed through is called the *exitEdge*.

Fig. 2 illustrates the *cutter* movement and intersection test performed during the cutting callback. The *dragVec* is shown in purple as the vector from the projected mouse coordinates, $M_{Projected}$ (drawn in red), to the *startPt* (drawn in green). The *exitEdge* is shown in blue, and the intersection of the *dragVec* with the *exitEdge* is shown in yellow as the *exitPt*.

The movement, tracking and testing phases are executed for each iteration of the cutting callback; retriangulation and update phases are performed only if the testing phase determines that there has been an intersection.

Retriangulation: A vertex pair, *exitVertLeft* and *exitVertRight,* is created at the *exitPt* dividing the intersected *exitEdge* into left and right segments. Using these new points, the *currentFace* is retriangulated using a previous vertex pair, *entryVertLeft* and *entryVertRight*, these points having been created when the *cutter*

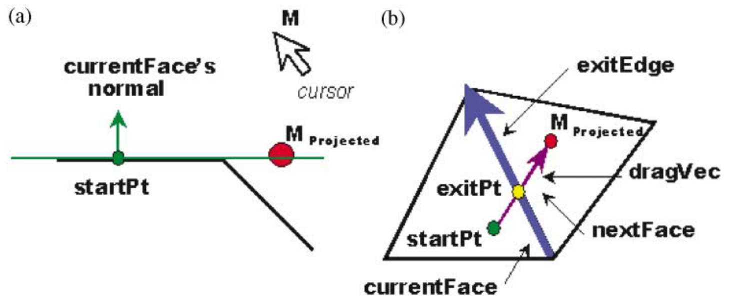

Fig. 2. (a) Cursor position after projection (Side view). (b) Intersection of the path with the edge of the polygon (top view).

entered the *currentFace*. The possible combinations of entry and exit edges determine the cut cases.

In addition to updating the mesh adjacency data structures, we carry a hash table of vertex sibling groups. Since a newly formed edge can also be split, a vertex quadruplet, which corresponds to four copies of the same vertex, is recorded in a hash table according to the vertex index of each copy. Fig. 3 details the creation of vertex siblings (drawn as circles). Dragging the *cutter* across the face creates two new vertices on entry (drawn in blue) and exit (drawn in yellow). Since only two copies are needed, sibling 3 and 4 are empty for now. However, when trying to drag the *cutter* across an edge that was created by a previous cut through the face (green lines), four copies of the same vertex are needed (drawn in red). The usefulness of this table will be discussed in the next section.

Update: When the *cutter* has moved past an edge of the *currentFace*, the *nextFace* that shares the *exitEdge* is found. Once the *nextFace* is found, the *cutter* is then positioned at the *exitPt*, oriented in the direction of the *nextFace's* normal and the local *tangentPlane* is updated.

If no face is found that shares the *exitEdge*, the vertices that form the *exitEdge* are tested to see if they have siblings and if these siblings form an edge in the mesh. If they do, this indicates that the *exitEdge* was formed by a previous cut. Thus, using the sibling vertices, we can find the *nextFace* when crossing over a gap in the mesh.

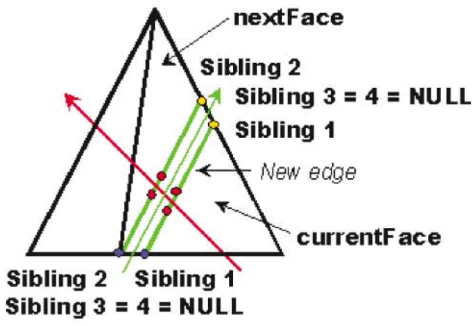

Fig. 3. Creation of sibling information. Green arrow shows first pass, siblings created in blue and orange. Red arrow shows second pass, siblings created in red.

There are, however, pathological cases where no *nextFace* will be found. For example, a mesh with a hole adjacent to the *exitEdge* would cause the *cutter* to stop at the edge and require the user to place another *cutter* at the other side of the hole and continue onward.

Because the connectivity information is updated with each user motion, the user is free to follow an arbitrary path. Backing up, cutting through the same triangle, and creating a path that crosses itself are all permitted. Fig. 4 demonstrates these special cut cases. The red arrow indicates a first pass through the face; the edges formed by retriangulating the face are also drawn in red. The green arrow indicates the second pass through the new faces; the edges formed by retriangulating these faces also drawn in green.

These five steps are recursively called allowing the *cutter* to move through as many adjacent triangles as needed in order to advance to the user's motion. This ensures a smooth cut path over the continuous surface of the object regardless of the size or shape of the underlying polygons or the sampling rate of the input device.

2.4. Deactivation

Once dragging has stopped, a method similar to SoDragger::dragFinish() is called which deactivates the *cutter*. At this step, the algorithm determines whether to close off the path and form a closed contour or to wait for future user motion, This decision is based on the proximity of the current *cutter* position to its original placement on the object. If dragging stops at a location that is not sufficiently close to the original placement, the *cutter* waits at the last projected user position in the last *currentFace*. At this time, the user is free to manipulate the object to expose hidden surfaces. Cutting

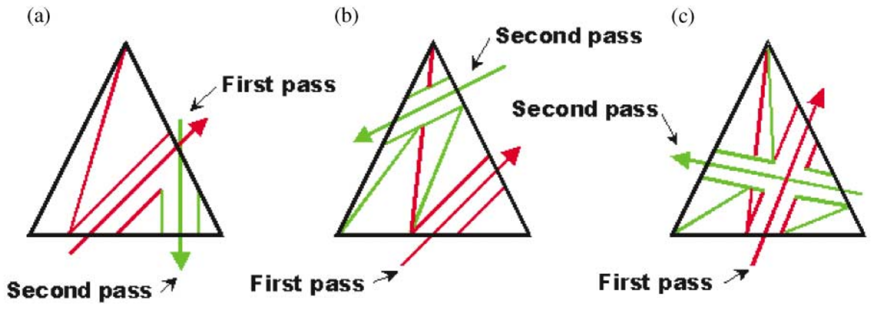

Fig. 4. (a) Backing up through a polygon, (b) cutting the same polygon, (c) self-crossing path.

can then resume on the newly exposed region by activating the *cutter* and moving it across the surface.

We have also implemented a modified version of this *cutter* that can cut two layers simultaneously. This dragger, *cut2Layers* consists of a primary *cutter* controlled by the user and a secondary *cutter* that operates by tracking the position of the primary *cutter*. The user deposits the primary *cutter* as before determining the *topCurrentFace* and *topStartPt*. A ray originating at this location is

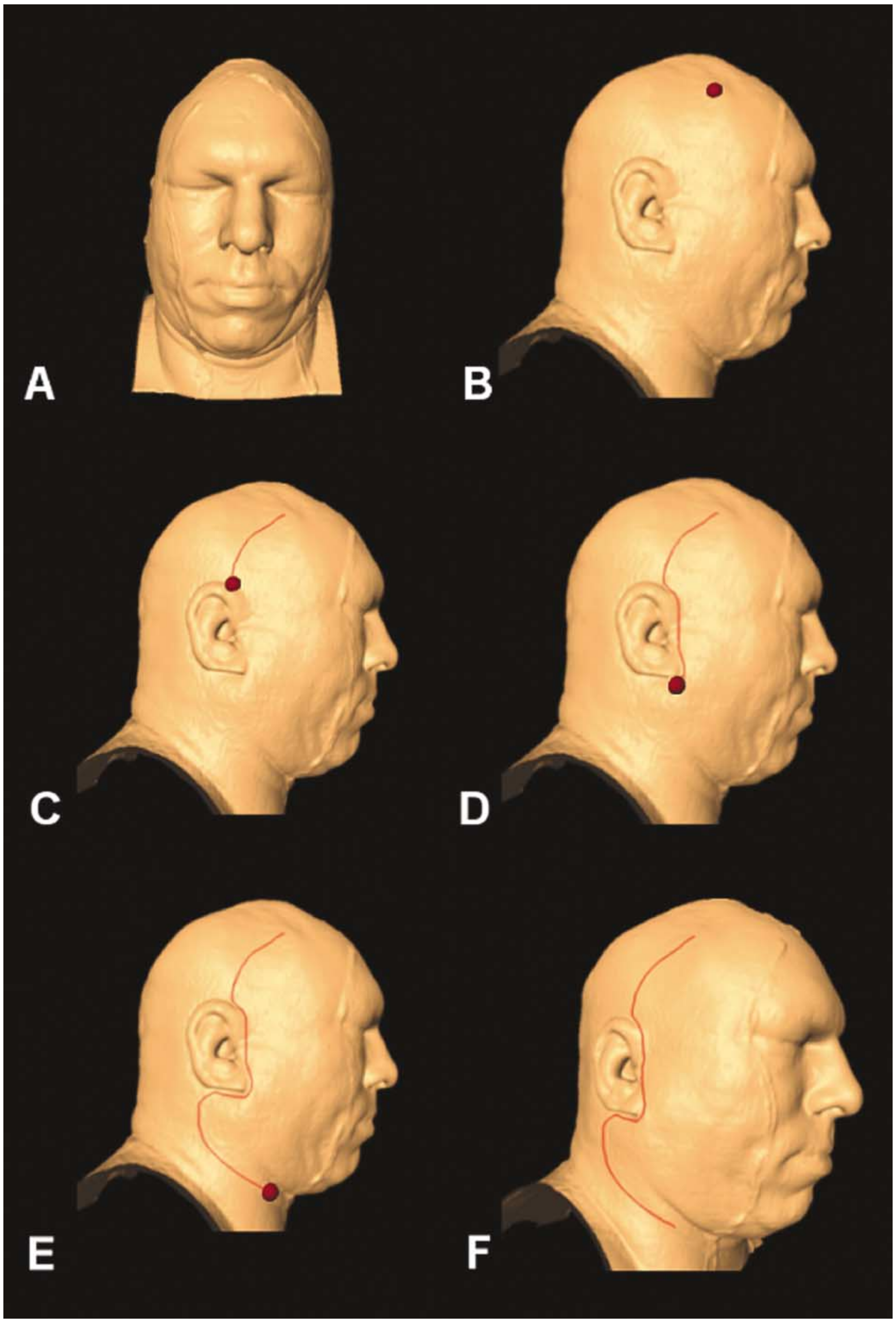

Fig. 5. Cutting a single surface, (A) the original mesh, (B) placement of the cutter, (C–E) dragging the cutter, retriangulating as it moves, (F) the resulting cut.

then cast in the opposite direction to find the *bottomCurrentFace* and *bottomStartP*, where the secondary cutter is then deposited, but does not respond to direct user interaction. The rest of the callbacks happen as before, with actions applied to the top surface first.

3. Application

The environment hardware consists of an Immersa-Desk DI (Fakespace Systems, Kitchener, Ontario), a Polhemus Fasktrak (Polhemus Inc., Colchester, Vt.) with Stylus, a SGI workstation and stereo glasses. This

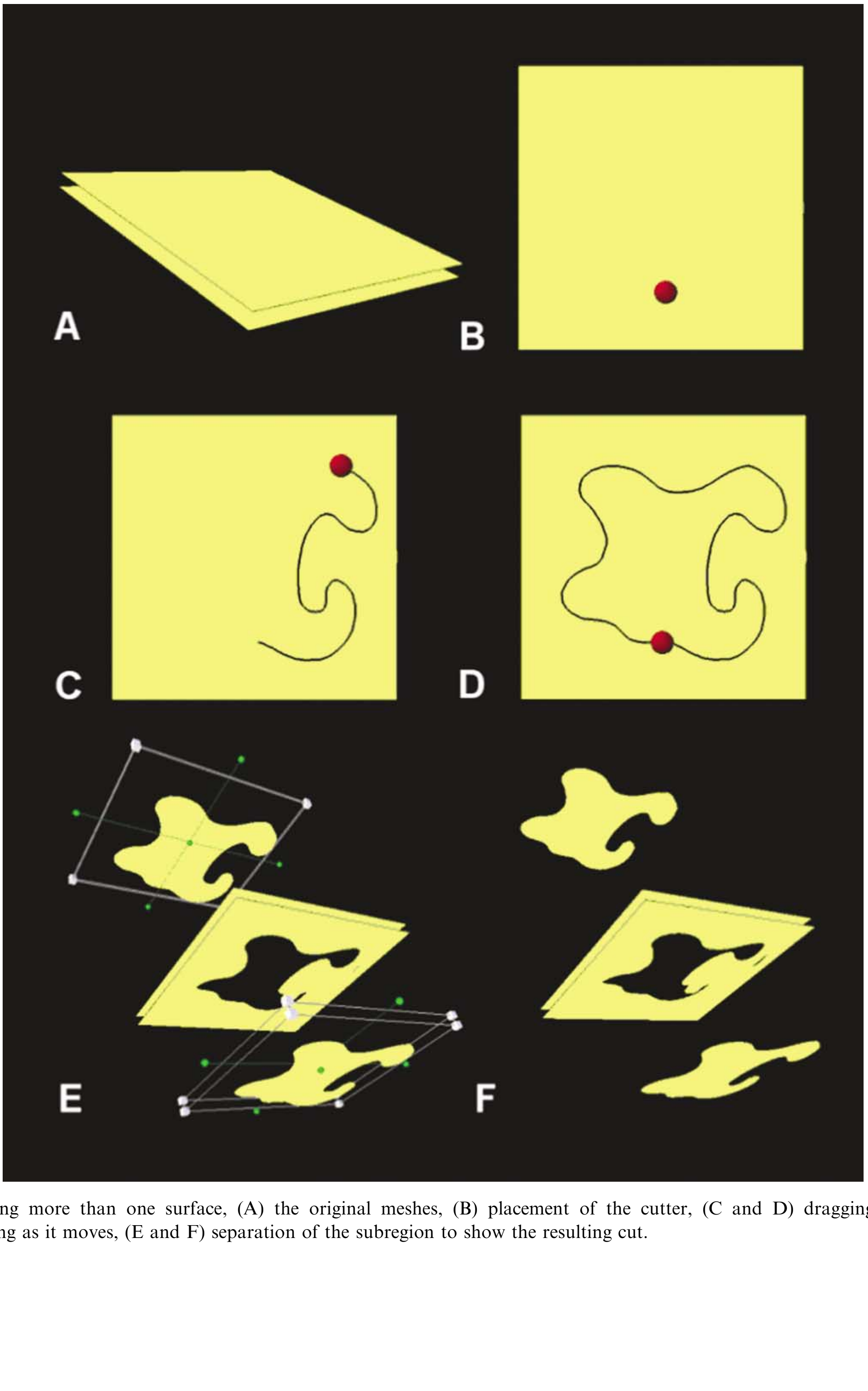

Fig. 6. Cutting more than one surface, (A) the original meshes, (B) placement of the cutter, (C and D) dragging the cutter, retriangulating as it moves, (E and F) separation of the subregion to show the resulting cut.

tool has been used to interactively cut meshes up to 900,000 polygons on an Onyx2 utilizing one RI2000, 2 GB RAM 300 MHz processor.

This tool can be used to make cuts that simulate complex biomedical incisions. In Fig. 5 we show the cutting tool applied to a 100,000-polygon mesh derived from the male data of the Visible Human Project produced by the National Library of Medicine. The series of frames demonstrates the process of creating a skin flap. Frame A shows the original mesh, while Frame B shows how the *cutter* is placed onto the surface at the desired location. The next three frames show how the user moves the *cutter* to follow the desired path. Frame F shows the resulting cut when the *cutter* has been removed from the surface.

In some cases it is desirable to cut more than one surface at once. For example, when approximating volumetric data by extruding the surface of an object along the vertex normal, thus producing a series of surfaces connected by springs [12]. Another instance occurs when trying to cut a mesh constructed from the bone-soft tissue boundary of a CT scan of the skull. In this situation the mesh contains regions of non-intersecting surfaces for the exterior and interior surfaces of the skull. The two-layer version of the *cutter* could be used to cut these surfaces (Fig. 6).

4. Discussion

The cutting tool described in this paper provides a natural interface for creating cuts through very large meshes. The figures above demonstrate the freedom with which the user can create arbitrary cuts. The key to performing this operation in real time is the use of data types that provide quick access to information. Each object carries look-up tables of vertex to vertex and vertex to face records for adjacency tests that are updated when new vertices and faces are formed. Because the *cutter* is moving from one face to an adjacent neighbor, the performance of the algorithm is independent of the size of the mesh.

The size of polygons in the mesh does impose some limits on the curvature of the cut path. This is because polygons are retriangulated after intersections with two edges, so subtle motions within the region bounded by the edges are not captured. However, since we are tracking the movement of the *cutter* at each iteration, we can use a change in direction in the interior of a polygon, as a signal to tessellate the polygon in order to create a more detailed cut path.

Another limitation of the cutting tool arises when the intersection of the cut path lands exactly on a vertex. One solution is to step off the vertex by a small amount, thus forcing the *cutter* to move through an edge. A side effect of this procedure is that narrow triangles are formed within the mesh, requiring extra storage without adding any detail to the model. As a post-processing step, various techniques for removing narrow triangles, such as merging faces, adding Steiner vertices and re-triangulating can be performed when the *cutter* is deactivated.

5. Conclusions

We presented a method that allows the user to intuitively perform cutting of complex geometries. The method includes anchoring the cutting object onto a 3D surface, eliminating the need for intermediate steps to define the cut path. By retriangulating the mesh as the *cutter* is moved, the user is free to sweep along an arbitrary path, creating complex path descriptions. The method is also general enough to allow the user to cut several non-intersecting surfaces at once. This implementation has been built on the OpenInventor framework, and can be inserted into commonly used visualization and editing environments. This method can be extended to other classes of input devices, such as a haptic devices or tracked real instruments. As stated earlier, because the cutting object stays attached to the surface of the object no matter what modifications occur in the mesh, this tool can be used to move through deformable objects as well.

Acknowledgements

This work was supported through grants to Richard Boyle, Jeff Smith and Muriel Ross at the Center for Bioinformatics at the NASA Ames Research Center.

References

[1] Wong KC, Sin TY, Heng P. Interactive volume cutting. http://www.dgp.toronto.edu/gi/gi98/papers/136/136.html.

[2] Keeve E, Girod S, Girod B. Computer-aided craniofacial surgery. Computer Aided Surgery 1996;3(2):6–10.

[3] Pieper S, Rosen J, Zeltzer D. Interactive graphics for plastic surgery: a task-level analysis and implementation. Proceedings of the Symposium on Interactive 3D Graphics '92, Cambridge, MA, 29 March–1 April. New York: ACM Press, 1992. p. 127–34.

[4] Neumann P. Near real-time cutting. Conference Abstracts and Applications: 27th Annual Conference on Computer Graphics and Interactive Techniques, New Orleans, LA, 23–28 July. New York: ACM Press, 2000. p. 239.

[5] Schutuser F, Van Cleyenbreugel J, Nadjmi N, Schoenaers J, Suetens P. 3D image-based planning for unilateral mandibular distraction. In: Heinz U, editor. Proceedings Computer Assisted Radiology and Surgery,

San Francisco, CA, 28 June–1 July. New York: Elsevier, 2000. p. 899–904.

[6] Frisken-Gibson SF. Using linked volumes to model object collisions, deformation, cutting, carving, and joining, IEEE Transaction on Visual Computing and Graphics 1999;5(4):333–48.

[7] Basdogan C, Ho C, Srinivasan MA. Simulation of tissue cutting and bleeding for laparoscopic surgery using auxiliary surfaces. In: Westwood J, editor. Medicine Meets Virtual Reality: the Convergence of Physical & Informational Technologies; Options for a New Era in Healthcare, San Francisco, CA, 20–23 January. Oxford: IOS Press, 1999. p. 38–44.

[8] Bielser D, Volker A, Gross M. Interactive cuts through 3-dimensional soft tissue. Computer Graphics Forum 1999;18(3):C31–8.

[9] Bro-Nielsen M, Helfrick D, Glass B, Zeng X, Connacher H. VR simulation of abdominal trauma surgery. In: Westwood J, editor. Medicine Meets Virtual Reality: the Convergence of Physical & Informational Technologies; Options for a New Era in Healthcare, San Diego, CA, 29–31 January 1998. Washington, DC: IOS Press, 1999. p. 117–23.

[10] Voss G, Hahn JK, Muller W, Lineman RW. In: Westwood J, editor. Medicine Meets Virtual Reality: the Convergence of Physical & Informational Technologies; Options for a New Era in Healthcare, San Francisco, CA, 20–23 January. Oxford: IOS Press, 1999. p. 381–3.

[11] Wernecke J. The Inventor mentor: programming object-oriented 3D graphics with Open Inventor. Rel 2. Reading, MA: Addison-Wesley, 1994.

[12] Waters K. A physical model of facial tissue and muscle articulation derived from computer tomography data. Proceedings Visualization in Biomedical Computing, Chapel Hill, NC, 13–16 October. Bellingham, WA: SPIE—The International Society for Optical Engineering, 1992. p. 574–83.

Cynthia Bruyns is currently a research scientist at the NASA Ames Research Center. Her research interests include computational geometry and the computer graphics. She received her MS in Mechanical Engineering from Stanford University in 1997 and her BS in Mechanical Engineering from Rensselaer Polytechnic Institute in 1995. She is a member of the ACM.

Steven Senger is currently a professor at the University of Wisconsin, LaCrosse, where he has been in the departments of computer science and mathematics since 1983. His research interests include mathematical logic and scientific visualization. He received his BS and Ph.D. degrees in mathematics from Purdue University, West Lafayette, Indiana. He is a member of the ACM, IEEE Computer Society, and the Association for Symbolic Logic.