ARTIST-DRIVEN FRACTURING OF POLYHEDRAL SURFACE MESHES

A Thesis

presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Tyler Casella

December 2013

COMMITTEE MEMBERSHIP

TITLE:	Artist-Driven Fracturing of Polyhedral Surface Meshes
AUTHOR:	Tyler Casella
DATE SUBMITTED:	December 2013
COMMITTEE CHAIR:	Zoë Wood, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Chris Lupo, Ph.D. Associate Professor of Computer Science
COMMITTEE MEMBER:	Phillip Nico, Ph.D. Professor of Computer Science

ABSTRACT

Artist-Driven Fracturing of Polyhedral Surface Meshes

Tyler Casella

This paper presents a robust and artist driven method for fracturing a surface polyhedral mesh via fracture maps. A fracture map is an undirected simple graph with nodes representing positions in UV-space and fracture lines along the surface of a mesh. Fracture maps allow artists to concisely and rapidly define, edit, and apply fracture patterns onto the surface of their mesh.

The method projects a fracture map onto a polyhedral surface and splits its triangles accordingly. The polyhedral mesh is then segmented based on fracture lines to produce a set of independent surfaces called fracture components, containing the visible surface of each fractured mesh fragment. Subsequently, we utilize a Voronoi-based approximation of the input polyhedral mesh's medial axis to derive a hidden surface for each fragment. The result is a new watertight polyhedral mesh representing the full fracture component.

Results are aquired after a delay sufficiently brief for interactive design. As the size of the input mesh increases, the computation time has shown to grow linearly. A large mesh of 41,000 triangles requires approximately 3.4 seconds to perform a complete fracture of a complex pattern. For a wide variety of practices, the resulting fractures allows users to provide realistic feedback upon the application of extraneous forces.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Computer graphics pertains to the process of deriving a visual representation of data via specialized data structures and algorithms. One of most common data structure utilized to represent 3-dimensional objects is a polygonal mesh. A polygonal mesh comprises of a list of vertices, edges, and faces defining a collection of triangles representing a polyhedral object [25], shown in Figure 1.1.

Through various rendering techniques, these polygonal meshes are transformed from raw data representing a 3D scene, into a 2D image depicting the scene as perceived through a camera. The ultimate goal of these techniques is to produce an image, or series of images, that accurately mimic the natural world. In order to do so, both the appearance and behavior must exhibit authenticity. As the techniques and capabilities of computer graphics have evolved, so too has a viewer's expectations for a fully immersive experience. This thesis aims to satisfy these expectations by helping to yield the immersive element of destruction.

The fracturing of objects within our environment upon an application of overbearing forces is a key characteristic of reality. Yet due to the complex and time-consuming nature of specifying and fabricating fracture behavior for polygonal meshes, it is an often omitted feature. As a solution to this handicap, this thesis proposes a method of

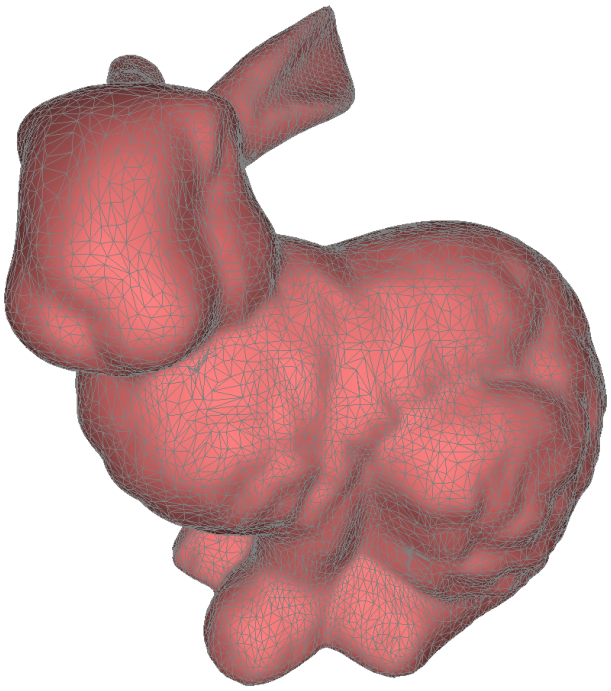


Figure 1.1: A polyhedral surface mesh of the Stanford bunny. The mesh consists of a set of vertices, edges, and polygonal faces.

defining hierarchical fracture behavior via generation and utilization of specialized 2D textures called `fracture maps`. An example of a fracture map is shown in Figure 1.2.

The technique presented in this paper is an algorithm that works in practice given a number of limitations are met. The algorithm is refered to as the `Fracture Map Algorithm`.

Traditionally, fracture techniques require a complex volumetric representation of an object and generate models via physically based modeling. This produces realistic results, but leaves very little flexibility for customization. Alternatively, the method introduced in this paper provides full artist control, while maintaining the ability to generate realistic fractures. Furthermore, by diverting a majority of computation work as a preprocessing step, the method is a suitable solution for both realtime and non-realtime applications.

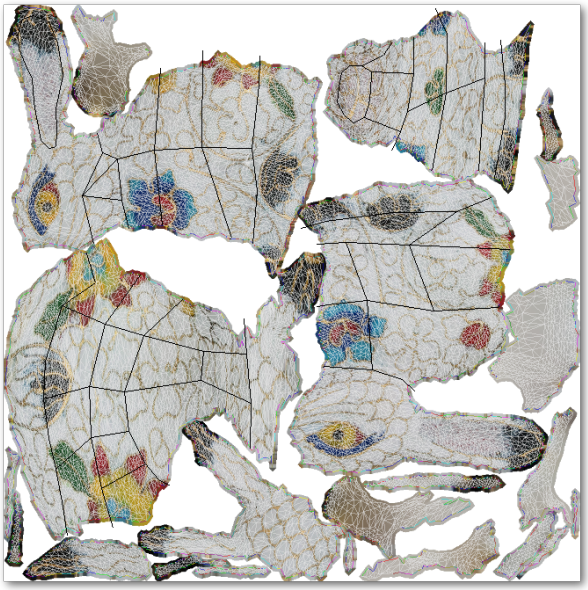


Figure 1.2: An example fracture map for the Standford bunny mesh. The black lines indicate edges between nodes within the fracture map and has been superimposed on top of the model's diffuse map.

The approach used for defining fractures within a polygonal mesh is designed to place the power and free creative expression in the hands of artists. This is achieved via the use of `fracture maps`. Fracture maps are simple 2D textures defining the lines of fracturing via black lines. When mapped onto the mesh, triangles intersecting fracture lines are split accordingly. This process results in a set of `fracture components`, shown in Figure 1.3, consisting of adjoining triangles within a fracture line loop.

A significant challenge faced by the algorithm is to derive the hidden surface of each fracture component. Unlike other mesh representations, polyhedral surface meshes contain explicitly only information regarding the surface of an object. To account for the hidden surface introduced when fracturing the mesh, the Fracture Map Algorithm derives an approximate medial axis of the mesh to be used as the hidden surface. The result is a set of new watertight meshes representing each individual piece of an input fracture.

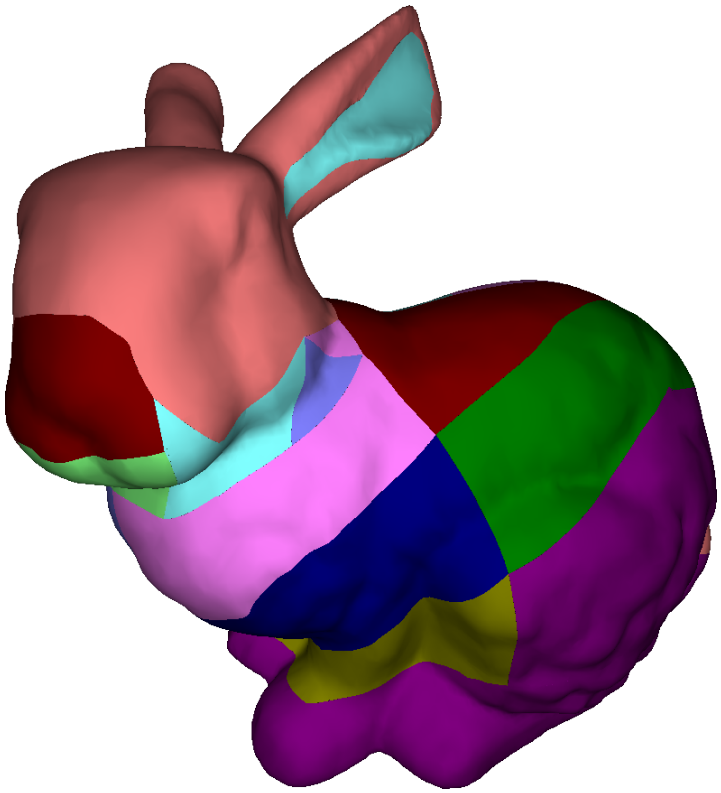


Figure 1.3: An input polyhedral surface mesh segmented based on the resulting fracture components.

Moreover, this process can be recursively applied to resulting components, achieving a hierarchical structure of increasingly smaller divisions, each with their own fracturing properties. This setup provides flexibility for a wide range of variability in both appearance and behavior of fractures among each level of division.

In addition, an accompanying tool called the Fracture Map Utility Application was developed to design, process, and export fractures using the algorithm presented in this paper. Fracture results on large meshes are obtained within three to four seconds, allowing for efficient iterative design practices.

Chapter 2

Background

2.1 Polyhedral Surface Meshes

The basic structure operated on within the Fracture Map Algorithm is a polyhedral surface mesh. A polyhedral surface mesh is a collection of polygons joined at their edges and encloses a 3-dimensional region. A mesh comprises of three elements:

Vertices :

Points in space representing corners of polygons within the polyhedral mesh.

Edges :

Joins two vertices together. Each edge consists of two `half edges` representing both directions along the edge.

Faces :

Polygons bounded by edges. Corners within the faces are represented by vertices.

A polyhedral mesh only contains explicit information about the surface of the object it represents. The implementation of the Fracture Map Algorithm found in the Fracture Map Utility Application handles only polyhedral meshes with triangular faces, however, can easily be adapted to handle more complex polygonal faces.

2.2 Texture Mapping

The given tessellation of a polyhedral surface mesh implies sufficient resolution for providing a description of the surface of an object in 3-dimensional space. However, the same resolution may be impractical for representing fine-grained detail along the surface. Common examples include diffuse color, opacity, specularity, and minute changes in topology. Generally, this additional information can be provided via an additional data structure called a `texture` [26]. Textures represent their data within a separate coordinate system, called `texture space`, of arbitrary dimensions. In 2-dimensional space, the X-axis is called the U-axis and the Y-axis is called the V-axis, forming `UV-coordinates`. Two primary types of texture exist, deriving distinction from its data representation.

One form of textures are procedural textures. Procedural textures represent data in mathematical form. This allows accurate samples to be retrieved at precise granularities. The Fracture Map Algorithm represents fracture maps as procedural textures, because accurate intersection tests are required.

A more common form of texture are bitmapped textures, the most prevalent type, stores a finite set of data elements, called `texels`. Texels are closely related to pixels, however, a texel may span multiple pixels in a final image. Unlike procedural textures, bitmapped textures are resolution dependent, trading speed for accuracy. The number of texels present determines the resolution of the bitmapped texture. Due to the fast nature of bitmapped textures during rendering, the Fracture Map Utility Application converts a fracture maps procedural representation to a bitmapped representation for display.

Textures can be applied to a mesh via a process called `texture mapping`, in which values are mapped to a mesh surface via texture coordinates provided at each vertex.

Sample positions within a texture are derived during rasterization by interpolating the texture coordinates of each triangle vertex. Each texture coordinate is a vector value representing the displacement from the bottom-left corner of a texture, within a range of [0-1] [26]. It is up to a user to determine assign texture coordinate values to a mesh. This can be done either procedurally, such as spherical mapping, or hard-coded within a mesh's file-format during.

During triangle rasterization, bitmapped texture data can be easily retrieved by fetching texel data nearest to the sample point's texture coordinate. More complex criteria exist to help against aliasing artifacts, but the concept remains the same. However, sampling procedural textures is comparably unintuitive and slow operation. For this reason, their use in rasterization is uncommon, although several attempts have been made introduce performant techniques [43, 47, 59, 38].

The Fracture Map Algorithm takes advantage of prevalent usage of textures within computer graphics. By allowing users to coordinate their fracture designs with supplementary textures, the user is given full control to align fractures with any details within the textures. This provides a principally artist-driven experience. A common example is designing fractures that align with features of a model's diffuse map.

2.3 Medial Axis

The medial axis of an object plays an important role in this work, as it contains important information about the geometric structure of an object. Figure 2.1 demonstrates the medial axis of a 2D shape. Intuitively, the medial axis of a shape is the skeletal structure implicitly representing the surface of the object. Although polyhedral surfaces only explicitly contain information regarding the surface of an object, the Fracture Map Algorithm uses the mesh's medial axis to derive information regarding

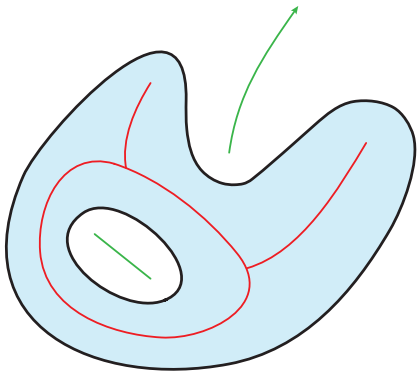


Figure 2.1: The medial axis of a 2D shape. The red and green lines indicate the medial axis of the shaped defined by the black lines. The axis can be classified into two categories: an inner (red) and outer (green) axis.

the inside of the mesh for hidden surface extraction.

The medial axis was first proposed by Blum [10, 11] as a tool for measuring biological shapes by representing a 2-dimensional shape via a 1-dimensional structure. Subsequently, the medial axis has been found to be useful in numerous applications including mesh deformations [60, 8], feature detection [27], shape manipulation [54, 9], and surface reconstruction [2, 3]. Thus, medial axis extraction has become a significant area of research [60, 8, 27, 54, 9].

The medial axis, commonly referred to as the skeleton, is formally defined in [61] as:

Definition 1. *Let S be a manifold surface embedded in* $\mathbb{R}^d$*. Let M be the closure of the locus of maximal inscribed* $(d-1)$*-spheres laying tangential to at least two points of S. These n-spheres are called* `medial balls`*. The medial axis is defined as the set of centers of M.*

A maximal ball is defined as an n-sphere that is not fully contained in any other n-sphere [46]. Thus, the medial axis in 2D consists of a set of lines and curves, while

the medial axis in 3D is formed of a stitching of quadric surfaces by curve segments of quadratic degree four [61]. When the surface S is convex, the medial axis in 2D comprises of only line segments and in 3D only planar surface segments. Notice in Figure 2.1 each point on the surface is associated with two medial balls. One originates from the inside of the surface while another from the outside. This makes a distinction between an inner and outer medial axis.

The medial axis has four major elements: `sheets`, `seams`, and `junction` points. A sheet is a subset manifold quadratic surface of an axis that is equidistant to two points on its boundary surface. A seam is a curve along the axis that is equidistant to three or more points on its boundary surface and stitches together two or more sheets. A junction is a point equidistant to three or more points on its boundary surface and is found at the intersection of three or more seams [17, 55].

2.4 Triangulation

Triangulation is the process of dividing a simple polygon into a set of triangles. Triangulation is used within the Fracture Map Algorithm to convert a triangle that has been divided into a set of fractured polygons back into a triangular mesh. The difficulty of a triangulation varies depending to its geometry properties. If a simple polygon is convex, triangulation is a trivial operation. Adding a diagonal from one vertex to all other vertices completes the triangulation in linear time. If a fracture polygon is found to be a concave polygon, triangulation becomes a non-trivial operation.

One of the first methods introduced for performing concave simple polygon triangulation in the late 1970's is the *ear clipping method* [29]. Figure 2.2 illustrates basic process of the algorithm. Although its implementation is simple, the solution is sub-optimal and has a time complexity of $O(n^2)$. Solutions have since strived to achieve

linear complexity alongside convex triangulation. Tarjan and Van Wyk [57] introduced a method with a time complexity of $O(n \log\log n)$. In 1991, Chazelle [15] presented a method that matched convex simple polygon's complexity of $O(n)$, however the implementation is widely regarded to be exceedingly complex [29].

The ear clipping method is a straightforward algorithm taking advantage of the fact that simple polygons without holes always have at least two non-overlapping *ears* [34]. Given three adjacent vertices V_1, V_2, and V_3, of a polygon $P = V_1, V_2, V_3, \ldots V_n$ where $n > 3$, the triangle formed by V_1, V_2, and V_3 is considered an ear if the diagonal between V_1 and V_3 lies entirely inside P [34]. Two ears are non-overlapping if their regions are disjoint.

Non-overlapping ears are recursively removed the polygon by creating an edge between V_1 and V_3 and removing V_2 from the polygon.

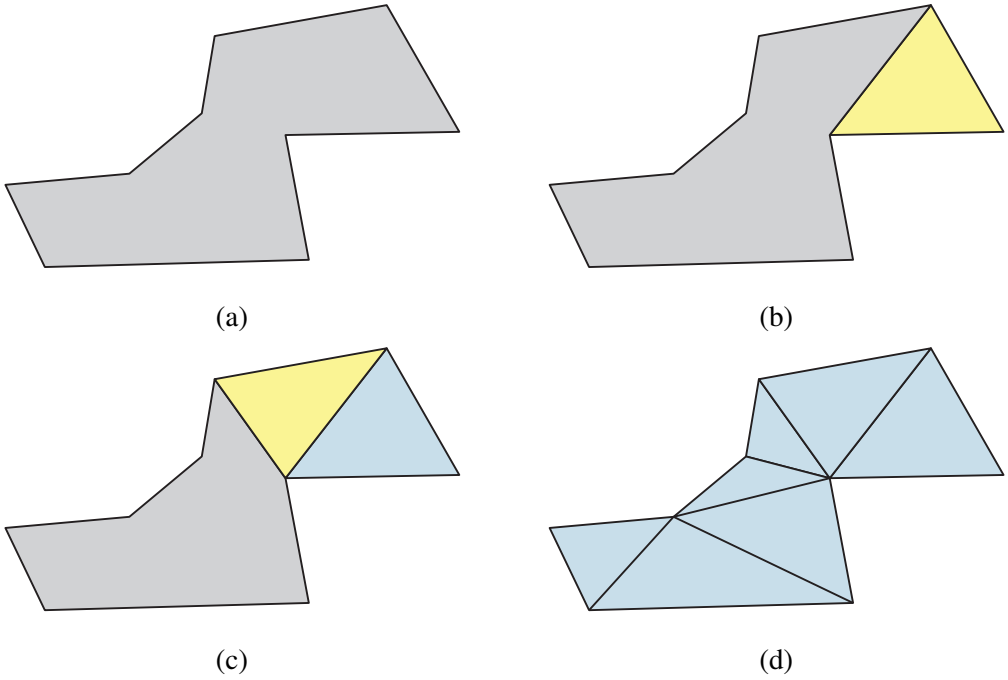


Figure 2.2: The ear clipping method: The input polygon is shown in (a), with the first ear highlighted in (b). After the first ear has been removed from consideration, the next ear is found, shown in (c), and the process is repeated until the final triangular mesh is computed.

Chapter 3

Related Work

There are a number of important areas of related work for the Fracture Map Algorithm. Section 3.1 will explore some of the work that exists for extracting a medial axis from various object types. Section 3.2 discusses some of the existing techniques around 3-dimensional mesh surface cutting.

3.1 Medial Axis Extraction

Although the medial axis is a notably powerful primitive for many operations, its exact extraction has proved to be a difficult task. This has led to the development of faster methods for deriving an approximation. Fortunately, for most applications the accuracy of a medial axis approximation suffices. This holds true for the Fracture Map Algorithm as well.

There are a number of methods for approximating the medial axis of a surface. Section 3.1.1 introduces methods that use a tracing approach. Another set of methods, described in Section 3.1.2, utilize voxels for extraction. A third method, examined in Section 3.1.3 and used in this paper, utilizes the Voronoi diagram geometric construct.

3.1.1 Tracing Method

Common and often more accurate algorithms use a tracing technique for extracting a medial axis. These techniques work by tracing seams of a medial axis until a junction point is reached. The process is repeated, tracing each seam emerging from the junction until another is found. The process is initiated along a seam that is easy to locate, such as a convex trivalent vertex. The quality of the medial axis generated depends on the size of the tracing step used.

Sherbrooke, et al. [50] presents one of these algorithms for deriving the medial axis of a polyhedron. Culver, et al. [17] extended this work to tetrahedrons and introduced a number of performance optimizations. Ramanathan and Gurumoorthy [45] introduces a variation that functions on free form objects by treating them as a finite set of Non-Uniform Rational B-Spline (NURBS) surface patches, whose medial axis are more easily and accurately derived than discretization techniques. Sud, et al. [55] improves upon medial axis generation for polyhedrons by removing unstable features while preserving homotopy.

3.1.2 Voxel-Based Method

Voxel-based techniques utilize a division an object's space into a set of volumetric units called voxels. Lam, et al. [33] presents a survey of early voxel-based techniques called thinning algorithms. These methods iteratively remove layers of voxels until a sufficiently minute set skeleton is obtained. However, the skeleton is not guaranteed to be medial. Saito and Toriwaki [48] and Pudney [42] extended thinning methods by also considering voxels in order of distance, producing a medial result.

Many voxel-based methods use a distance field. Danielsson [19] introduced an efficient method for deriving the distance to the surface for each pixel, establishing

a distance field. Ragnemalm [44] extended this work to 3-dimensions. Siddiqi, et al. [51] uses a distance field to derive the Hamiltonian flow and outward flux of each voxel to deduce their proximity to the medial axis. Foskey, et al. [23] iteratively finds lines intersecting sheets of the medial axis by matching voxels that meet a separation criteria. That is, their respective vectors diverge, typically representing voxels on the opposite sides of an axis sheet. Dryden, et al. [41] and Prohaska and Hege [18] introduces the use of geodesic distance to take the shape of an object's surface into account.

3.1.3 Voronoi-Based Method

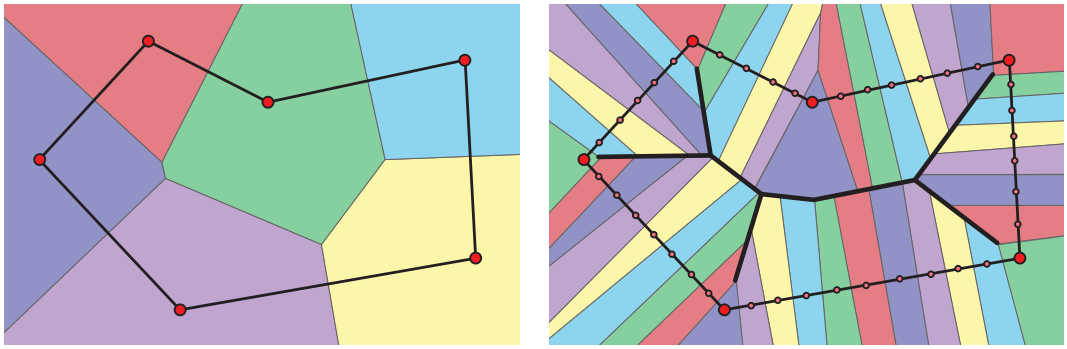
Voronoi-based techniques utilize Voronoi diagrams to derive the medial axis. Given a set of vertices, Voronoi diagrams divide space into regions based on the smallest proximity to a single vertex. Under suitable circumstances, the edges of each regions will converge towards the center of regions surrounded by the set of vertices. A variation of this technique is utilized by the Fracture Map Algorithm, due to its relatively fast computation and robust results.

Definition 2. *The Voronoi diagram $VD(P)$ of a discrete set $P \in \mathbb{R}^d$ is the subdivision of space into Voronoi cells V_p for each point $p \in P$ such that:*

$$V_P = \{x \in \mathbb{R}^d \,|\, \|x-p\| \leq \|x-q\|, \forall q \in P\}$$

where $\|x-p\|$ is the Euclidiean distance between points $x, p \in \mathbb{R}^d$

The Voronoi diagram can be interpretted as a collection of Voronoi faces of varying dimensions. A k-dimensional Voronoi face in $\mathbb{R}^d$ represents the set of points equidistant from $d+1-k$ points in P and the intersection of $d+1-k$ Voronoi cells. A 0-dimensional Voronoi face is a Voronoi vertex, a 1-dimensional Voronoi face is a



(a) (b)

Figure 3.1: The Voronoi diagram of a 2D shape. As the number of vertices of the shape increases, the Voronoi vertices begin to converge onto the shape's medial axis.

Voronoi edge, a 2-dimensional Voronoi face is a Voronoi polygon, and a 3-dimensional Voronoi face is a Voronoi cell.

Each Voronoi cell V_p of a point p is a closed and possibly unbounded polytope and contains `Voronoi vertices` v_p equidistant from p and $d+1$ other points of P. The `Voronoi ball` centered at v_p contains these points on its boundary and no other points in its interior. This is analogous to the maximal balls which define the medial axis, making the Voronoi diagram an important tool for medial axis derivation, as shown in Figure 3.1.

When P is a dense sampling of a surface S, the Voronoi vertices can be expected to approximate the medial axis. This holds true in 2D, as Brandt [12] proved that as the density of surface sampling approaches infinity, the Voronoi vertices converge on the medial axis. The same property has not been found to be true in 3D, as Voronoi vertices can often be found close to the surface S and consequently far from the medial axis.

This occurs when four sample points lay close together on S, placing the corresponding Voronoi vertex very close to S as well. When densely sampled, Voronoi cells

are long, skinny, and perpendicular to the tangent of S at p. This is due to V_p being bounded quickly after extending along the tangent by the bisecting planes of each neighbor on S. Furthermore, V_p is bound by the medial axis in the direction perpendicular to the surface tangent at p, as other points on S become closer than p beyond this point. Amenta [2] noted this fact and found that convergence could be guaranteed in 3D if only a subset of Voronoi vertices, called `poles`, are considered. Poles are the two Voronoi vertices in V_p that are the furthest distance from p and therefore lay near the medial axis.

3.2 3-Dimensional Mesh Cutting Algorithms

Cutting of 3-dimensinal meshes is a widely researched field of computer graphics, with significant applications in medical simulations, computer aided design, feature films, and video games [53]. The concepts found in 3-dimensional cutting techniques are very similar in nature to the Fracture Map Algorithm. A large portion of research has been placed on supplying a realistic re-creation of surgical cutting in simulations, whereas the technique presented in this paper deals exclusively with a pre-determined cutting path.

Many of the ideas were first introduced by Mortensen and Barrett [36], with a method for intelligently segmenting 2-dimensional images around a regions of interest. Chun-Ho, et al. [58] extends this method into 3-dimensions, representing a mesh via surface voxels and their gradient values. Dijkstra's algorithm is then used to find vertices on the mesh that most closely resemble a target cutting contour.

Many 3-dimensional techniques focus minimal effort to accurately split along the cutting edge, such as Delingette, et al. [20] and Frisken-Gibson [24], and instead simply remove intersecting elements from the mesh. Alternatively, Viet, et al. [28]

duplicate vertices closest to the cutting instrument and generate separated connectivity.

Cuts are refined further in Bruyns and Senger [13] by using a state-machine to determine when vertices need to be inserted along intersection edges. Bielser, et al. [7] continues along this concept and uses a state machine based on the transitions of a cutting tool across a tetrahedral's topology to perform the necessary subdivision to represent arbitrary intersections. This is extended in Seokyeol, et al. [30] by using a turning point determination algorithm based on local curvature of surface meshes.

Although all of these provide methods for performing cuts on a mesh, they do not adequately handle the exposure of a new hidden surface within a surface mesh. Sela, et al. [49] handles the hidden surface in a manner geared towards surgical simulation by duplicating the start and end cut vertices and extending them in the opposite direction of a cutting instruments orientation. This produces an incision into the mesh but restricts the cut from producing a disjoint segmentation.

Techniques such as Delingette, et al. [20], Bielser, et al. [7], O'Brien and Hodgins [40], Smith, et al. [52], Steinemann, et al. [53] instead operate on tetrahedrals and Frisken-Gibson [24] on voxel arrays, both with explicitly defined interiors.

Chapter 4

Fracture Map Algorithm

4.1 Overview

The goal of the Fracture Map Algorithm is to allow users to take a polyhedral surface mesh and derive a new set of meshes based on fracture lines defined within a 2-dimensional texture. Although no mathematical guarantees are made by the algorithm, it works in practice given proper limitations are met. The algorithm takes two inputs: a mesh, and a data structure defining the lines of fracture. The input mesh is a polyhedral surface mesh bounding 3-dimensional space with a texture coordinate mapping. The input fracture data, defined by the user and called a `fracture map`, is mapped onto the surface of the polyhedral mesh according to its UV coordinates. From these inputs, the algorithm produces a set of meshes that would result had the input mesh been fractured along the lines of fracture defined by the fracture map.

The algorithm is performed in seven distinct steps:

1. **Fracture edge generation** – Identify locations along the edges of each triangle that intersect a fracture line.

2. **Fracture point matching** – Identify the interconnectivity of fracture lines within each triangle.

3. **Fracture edge resolution** – Resolve the exact location of a fracture along a triangle's edge.

4. **Triangulation** – Translate the connectivity information into a set of new triangles whose edges align with the fracture lines.

5. **Fracture component segmentation** – Segment the new set of triangles into groups based on those surrounded by the same fracture lines.

6. **Hidden surface generation** – Generate a hidden surface of the input mesh that is exposed by the fractures.

7. **Surface stitching** – Stitch together the hidden and visible surfaces to create a final set of watertight meshes.

The algorithm has been observed to derive correct results given a number of limitations:

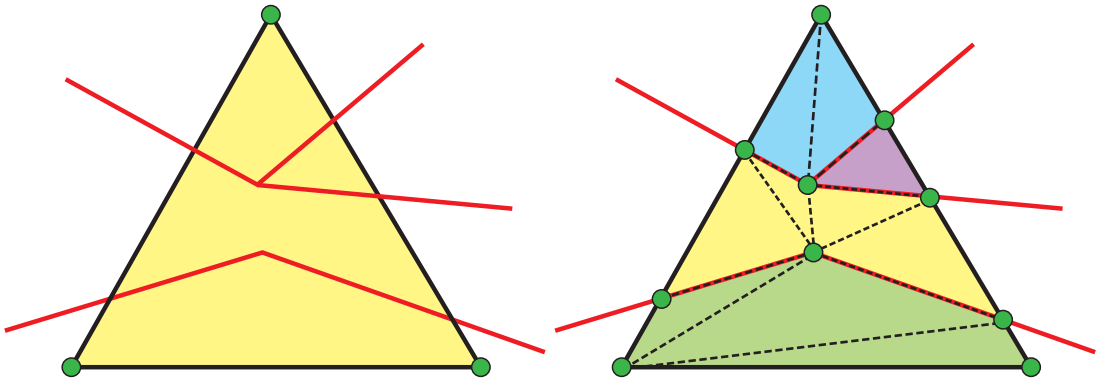
1. There are no intersections across edges of the fracture map.

2. The mesh is sufficiently tessellated that there is no greater than one fracture component enclosed completely within a triangle.

3. The number of edges a fracture node contains is no larger than thirty-two.

4. The input mesh does not contain exceedingly narrow curved regions that may result in the artifacts discussed in Section 6.3.1.

Section 4.3 will discuss the process of extracting points of fracture within each triangle. Section 4.4 will describe the fracture matching process, in which spans are established between points of fracture. Additionally, fracture forks within the interior



(a) (b)

Figure 4.1: An input mesh triangle with intersecting fracture lines shown in (a). (b is the resulting triangular mesh and fracture components after applying the Fracture Map Algorithm.

of the triangle are identified. In Section 4.5, the process of resolving fractures between adjacent triangles is discussed. Section 4.6 examines the re-triangulation of the mesh after additional fracture vertices have been added. The result up to this step is shown in Figure 4.1. Section 4.7 shows how the fracture mesh is segmented based on the lines of fracture. The process is concluded with Section 4.8 and 4.9, presenting the process of extracting a hidden surface from the original mesh and stitching it together to create a finished watertight mesh.

4.2 Terminology

There are a number of important terms used repeatedly throughout the following sections.

Fracture map :

A fracture map is an undirected simple graph with nodes that represent a position in UV-space and edges that represent a fracture line between the two points. For efficient retrieval, edges are stored within a quadtree, based on the bounding

box formed by its two nodes. Representing fractures in UV-space provides the advantage of being resolution-independent. If fracture maps were represented directly as raster images, the quality of fractures would be dependent highly on a fracture map's resolution.

(The last two sentences may not fit here.)

Fracture point :

Represents a point within a mesh that must be resolved to a new vertex within a fractured mesh. There are three types of fracture points:

Half edge fracture point :

Represents a point along a mesh's half edge that intersects a fracture line. For a triangle, its half edge fracture points represent all positions along its boundary that cross a fracture line. A half edge fracture point's position is stored relative to the parameterization of its half edge.

Fracture point branch :

Represents a convergence of multiple fracture lines into a single point within a triangle. If a fracture map contains nodes of only degree 2, there will be no fracture point branches.

Fracture point join :

Represents a point inside a triangle that corresponds to a node of degree 2 within a fracture map.

Each fracture point type is illustrated in Figure 4.2.

Fracture edge vertex :

Represents a vertex created from a half edge fracture point.

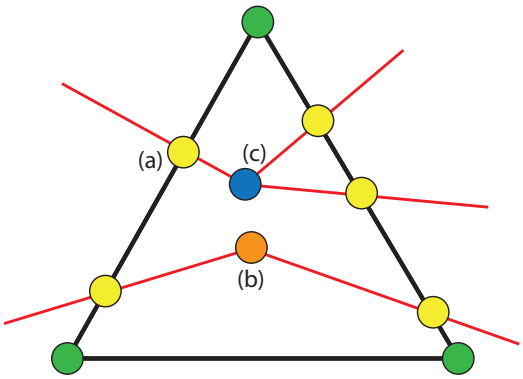


Figure 4.2: The three different types of fracture point. (a) represents a half edge fracture point, (b) represents a fracture point join, and (c) represents a fracture point branch.

Fracture interior vertex :

Represents a vertex created from a fracture point branch or fracture point join.

Fracture half edge :

Represents a half edge and all of its fracture points. Half edge fracture points are stored in ascending order of their distance away from its half edge's first vertex in UV-space.

Fracture edge :

Represents an edge and all of its fracture points that have been realized into fracture edge vertices. A fracture edge is created by resolving two adjacent fracture half edges. The resolving process establishes a final position for each fracture point and creates a corresponding fracture edge vertices.

Matched triangle :

Represents a triangle with fracture half edges, fracture point branches, and connectivity information between fracture points. A matched triangle is used during triangulation to establish a set of polygons defining separate segments of a fracture.

4.3 Fracture Edge Generation

The first stage of the Fracture Map Algorithm is to generate a `fracture half edge` for each half edge within the polyhedral mesh. Generating a fracture half edge is a simple process, shown below:

Algorithm 1: Fracture half edge generation

```
input  : A 3-dimensional polyhedral mesh and a fracture map
output: A fracture half edge for every half edge within input mesh
foreach triangle in mesh do
    bounds ← bounding box of triangle
    candidates ← fracture lines intersecting bounds
    foreach half edge in triangle do
        fracedge ← empty fracture half edge
        foreach line in candidates do
            if line intersects half edge then
                edgedist ← distance along half edge
                add new half edge fracture point to fracedge
            end
        end
        sort half edge fracture points by edgedist
    end
end
```

Fracture half edges are generated by first iterating over each half edge within the input mesh. For each half edge, a bounding box of the edge in UV-space is used to

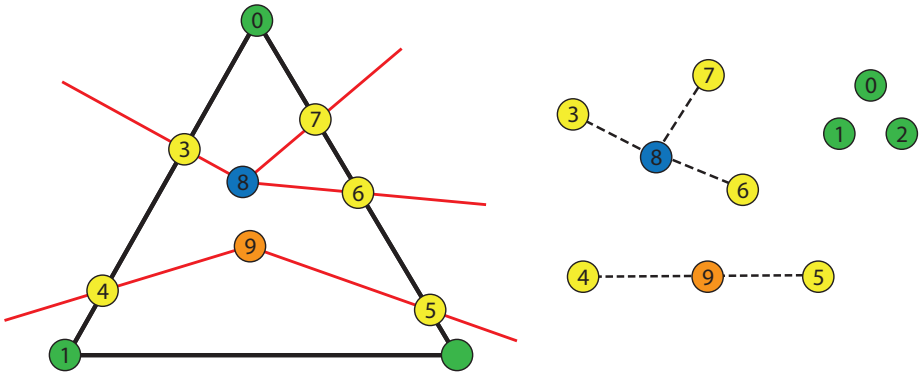
retrieve candidate fracture lines from the fracture map's quadtree. This minimizes the number of line-intersection tests required for generation. The intersections are found via the following line-line intersection test:

Definition 3. *Let L_1 and L_2 represent two lines in 2-dimensional space defined by the points (U_1, V_1) and (U_2, V_2) for L_1 and (U_3, V_3) and (U_4, V_4) for L_2. The point of intersection of line L_1 and L_2 is defined as:*

$$P_x = \frac{\begin{vmatrix} \begin{vmatrix} U_1 & V_1 \\ U_2 & V_2 \end{vmatrix} & \begin{vmatrix} U_1 & 1 \\ U_2 & 1 \end{vmatrix} \\ \begin{vmatrix} U_3 & V_3 \\ U_4 & V_4 \end{vmatrix} & \begin{vmatrix} U_3 & 1 \\ U_4 & 1 \end{vmatrix} \end{vmatrix}}{\begin{vmatrix} \begin{vmatrix} U_1 & 1 \\ U_2 & 1 \end{vmatrix} & \begin{vmatrix} V_1 & 1 \\ V_2 & 1 \end{vmatrix} \\ \begin{vmatrix} U_3 & 1 \\ U_4 & 1 \end{vmatrix} & \begin{vmatrix} V_3 & 1 \\ V_4 & 1 \end{vmatrix} \end{vmatrix}} \quad P_y = \frac{\begin{vmatrix} \begin{vmatrix} U_1 & V_1 \\ U_2 & V_2 \end{vmatrix} & \begin{vmatrix} V_1 & 1 \\ V_2 & 1 \end{vmatrix} \\ \begin{vmatrix} U_3 & V_3 \\ U_4 & V_4 \end{vmatrix} & \begin{vmatrix} V_3 & 1 \\ V_4 & 1 \end{vmatrix} \end{vmatrix}}{\begin{vmatrix} \begin{vmatrix} U_1 & 1 \\ U_2 & 1 \end{vmatrix} & \begin{vmatrix} V_1 & 1 \\ V_2 & 1 \end{vmatrix} \\ \begin{vmatrix} U_3 & 1 \\ U_4 & 1 \end{vmatrix} & \begin{vmatrix} V_3 & 1 \\ V_4 & 1 \end{vmatrix} \end{vmatrix}}$$

Given the lines L_1 and L_2 and not parallel.

If a point of intersection resides within the line segment (x_1, y_1) and (x_2, y_2), a half edge fracture point is created for the point and is given a `fracture ID` unique to the triangle. The half edge fracture point is then added to the fracture half edge. Once all candidate fracture lines have been considered, the fracture points are sorted by their distance along the half edge.



(a) (b)

Figure 4.3: Nodes in the undirected simple graph represent vertices or fracture points of the triangle, and edges represent connectivity via fracture lines. The numbers within each node represent the node's fracture ID.

4.4 Fracture Point Matching

After all three fracture half edges of a triangle have been generated, the triangle moves on to a matching phase of the algorithm. This process resolves inter-connectivity between fracture points of each fracture half edge. As a result, `fracture point branches` and `fracture point joins` necessary for connectivity are identified and added to the triangle. Connectivity within a triangle is represented as an undirected simple graph of vertices, fracture points, and their connectivity via fracture lines, as shown in Figure 4.3. When all connectivity has been established, the result is a matched triangle, ready to be triangulated.

Matched triangles are created by iterating over each half edge fracture point within the given triangle, and performing the following steps:

1. The half edge fracture point is checked to see if it has already been matched with another fracture point. If so, no further processing is needed.

2. If unmatched, the fracture line that intersects the fracture point's position is re-

trieved and set as the current line.

3. All other half edge fracture points within the triangle are checked to see if they derive from the current fracture line.

4. If there is a match, a span between the two fracture points is added and no further processing is needed for the current fracture point.

5. If no matches were found, then an endpoint of the current fracture line resides within the interior of the triangle. This leaves two possible scenarios:

   (a) The interior endpoint has multiple adjacent fracture lines, corresponding to a fracture branch. If a fracture point branch does not already exist at this position, one is created and a span is added between the current half edge fracture point and the newly created fracture point branch. No further processing of the current fracture point is needed, but the fracture point branch is added to a queue for additional matching later.

   (b) The interior endpoint has only one additional adjacent fracture line. A new fracture point join is created at the endpoint and a span is added to the current fracture point. The algorithm now sets the fracture point join as the current fracture point and advances to the adjacent fracture line and sets it as the current line. Repeat from Step 3.

This process fully establishes all spans that do not traverse more than a single fracture point branch. For most cases, this will suffice. However, when multiple fracture point branches are connected within a triangle, further processing is required to establish its span.

The algorithm processes a queue of fracture point branches, which are enqueued after creation. For each fracture point branch in the queue, the following is performed:

1. Each fracture line associated with the fracture point branch is first added to a processing queue.

2. A fracture line is popped from the queue and treated as the current line.

3. The current fracture line is checked to see if its next endpoint matches that of another fracture point branch. If so, a span is added.

4. Otherwise, three scenarios are possible:

   (a) The fracture lines intersects one of the triangle's half edge. This means a span has already been created between the fracture point branch and its connected half edge fracture point.

   (b) There is no intersection with a half edge, and the next endpoint branches into multiple fracture lines. This means a previously unknown fracture point branch has been found. A new fracture point branch is created and added to the processing queue, and a span between the two fracture points is added.

   (c) There is no intersection with a half edge, and the next point has only one additional adjacent fracture line, meaning it is a fracture point join. A fracture point join is created, a span is added between the two fracture points, and the next line segment is added to the processing queue, repeating the process.

5. Repeat Step 2 until the queue is empty.

The result is a fully populated matched triangle, which contains all topology information of the fractures within a given triangle. However, due to the potential disconnect between texture coordinates across triangle edges, the positions of half edge fracture points in model-space may vary slightly across edges.

4.5 Fracture Edge Resolution

Fracture edge resolution negotiates the fracture information of two adjacent fracture half edges into a single unified fracture edge. Fracture edge resolution is also responsible for creating fracture edge vertices along the edge for use in triangulation. An assumption made for the sake of fracture edge resolution is that both fracture half edges have the same number of half edge fracture points and differ only in their position along the half edge. If the number of half edge fracture points differs, then the fracture map is ill-formed.

The process of resolving fracture edges changes based on the configuration of the two fracture half edges:

The two half edges share the same texture coordinates :

Both fracture half edges have fracture points positioned at identical positions, but in opposite order. Each fracture point's position in model-space and texture coordinate is derived via linear interpolation of edge vertices and texture coordinates, respectively. The order within the fracture edge is inconsequential.

Only one half edge exists :

This can arise when holes exist within a mesh. The fracture edge vertices are created from the single fracture half edge in the same manner as above.

The two half edges have different texture coordinates :

Both fracture half edges represent the same fracture points, but each fracture point may lay at different positions along the half edge. When resolving the edge, positions along the half edges are averaged and used to linearly interpolate as above.

With the conclusion of fracture edge resolution, all vertices and texture coordinates along an original mesh's edges have been realized into fracture edge vertices. As a final step, each fracture point branch and fracture point join is realized into a fracture interior vertex based on its barycentric coordinate.

4.6 Triangulation

4.6.1 Fracture Polygon Creation

Up to this point, all vertices and texture coordinates sufficient for modeling the fractures along the original mesh's surface have been created, and fracture topology information derived. The topology information is now translated from the matched triangles into a triangulation of a polyhedral surface mesh. Triangulation occurs on a per-triangle basis, dividing each triangle into a set of `fracture polygons`. Each fracture polygon is then triangulated, creating the final polyhedral representation of a fractured triangle.

Although all necessary vertices exist at this point, triangulation is performed using fracture IDs of each triangle's fracture points. This is because all connectivity information is captured for fracture points. After each fracture polygon has been created, it is *realized* by translating all fracture IDs into their respective fracture vertex IDs.

The process for creating the set of fracture polygons, also shown in Figure 4.4, works by first establishing a single edge of the fracture polygon that lays against the triangle's boundary. The algorithm then walks along the triangle's edge or spans between two fracture points and vertices until it returns to the original fracture point. At the end of this process, all fracture polygons with an edge along the triangle's boundary will be created.

For most fractures, this will suffice, however, it may be possible that a fracture polygon is surrounded entirely by other fracture polygons. To address this, each half edge is treated as having individual segments, separated by half edge fracture points, or lack thereof. A bitflag is used to store a visitation state of each segment. The segments' order follows the triangle's winding order and decides which bit corresponds to which segment. For interior spans, the same idea is used. All fracture point branches and fracture point joins maintain a bitflag to store which of its spans have been visited. A fracture point only marks a span as visited if the algorithm traverses away from the fracture point along its span.

For the purpose of this section, both half edge fracture points and vertices of the triangle are referred to as `edge triangulation points`. Fracture point branches and fracture point joins are referred to as `interior triangulation points`. For each edge triangulation point, its adjacent segment extending in the direction of the triangle's winding order is called its `forward segment`.

To begin the process, each vertex and half edge fracture point is added to a processing queue following the winding order. The process next performs the following steps:

1. The processing queue is popped to retrieve the next edge triangulation point to be processed.

2. If its forward segment has already been visited, the triangulation point is skipped and Step 1 is repeated.

3. Otherwise, an empty fracture polygon is created and the first triangulation point is added to it.

4. The edge triangulation point adjacent to the forward segment is set as the current

triangulation point, and the forward segment is set as visited.

5. While the algorithm hasn't looped back to the fracture polygon's first triangulation point, the following is repeated:

   (a) The current triangulation point is added to the fracture polygon. This establishes the fracture polygon's first edge along the triangle's boundary during the first iteration of the loop.

   (b) Based on the number of spans the current triangulation point has (excluding spans to the previous triangulation point), the following is performed:

   The triangulation point has a single span :

   The algorithm advances along the span, marking it as visited, and sets the adjacent triangulation point as current.

   The triangulation point has multiple spans :

   The algorithm traverses along the span that minimizes θ in the equation $\theta = \frac{(T_c - T_p)}{\|(T_c - T_p)\|} \cdot \frac{(T_s - T_c)}{\|(T_s - T_c)\|}$ where T_c is the texture coordinate of the current triangulation point, T_p is the texture coordinate of the previous triangulation point, and T_s is the texture coordinate of the triangulation point adjacent to the span. The traversed span is marked as visited and the adjacent triangulation point is set as current.

   Otherwise :

   The algorithm advances along the forward segment, marks it visited, and retrieves the next edge triangulation point. It is set as the current triangulation point.

   (c) Repeat Step 5.

6. The fracture polygon now contains a list of all triangulation points, representing the traversal along each edge of the polygon. Add the fracture polygon to the

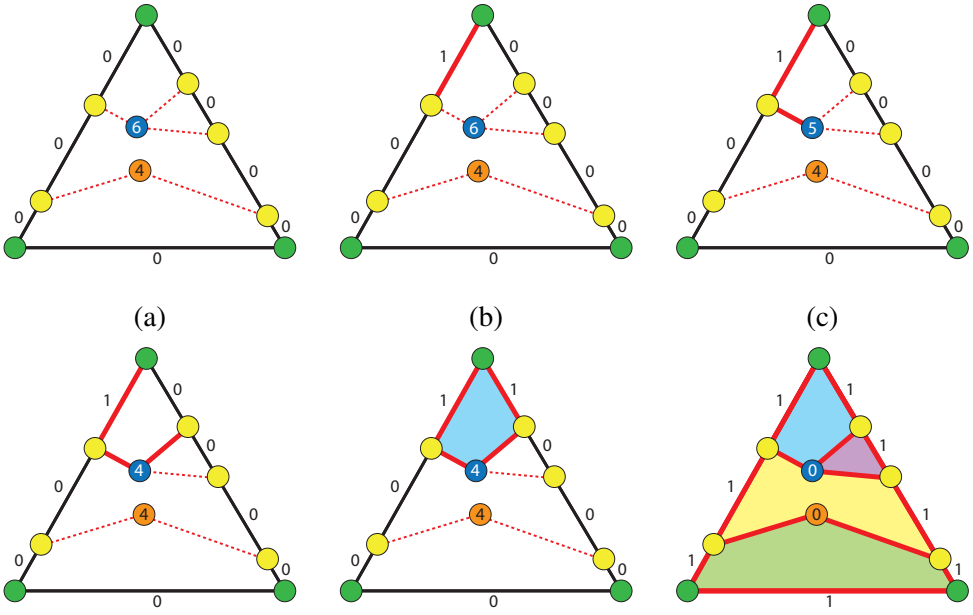


Figure 4.4: The step-by-step process for triangulating a matched triangle into a set of fracture polygons.

triangle's list of polygons and repeat Step 1.

At the end of this process, each interior triangulation point is checked to ensure each span has been visited. If not, the same process is repeated using the unvisited span as the initial edge of a fracture polygon.

Once finished, each fracture polygon has to be *realized*, translating its representation from fracture points to fracture vertices. This results in a final set of fracture polygons. The last step of the process is to triangulate each polygon to create the final polyhedral representation of the fractured triangle.

4.6.2 Convex Fracture Polygon Triangulation

During fracture polygon creation, as each triangulation point is added, a convex test is performed:

Definition 4. *Given three triangulation points, T_c, T_p, T_n, representing the texture coordinates current, previous, and next triangulation points respectively, and the vectors $V^c = \frac{(T_c - T_p)}{\|(T_c - T_p)\|}$ and $V^n = \frac{(T_n - T_c)}{\|(T_n - T_c)\|}$, the fracture polygon is flagged as concave if it fails the point-plane test:*

$V^{c\perp} \cdot V^n \geq 0$ *where* $V^{c\perp} = (-V^c_y, V^c_x)$.

If a fracture polygon passes all tests, triangulation of a convex simple polygon is trivial. Adding a diagonal from one vertex to all other vertices completes the triangulation in linear time.

4.6.3 Concave Fracture Polygon Triangulation

If a fracture polygon is found to be a concave polygon, triangulation is non-trivial. Due to the limited occurrence of large concave polygons, the Fracture Map Algorithm uses the ear clipping method. However, if increased performance is desired, alternative methods described above are fully capable of achieving identical results. Upon completing the triangulation process on all triangles, a new polyhedral representation of the input mesh is produced, such that sufficient edges have been added to separate individual fracture component.

4.7 Fracture Component Segmentation

Fracture component segmentation is the process of separating a fracture surface mesh into subsets of triangles called `fracture components`. A fracture component is a polyhedral formed from a connected subset of triangles within a fractured mesh that are bounded completely by `fracture line edges`. Fracture line edges are edges created from a span to a fracture point, and therefore a line of fracture. The algorithm for segmenting a fracture mesh is a trivial recursive expansion:

Algorithm 2: GenerateFractureComponents(...)

```
input  : A fracture mesh
output: A set of polyhedrals
begin
    componentID ← 0
    foreach triangle in fracture mesh do
        if triangle has not been visited then
            ExpandComponent (rootTriangle, componentID)
            componentID ← componentID + 1
        end
    end
end
```

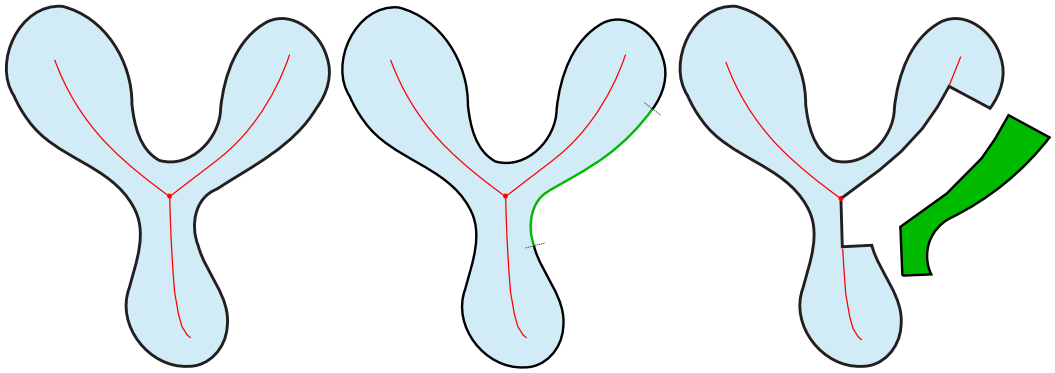
Algorithm 3: ExpandComponent(...)

```
input  : The current triangle
output: The current fracture component id
begin
    triangle.visited ← true
    triangle.componentID ← componentID
    foreach edge in triangle do
        if edge is not a fracture line edge then
            // OtherTri returns the other triangle shared by the edge
            ExpandComponent(OtherTri(edge, triangle), componentID)
        end
    end
end
```

4.8 Hidden Surface Generation

The fracturing process above results in a segmentation of the fracture mesh surface, but no information regarding the hidden surface of each fracture segment is explicitly known. However, implicit information can be derived from the mesh's medial axis. For this paper's objective, the inner medial axis can equate to the hidden surface of individual fracture segments, as illustrated in Figure 4.5.

Following the fracturing of the surface mesh from a fracture map, the medial axis is extracted via a Voronoi-based method extended from the method described by [27]. Rather than directly attempting to calculate the Voronoi diagram, the first step taken is



(a) (b) (c)

Figure 4.5: Hidden surface generation of a 2D shape. The red lines represent the medial axis of the 2D shape. The green line represents the portion of the surface contained in a fracture segment.

to generate the Delaunay triangulation of the original surface mesh via the *Quickhull algorithm* [6]. A Delaunay triangulation is an important geometric construct complimentary to the Voronoi diagram, and allows for its construction. u

Definition 5. *The Delaunay triangulation D_P for a set of points $P \subset \mathbb{R}^d$ is a triangulation such that the circumscribing ball of all d-simplices does not contain any other points inside.*

The Delaunay triangulation has the fortunate property of being the dual graph of the Voronoi diagram. This allows the Voronoi diagram to be derived directly from the Delaunay triangulation obtained from the Quickhull algorithm. Each k-simplex in D_P is dual to the $d-k$ dimensional face of $VD(P)$. Thus, in 2D the circumcenter of a Delaunay triangle is dual to a Voronoi vertex, a Delaunay edge is dual to a Voronoi edge bisecting the edge of D_p, and a Delaunay vertex is dual to a Voronoi cell. In 3D, a Delaunay tetrahedron is dual to a Voronoi vertex positioned at the center of the tetrahedron's circumsphere, a Delaunay triangle is dual to a Voronoi edge, a Delaunay edge is dual to a Voronoi facet, and a Delaunay vertex is dual to a Voronoi cell. This property allows the Voronoi diagram to be quickly and effectively generated for the

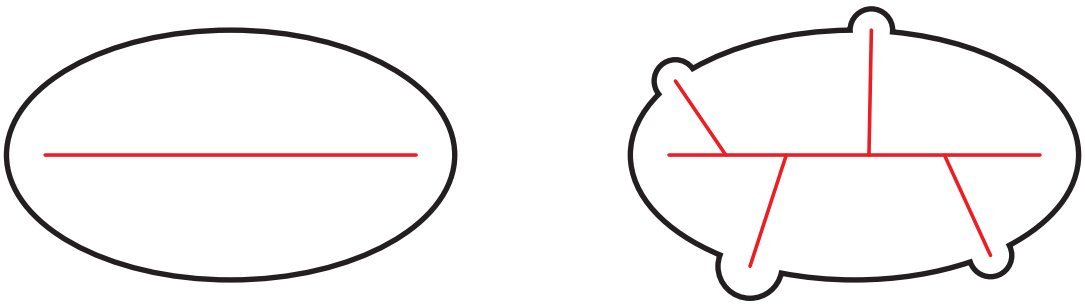


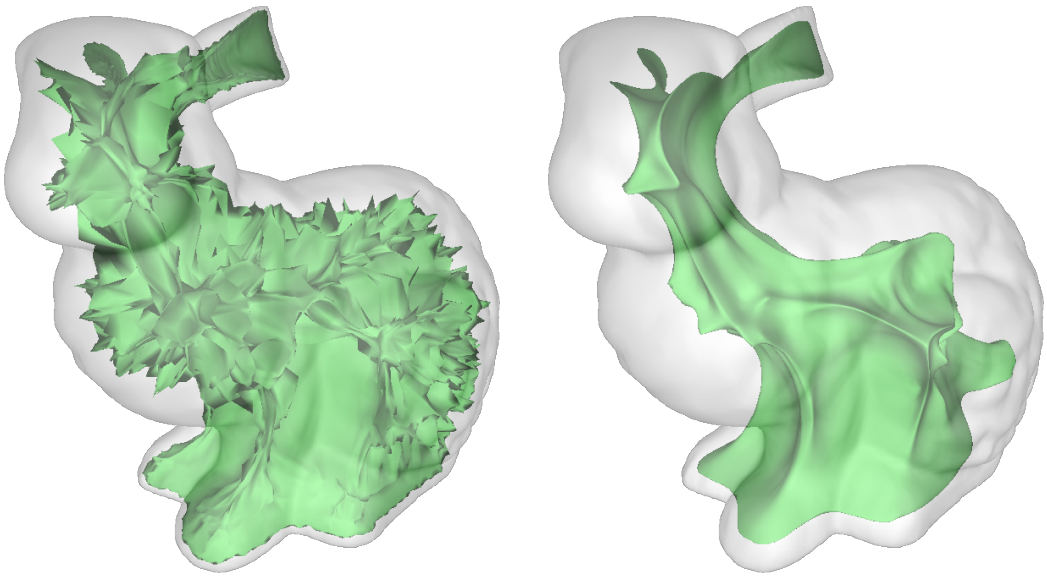
Figure 4.6: Slight perturbations within a surface mesh can cause dramatic changes in the structure of the medial axis.

surface mesh.

The poles of each Voronoi cell are found after discarding Voronoi vertices that lay outside the plane defined by the surface point p's position and averaged normal. The resulting poles converge on only the desired inner medial axis.

A major disadvantage of the medial axis is its sensitivity to perturbations of the original surface mesh. As shown in Figure 4.6, small variations in the surface mesh can cause dramatic structure changes of the medial axis. This produces undesired results for meshes containing many perturbations, generating a hidden surface that is jagged and unnatural.

Unlike other methods which extract the medial axis as a non-manifold cell complex [2, 22], this method produces a two-sided manifold surface. We call this approximation the `medial surface`. This provides the advantage of allowing standard mesh operations to be performed on it. Thanks to this, noise can be combated by applying *Laplacian smoothing* to the original surface prior to being processed by Quickhull. After applying ten smoothing operations to the input mesh and ten operations to the output medial surface, the results are much cleaner as seen in Figure 4.7.



(a) (b)

Figure 4.7: The medial surface produced from the original mesh exhibits a large number of fins in comparison to the medial axis produced after applying smoothing to the mesh prior to generation.

Definition 6. *The Laplacian smoothing operation is defined as the per-vertex application of:*

$$\bar{x_i} = \frac{1}{m} \sum_{j \in N_1(i)} x_j - x_i$$

Where x_j are the 1-ring neighbors of the vertex x_i, $m = \#N_1(i)$ is the number of these neighbors, and $\bar{x_i}$ is the new position for vertex x_i [21].

Another advantage the medial surface has over a non-manifold medial axis extraction is that the topology is identical to that of the surface mesh. Edges which exist between vertices of the original surface mesh remain identical between each vertex's associated medial axis point. This allows the hidden surface to be quickly and easily stitched together from the medial axis.

In practice, the Fracture Map Algorithm is able to perform the medial surface generation in parallel with the initial five steps of the algorithm. Upon completion of

both, each fracture component is combined with its associated medial surface vertex, and triangles are formed along boundaries separating the surface vertices from the medial surface vertices. The result is a watertight mesh that extends to the approximate bisection of the given fracture segment.

4.9 Surface Stitching

The final step is to stitch together the two disjoint polyhedral surfaces of each fracture component to form a single watertight polyhedral surface mesh. This step is trivial thanks to the identical topology both surfaces share with each other. Stitching is performed by adjoining a new quad between every unshared edge and its corresponding edge in the medial surface. The result is a set of watertight polyhedral meshes that represent the original mesh after being fractured by a fracture map.

Chapter 5

Results

Results were obtained for the Fracture Map Algorithm on a number of polyhedral surface meshes with a variety of contrasting inputs. A utility application was developed to provide accessible user-interaction for the Fracture Map Algorithm. Section 5.1 introduces this application. Section 5.2 presents a number of example fractures generated via the Fracture Map Algorithm. Sections 5.4, 5.5, and 5.6 discuss the impact a number of variables have on the performance of fractures. This includes the size of a mesh, the fracture map size, and the fracture map complexity, respectively.

5.1 Implementation Details

The Fracture Map Utility Application was written in C++11 using OpenGL 2.0. The utility has support for Windows, OS X, and Linux, providing a mechanism for loading mesh files with defined texture coordinates, creating fracture maps, and performing the Fracture Map Algorithm. Figure 5.1 showcases a screenshot of the utility on Windows 7. The medial surface derivation code is based on the original work of [60], with major modifications to mesh preprocess and medial surface post-process steps.

In addition, a number of external open-source cross-platform libraries are leveraged within the codebase:

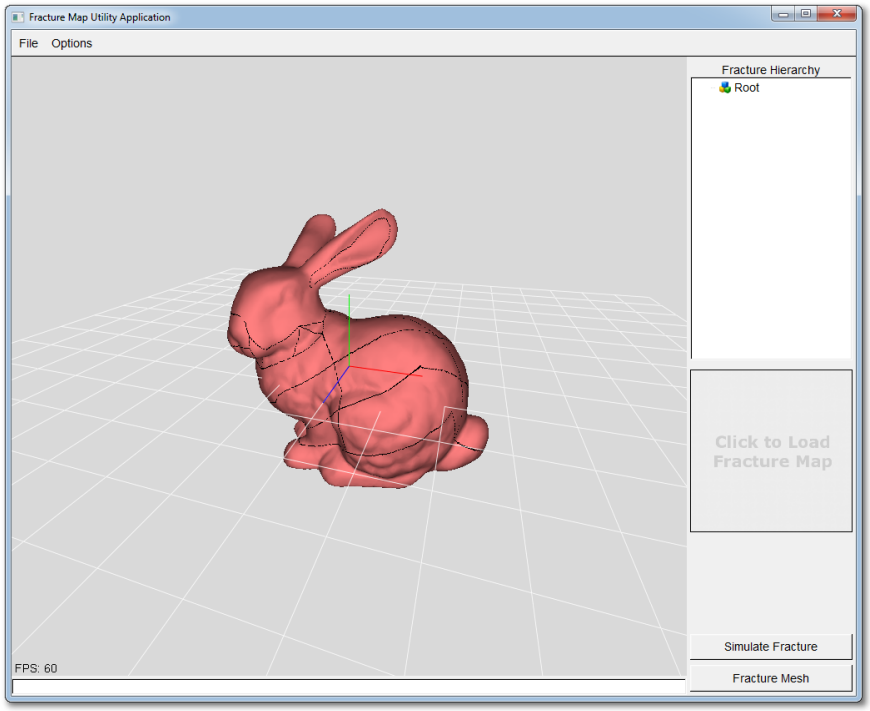


Figure 5.1: Screenshot of the Fracture Map Utility Application in Windows 7.

OpenGL Mathematics (GLM) – An OpenGL mathematics library providing consistent datatypes and functionality based on the OpenGL Shading Language (GLSL).

Developer's Image Library (DevIL) – An image library for loading, saving, and processing images of numerous formats.

Qhull – Provides efficient computation of the convex hull, Delaunay triangulation, and Voronoi diagram of a mesh via the Quickhull algorithm.

OpenGL Extension Wrangler Library (GLEW) – Provides efficient run-time mechanisms for resolving supported OpenGL extensions on a platform.

Fast Light Toolkit (FLTK) – A user interface library with OpenGL/GLUT support.

Bullet 3D Game Multiphysics Library – Provides collision detection, soft body, and

rigid body dynamics for simulations.

Within the Fracture Map Utility Application, a model's UV-space representation is projected onto a 2-dimensional raster image. This displays the texture atlas of the model and allows users to easily visualize fractures. This is shown in Figure 5.2. Alternatively, a fracture map can be viewed by overlaying its line segments on top of a pre-existing texture atlas, such as a diffuse map. This can be useful if a user wants a fracture to align with features of a diffuse map.

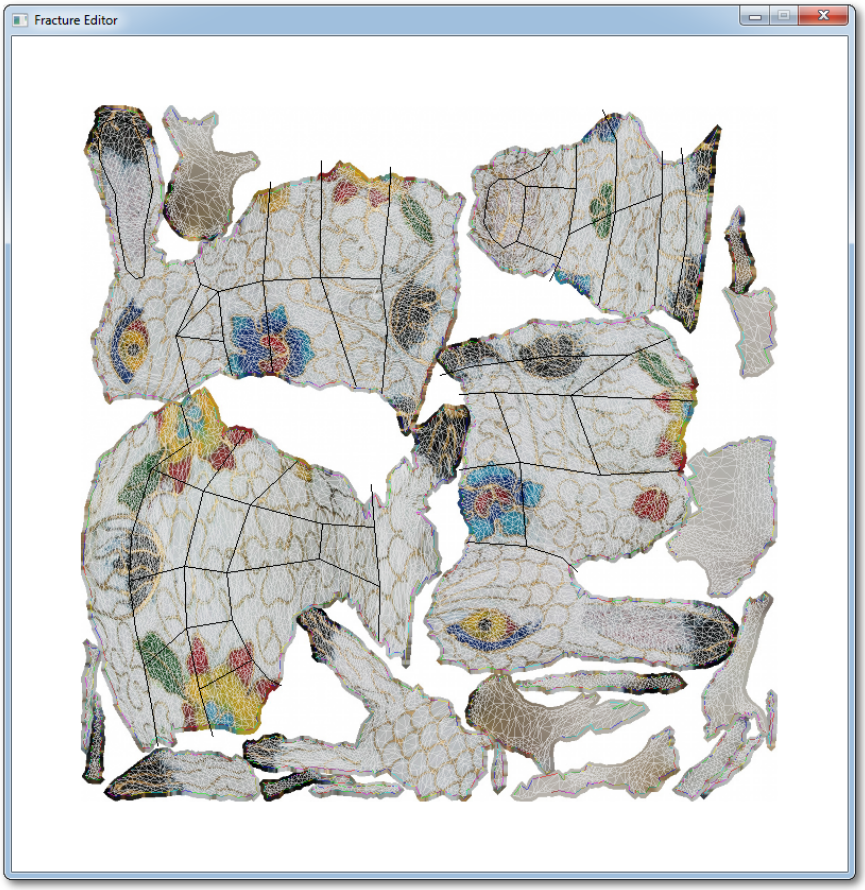


Figure 5.2: The 2D raster image generated by the Fracture Map utility application. This allows users to visualize the projection of the fracture lines onto the model's 3-dimensional surface

5.2 Example Fractures

A number of example fractures have been performed using textured models generated via the Texturemontage algorithm presented by [62]. These meshes contain texture atlases absent of overlapping geometry in UV-space. Additionally, for reference, the fracture maps have been superimposed on top of diffuse maps 1024x1024 pixels in size.

5.2.1 Buddha

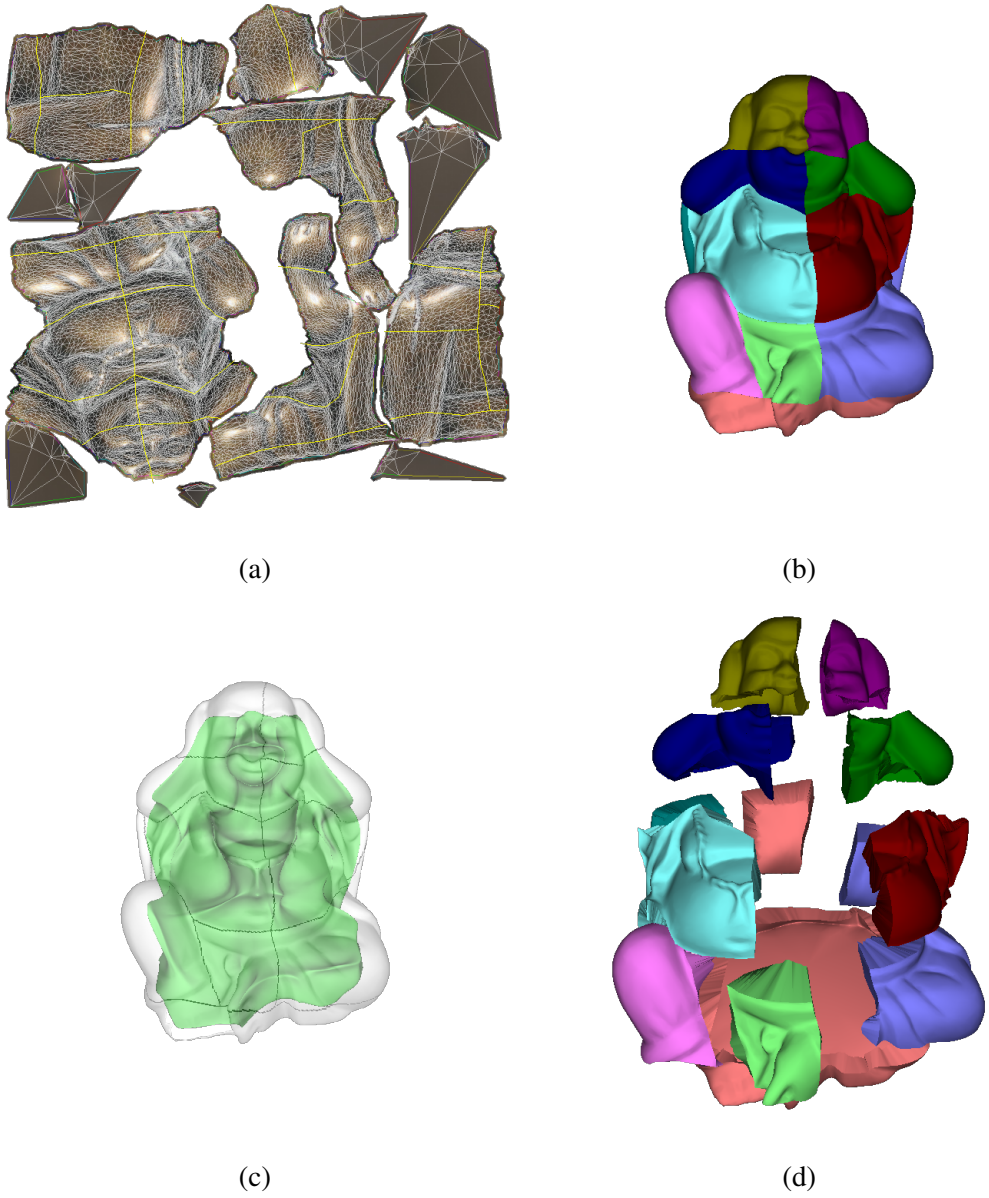
The Buddha mesh comprises of 30272 triangles with a relatively simple texture atlas. The fracture, shown in Figure 5.3 uses a fracture map that fragments the mesh into 13 components in 3.2 seconds.

5.2.2 Bunny

The bunny mesh comprises of 30338 triangles and has been fractured into 13 components. The fracture, shown in Figure 5.4, created 29 fracture components and was completed in 2.5 seconds.

5.2.3 Cat

The cat mesh comprises of 14894 triangles and has been fractured into 13 components. The fracture, shown in Figure 5.5, created 58 fracture components and was completed in 2.0 seconds.



(a) (b)

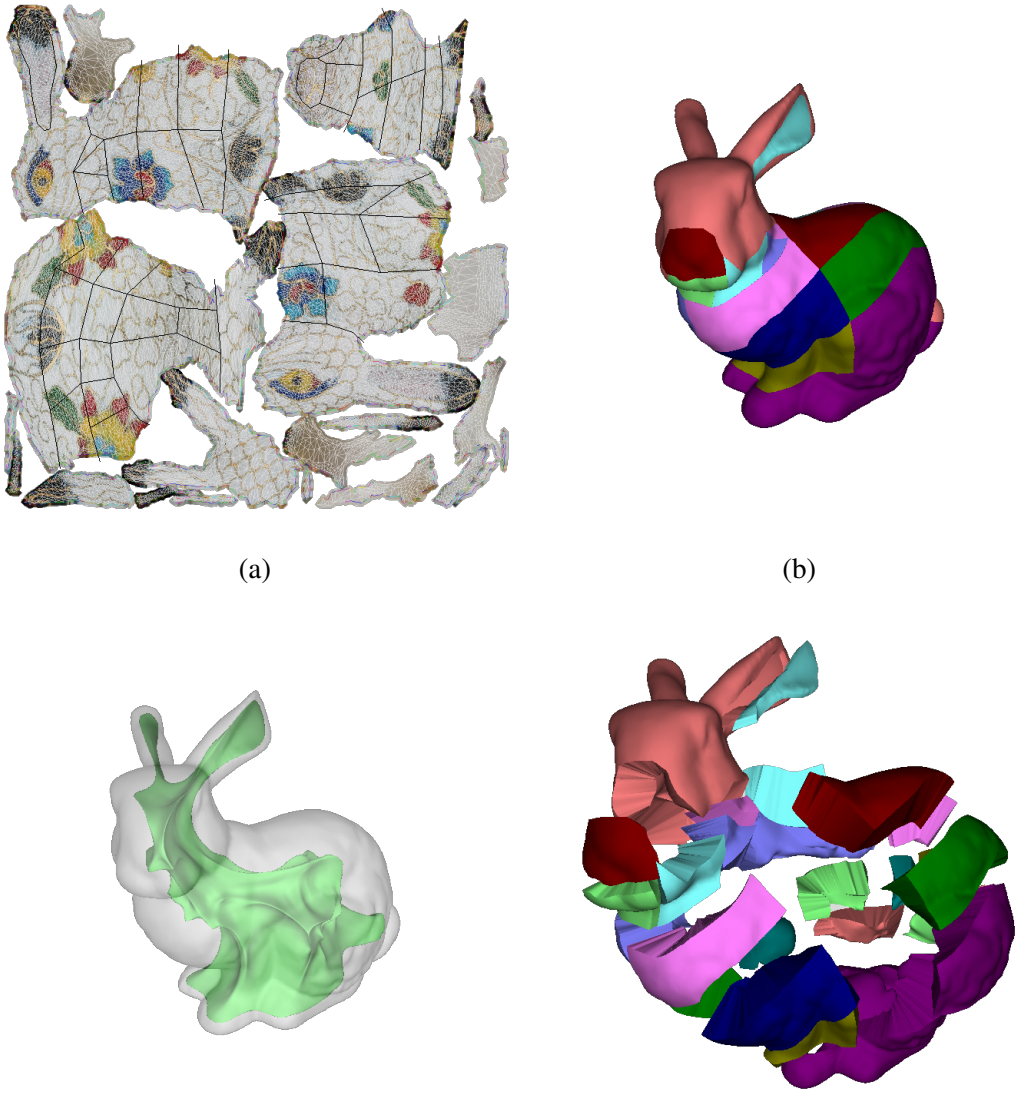
(c) (d)

Figure 5.3: The resulting fracture, shown in (b) and (d) after using the fracture map shown in (a). The medial surface generated for the hidden surface during the fracture process is shown in (c).

5.2.4 Feline

The feline mesh comprises of 41066 triangles with a relatively complex texture atlas.

The fracture, shown in Figure 5.6 uses a fracture map that fragments the mesh into 9

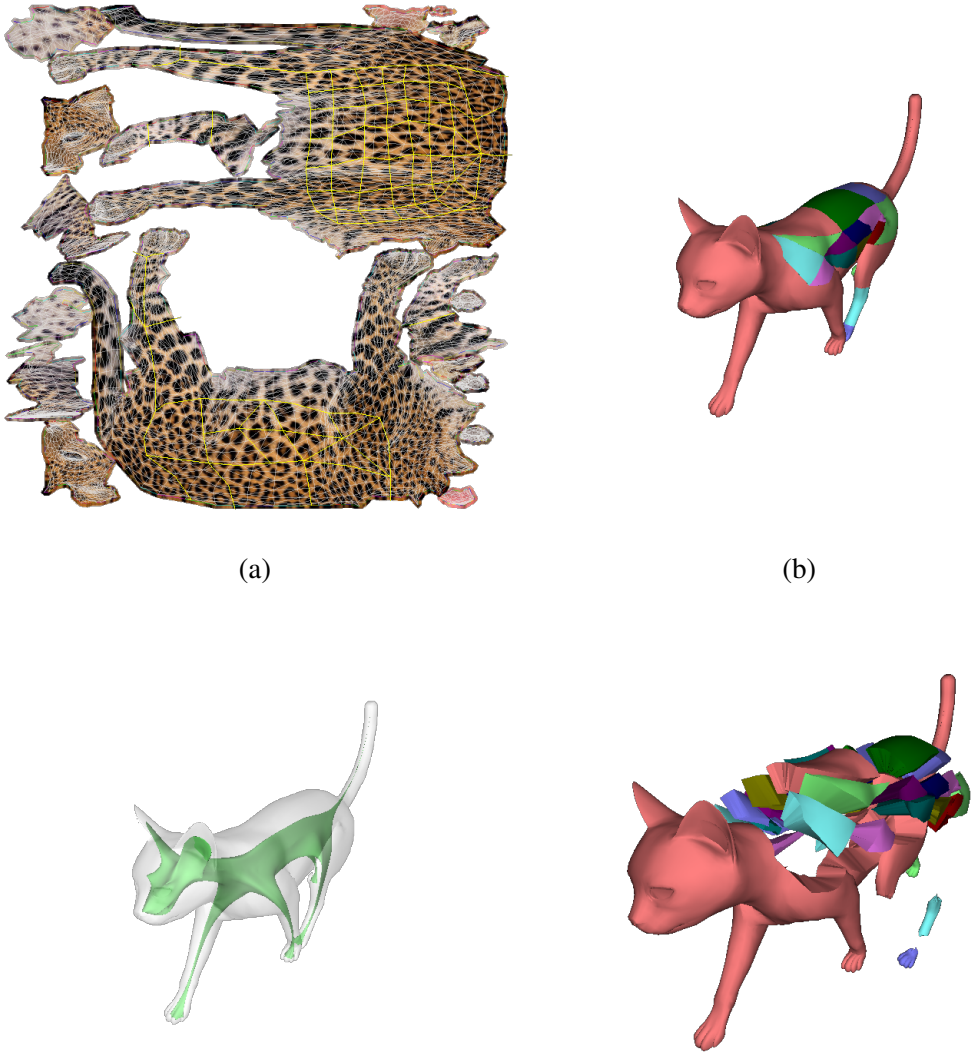


(a) (b)

(c) (d)

Figure 5.4: The resulting fracture, shown in (b) and (d) after using the fracture map shown in (a). The medial surface generated for the hidden surface during the fracture process is shown in (c).

components in 3.4 seconds.



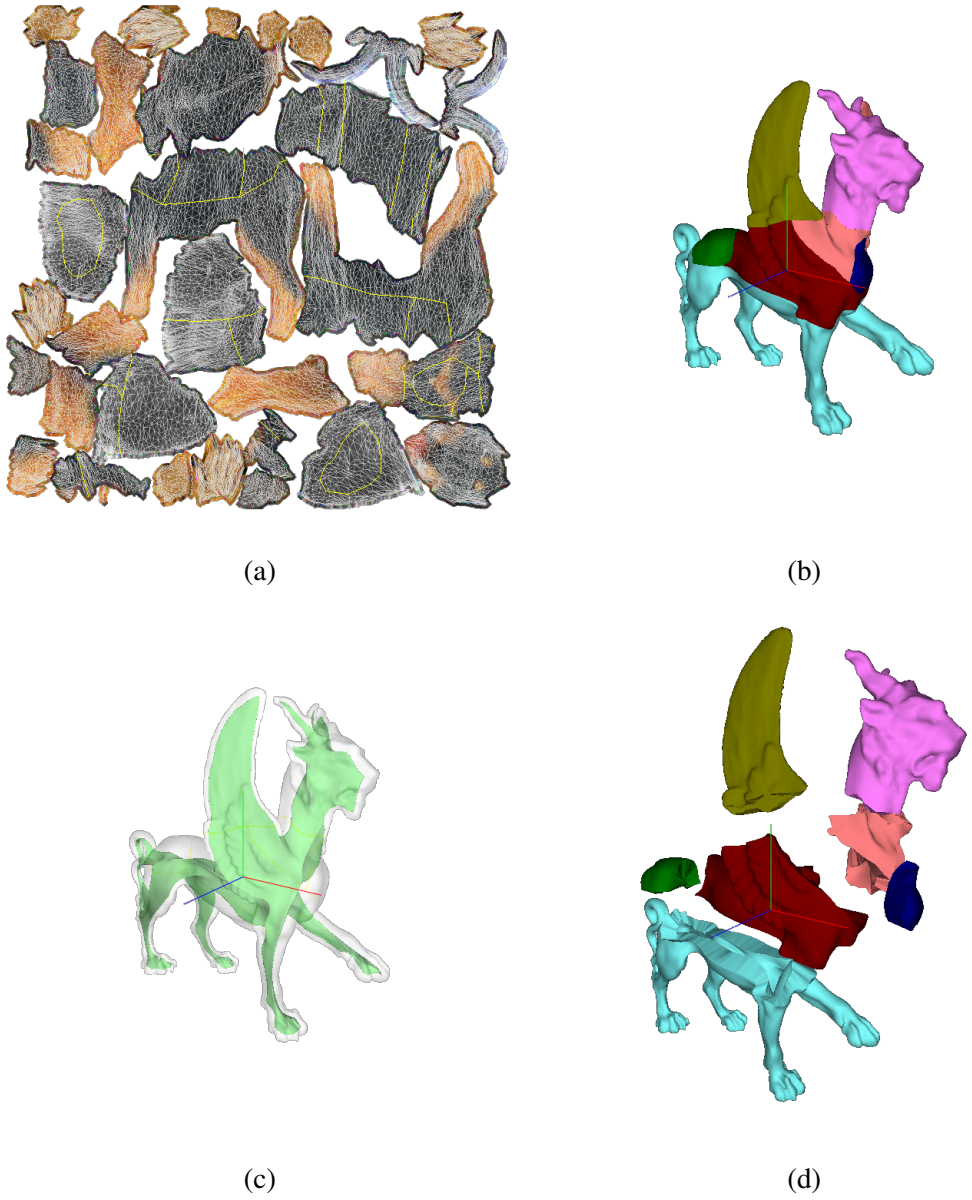
(a) (b)

(c) (d)

Figure 5.5: The resulting fracture, shown in (b) and (d) after using the fracture map shown in (a). The medial surface generated for the hidden surface during the fracture process is shown in (c).

5.3 Experimental Setup

Performance results were obtained via automated runs on a machine with a 3.214 Mhz 6 core AMD Phenom II X6 1090T processor, 4 GB of system RAM and running



(a) (b)

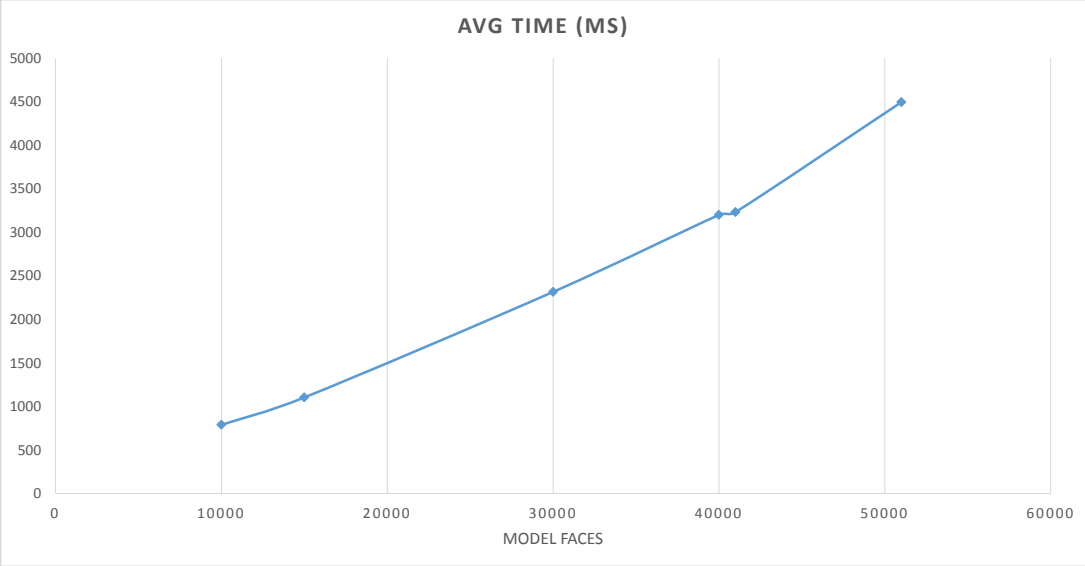
(c) (d)

Figure 5.6: The resulting fracture, shown in (b) and (d) after using the fracture map shown in (a). The medial surface generated for the hidden surface during the fracture process is shown in (c).

Windows 7. Performance data was gathered with a sample size of 100 runs on a headless version of the Fracture Map Utility Applictation, excluding graphical display overhead from the results.

5.4 Mesh Size

Many of the procedures within the Fracture Map Algorithm require iteration over each element of an input mesh. As such, the size of an input mesh is expected to have strong influence on the performance of the algorithm. Data was acquired using six meshes with increasing face counts. All fractures were performed with fracture maps of identical size and complexity. The results, shown in Table 5.1, follow this expected outcome.



Triangle Faces	Avg Elapsed Time (ms)
10,000	789.49
15,000	1104.225
30,000	2316.495
40,000	3200.435
41,000	3200.435
51,000	4495.78

Table 5.1: The elapsed time to perform a fracture as the number of triangles in an input mesh increases.

The results show steady linear growth of the elapsed time to fracture as the number of faces in the input mesh are increased. This equates to approximately 80 milliseconds per 1000 input faces.

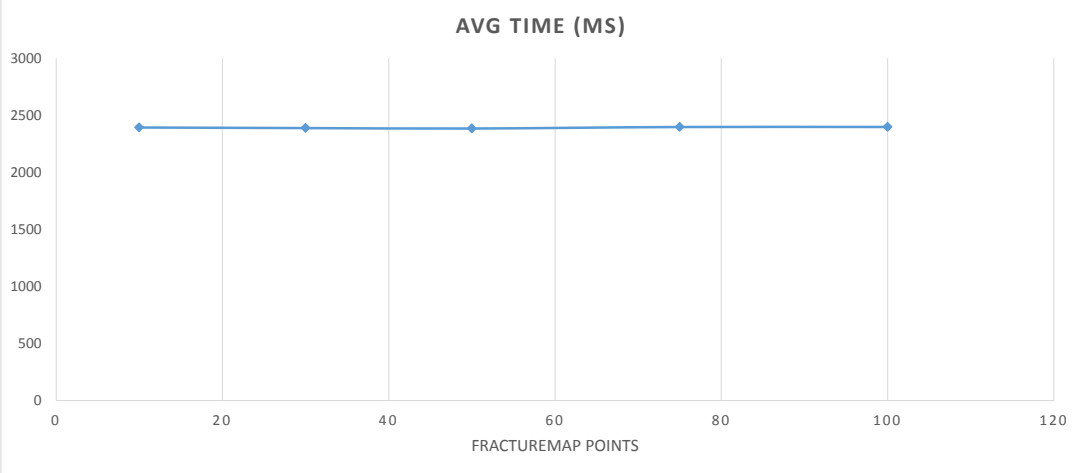
5.5 Fracture Map Size

The size of a fracture map plays an important role during the fracture edge generation step, as each half edge of an input mesh is tested for intersections against nearby fracture lines. The effect fracture map size has on performance was examined by comparing fracture times of a single input mesh with fracture maps of increasing fracture node counts. The results, shown in Table 5.2, demonstrates little impact on performance.

The stability seem can be attributed to the infrequent presence of acute fracture line density. The quadtree used to store fracture lines in a spatially efficient manner ensures the majority of half edges do not perform intersection tests over a large set of lines despite the presence of a large number of overall fracture lines.

5.6 Fracture Map Complexity

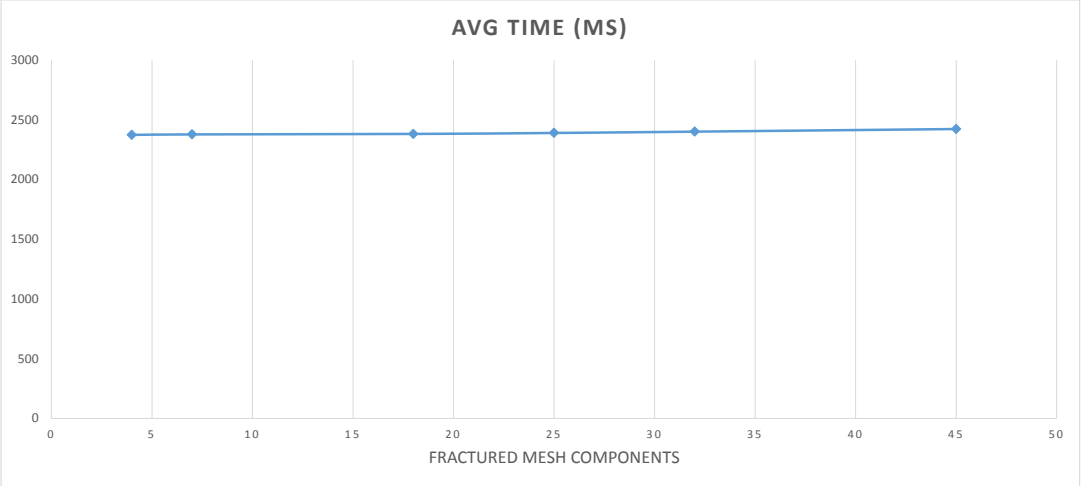
The complexity of a fracture map is considered here as the number of fracture components that result from the fracture. As such, increasing the number of fracture components resulting from a fracture map also increases the number of intersections present within the fracture map. This can have a potential impact during the fracture point matching phase, as the number of fracture point branches that require matching will be larger. The results, shown in Table 5.3, exhibit a stability similar to fracture map size.



Fracture Map Points	Avg Elapsed Time (ms)
10	2392.84
30	2387.97
50	2384.05
75	2397.1
100	2397.33

Table 5.2: The elapsed time to perform a fracture as the number of points in an input fracture map increases.

This demonstrates the efficient handling of complex fracture types within the fracture point matching phase, as the additional computation time required differs only by 49 milliseconds between 4 and 45 fracture components.



Fracture Components	Avg Elapsed Time (ms)
4	2374.29
7	2378.42
18	2382.29
25	2390.32
32	2401.3
45	2423.35

Table 5.3: The elapsed time to perform a fracture as the number of resulting fracture components from an input fracture map increases.

Chapter 6

Future Work

6.1 Multithreading

Due to the nature of the Fracture Map Algorithm, it is potentially highly parallelizable. The current implementation takes only minor advantage of an asynchronous environment by performing mesh fracturing and medial surface generation concurrently, reducing the elapsed time by approximately 22% over a synchronous implementation. However, each step of the algorithm aligns well with a producer-consumer design pattern, allowing for further performance gains to be reaped.

The fracture edge generation process produces a triangle ready to be consumed by the fracture point matching process once all three of its fracture half edges have been generated. Once both triangles adjacent to an edge have completed matching, the fracture edge can be resolved. Furthermore, a triangle can be triangulated once all three edges have been resolved. Despite the potential performance benefit, the current implementation of the Fracture Map Algorithm does not take advantage of a producer-consumer design due to time constraints.

6.2 Fracture Map Complexity

The current Fracture Map Algorithm implementation allows for fracture maps represented only by simple piecewise linear segments, but results in a cumbersome workflow when applying topological changes to a fracture map. Extending the fracture map representation to handle complex curves such as Bezier curves or NURBS would allow faster fracture map creation and more accurate representation of the desired fracture topology.

6.3 Medial Surface

The Fracture Map Algorithm is not restricted to exclusive use of Voronoi-based extraction methods, allowing the use of various other implementations to be swapped in if better results are provided.

6.3.1 Noise Reduction

One of the primary variables for the effectiveness of a medial surface to model a hidden surface is its ability to accurately model the major topological features of a mesh's geometry. The medial surface' sensitivity to small perturbations makes this difficult to achieve. The current implementation uses Laplacian smoothing to reduce noise, but this can produce artifacts within narrow regions of a mesh. The result is a medial surface that protrudes from the mesh surface. Figure 6.1 showcases an example of such an artifact.

There are a number of alternative techniques for extracting major features within a medial surface. A large subset of such techniques derive an importance value for

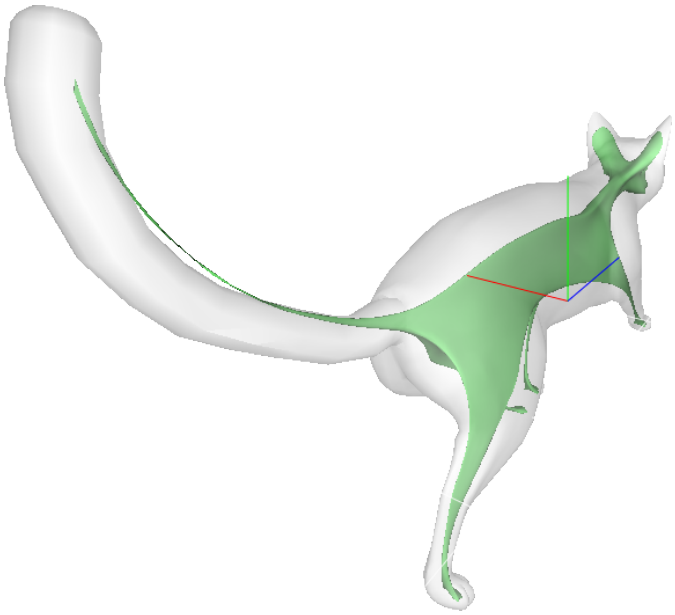


Figure 6.1: An artifact resulting from smoothing methods for noise reduction in a medial surface. The medial surface protrudes from the mesh surface.

each medial point, filtering out points deemed unnecessary based on a given threshold. [14] derives importance based on the circumradius of the closest boundary points, whereas [4] bases the value on the angle formed between the closest boundary point. [35] constructs a scale-axis transform by computing the medial axis on the union of an object's scaled medial balls followed by an inverse scaling of the resulting axis.

Such techniques can provide more consistent results across input meshes without requiring a large degree of parameter modification before converging on an acceptable medial surface.

6.3.2 Hidden Surface

The medial surface of a mesh provides a proficient representation for the hidden surface of fracture components. However, the technique results in a hollow representation of the mesh. The technique used by the Fracture Map Algorithm and extended

from [27] provides important advantages by utilizing identical topology of the original mesh. Unfortunately, the technique also gives the input mesh an appearing of having been a hollow object. For most applications this does not produce undesirable results, however, applications that require a mesh to appear entirely solid will find the results insufficient.

Alternatively, an approach that sacrifices the advantages of a manifold medial surface can use proximity information from the Voronoi diagram to generate a hidden surfaces only along contours that have been exposed by the fracturing process. The Voronoi cells adjacent to an inner Voronoi vertex indicate vertices along the surface mesh that share the same point along the medial surface. Vertices on the same side of the surface mesh can be filtered out with $(V_0 - M) \cdot (V_1 - M) \leq \eta$ where V_i is a vertex in the surface mesh associated with a Voronoi cell adjacent to the inner Voronoi vertex, M, in question.

Bibliography

[1] T. Akenine-Moller, E. Haines, and N. Hoffman. *Real-Time Rendering*. AK Peters, 3rd edition, July 2008.

[2] N. Amenta, M. Bern, and M. Kamvysselis. A new voronoi-based surface reconstruction algorithm. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '98, pages 415–421, New York, NY, USA, 1998. ACM.

[3] N. Amenta, S. Choi, and R. K. Kolluri. The power crust. In *Proceedings of the sixth ACM symposium on Solid modeling and applications*, SMA '01, pages 249–266, New York, NY, USA, 2001. ACM.

[4] N. Amenta and R. K. Kolluri. The medial axis of a union of balls. *Computational Geometry*, 20(12):25 – 37, 2001. ¡ce:title¿Selected papers from the 12th Annual Canadian Conference on¡/ce:title¿.

[5] Z. Bao, J.-M. Hong, J. Teran, and R. Fedkiw. Fracturing rigid materials. *Visualization and Computer Graphics, IEEE Transactions on*, 13(2):370 –378, march-april 2007.

[6] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, Dec. 1996.

[7] D. Bielser, P. Glardon, M. Teschner, and M. Gross. A state machine for real-time cutting of tetrahedral meshes. In *Computer Graphics and Applications, 2003. Proceedings. 11th Pacific Conference on*, pages 377–386, 2003.

[8] J. Bloomenthal. Medial-based vertex deformation. In *Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '02, pages 147–151, New York, NY, USA, 2002. ACM.

[9] J. Bloomenthal and C. Lim. Skeletal methods of shape manipulation. In *Proceedings of the International Conference on Shape Modeling and Applications*, SMI '99, pages 44–, Washington, DC, USA, 1999. IEEE Computer Society.

[10] H. Blum. A Transformation for Extracting New Descriptors of Shape. In W. Wathen-Dunn, editor, *Models for the Perception of Speech and Visual Form*, pages 362–380. MIT Press, Cambridge, 1967.

[11] H. Blum. Biological shape and visual science (part I). *Journal of Theoretical Biology*, 38:205–287, 1973.

[12] J. Brandt. Convergence and continuity criteria for discrete approximations of the continuous planar skeleton. *CVGIP: Image Understanding*, 59(1):116 – 124, 1994.

[13] C. D. Bruyns and S. Senger. Interactive cutting of 3d surface meshes. *Computers and Graphics*, 25(4):635 – 642, 2001. Intelligent Interactive Assistance and Mobile Multimedia Computing.

[14] F. Chazal and A. Lieutier. The "λ-medial axis". *Graph. Models*, 67(4):304–331, July 2005.

[15] B. Chazelle. Triangulating a Simple Polygon in Linear Time. In *IEEE Symposium on Foundations of Computer Science*, pages 220–230, 1990.

[16] S. W. Choi and H.-P. Seidel. Hyperbolic hausdorff distance for medial axis transform. *Graphical Models*, 63(5):369 – 384, 2001.

[17] T. Culver, J. Keyser, and D. Manocha. Exact computation of the medial axis of a polyhedron. *Computer Aided Geometric Design*, 21(1):65 – 98, 2004.

[18] L. d. F. Da. Multidimensional Scale Space Shape Analysis. In *Proc. International Workshop on Synthetic-Natural Hybrid Coding and Three Dimensional Imaging*, pages 214–217, Santorini, Greece, Sept. 1999.

[19] P.-E. Danielsson. Euclidean distance mapping. *Computer Graphics and Image Processing*, 14(3):227 – 248, 1980.

[20] H. Delingette, S. Cotin, and N. Ayache. A hybrid elastic model allowing real-time cutting, deformations and force-feedback for surgery training and simulation. In *Computer Animation, 1999. Proceedings*, pages 70–81, 1999.

[21] M. Desbrun, M. Meyer, P. Schröder, and A. H. Barr. Implicit fairing of irregular meshes using diffusion and curvature flow. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '99, pages 317–324, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[22] T. K. Dey and W. Zhao. Approximate medial axis as a voronoi subcomplex. In *Proceedings of the seventh ACM symposium on Solid modeling and applications*, SMA '02, pages 356–366, New York, NY, USA, 2002. ACM.

[23] M. Foskey, M. C. Lin, and D. Manocha. Efficient computation of a simplified medial axis. In *Proceedings of the eighth ACM symposium on Solid modeling and applications*, SM '03, pages 96–107, New York, NY, USA, 2003. ACM.

[24] S. Frisken-Gibson. Using linked volumes to model object collisions, deformation, cutting, carving, and joining. *Visualization and Computer Graphics, IEEE Transactions on*, 5(4):333–348, 1999.

[25] T. Funkhouser. Polygonal meshes, Fall 2002.

[26] P. S. Heckbert. Survey of texture mapping. *IEEE Comput. Graph. Appl.*, 6:56–67, November 1986.

[27] M. Hisada, A. G. Belyaev, and T. L. Kunii. A skeleton-based approach for detection of perceptually salient features on polygonal surfaces. *Computer Graphics Forum*, 21(4):689–700, 2002.

[28] H. Huy Viet, T. Kamada, and H. Tanaka. An algorithm for cutting 3d surface meshes. In *Pattern Recognition, 2006. ICPR 2006. 18th International Conference on*, volume 4, pages 762 – 765, aug. 2006.

[29] B. Kajak. *Improved algorithms for ear-clipping triangulation*. PhD thesis, 2011.

[30] S. Kim, J. Park, and J. Park. Progressive mesh cutting for real-time haptic incision simulator. In *ACM SIGGRAPH ASIA 2010 Posters*, SA '10, pages 61:1–61:1, New York, NY, USA, 2010. ACM.

[31] R. Kimmel, R. Malladi, and N. Sochen. Images as embedded maps and minimal surfaces: Movies, color, texture, and volumetric medical images. *Int. J. Comput. Vision*, 39:111–129, September 2000.

[32] Y.-K. Lai, Q.-Y. Zhou, S.-M. Hu, and R. R. Martin. Feature sensitive mesh segmentation. In *Proceedings of the 2006 ACM symposium on Solid and physical modeling*, SPM '06, pages 17–25, New York, NY, USA, 2006. ACM.

[33] L. Lam, S.-W. Lee, and C. Suen. Thinning methodologies-a comprehensive survey. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 14(9):869–885, 1992.

[34] G. H. Meisters. Polygons Have Ears. *The American Mathematical Monthly*, 82(6):648–651, June 1975.

[35] B. Miklos, J. Giesen, and M. Pauly. Discrete scale axis representations for 3d geometry. In *ACM SIGGRAPH 2010 papers*, SIGGRAPH '10, pages 101:1–101:10, New York, NY, USA, 2010. ACM.

[36] E. N. Mortensen and W. A. Barrett. Intelligent scissors for image composition. In *In Computer Graphics, SIGGRAPH Proceedings*, pages 191–198, 1995.

[37] M. Muller, M. Teschner, and M. Gross. Physically-based simulation of objects represented by surface meshes. In *Computer Graphics International, 2004. Proceedings*, pages 26 –33, june 2004.

[38] D. Nehab and H. Hoppe. Random-access rendering of general vector graphics. In *ACM SIGGRAPH Asia 2008 papers*, SIGGRAPH Asia '08, pages 135:1–135:10, New York, NY, USA, 2008. ACM.

[39] H.-W. Nienhuys and A. Frank van der Stappen. A surgery simulation supporting cuts and finite element deformation. In W. Niessen and M. Viergever, editors, *Medical Image Computing and Computer-Assisted Intervention MICCAI 2001*, volume 2208 of *Lecture Notes in Computer Science*, pages 145–152. Springer Berlin Heidelberg, 2001.

[40] J. F. O'Brien and J. K. Hodgins. Graphical modeling and animation of brittle fracture. In *Proceedings of ACM SIGGRAPH 1999*, pages 137–146. ACM Press/Addison-Wesley Publishing Co., Aug. 1999.

[41] S. Prohaska and H.-C. Hege. Fast visualization of plane-like structures in voxel data. In *Visualization, 2002. VIS 2002. IEEE*, pages 29–36, 2002.

[42] C. Pudney. Distance-ordered homotopic thinning: A skeletonization algorithm for 3d digital images. *Computer Vision and Image Understanding*, 72(3):404 – 413, 1998.

[43] Z. Qin, M. D. McCool, and C. Kaplan. Precise vector textures for real-time 3d rendering. In *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, I3D '08, pages 199–206, New York, NY, USA, 2008. ACM.

[44] I. Ragnemalm. The euclidean distance transform in arbitrary dimensions. In *Image Processing and its Applications, 1992., International Conference on*, pages 290–293, 1992.

[45] M. Ramanathan and B. Gurumoorthy. A tracing algorithm for constructing medial axis transform of 3d objects bound by free-form surfaces. In *Shape Modeling and Applications, 2005 International Conference*, pages 226 – 235, june 2005.

[46] J. M. Reddy and G. M. Turkiyyah. Computation of 3d skeletons using a generalized delaunay triangulation technique. *Computer-Aided Design*, 27(9):677 – 694, 1995.

[47] J. Rhoades, G. Turk, A. Bell, A. State, U. Neumann, and A. Varshney. Real-time procedural textures. In *Proceedings of the 1992 symposium on Interactive 3D graphics*, I3D '92, pages 95–100, New York, NY, USA, 1992. ACM.

[48] T. Saito and J. ichiro Toriwaki. A sequential thinning algorithm for three dimensional digital pictures using the euclidian distance transform. In *Proc. 9th SCIA*, pages 507–516, Uppsala, Sweden, June 1995.

[49] G. Sela, J. Subag, A. J. Lindblad, D. Albocher, S. Schein, G. Elber, and G. Turkiyya. Real-time haptic incision simulation using fem-based discontinuous free form deformation. In *In SPM 06: Proc. of the 2006 ACM symposium on Solid and physical modeling*, pages 75–84. ACM Press, 2006.

[50] E. Sherbrooke, N. Patrikalakis, and E. Brisson. An algorithm for the medial axis transform of 3d polyhedral solids. *Visualization and Computer Graphics, IEEE Transactions on*, 2(1):44–61, 1996.

[51] K. Siddiqi, S. Bouix, A. Tannenbaum, and S. Zucker. The hamilton-jacobi skeleton. In *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*, volume 2, pages 828–834 vol.2, 1999.

[52] J. Smith, A. Witkin, and D. Baraff. Fast and controllable simulation of the shattering of brittle objects. In *In Graphics Interface*, pages 27–34. Blackwell Publishing, 2000.

[53] D. Steinemann, M. A. Otaduy, and M. Gross. Fast arbitrary splitting of deforming objects. In *Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '06, pages 63–72, Aire-la-Ville, Switzerland, Switzerland, 2006. Eurographics Association.

[54] D. W. Storti, G. M. Turkiyyah, M. A. Ganter, C. T. Lim, and D. M. Stal. Skeleton-based modeling operations on solids. In *Proceedings of the fourth ACM symposium on Solid modeling and applications*, SMA '97, pages 141–154, New York, NY, USA, 1997. ACM.

[55] A. Sud, M. Foskey, and D. Manocha. Homotopy-preserving medial axis simplification. In *Proceedings of the 2005 ACM symposium on Solid and physical modeling*, SPM '05, pages 39–50, New York, NY, USA, 2005. ACM.

[56] R. Tam and W. Heidrich. Shape simplification based on the medial axis transform. In *Visualization, 2003. VIS 2003. IEEE*, pages 481–488, 2003.

[57] R. E. Tarjan and C. J. V. Wyk. An o(n log log n)-time algorithm for triangulating a simple polygon, 1988.

[58] K. C.-H. Wong, T. Y.-H. Siu, W. Tommy, Y. hang Siu, P. ann Heng, and H. Sun. Interactive volume cutting, 1998.

[59] A. G. Ye and D. M. Lewis. Procedural texture mapping on fpgas. In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, FPGA '99, pages 112–120, New York, NY, USA, 1999. ACM.

[60] S. Yoshizawa, A. G. Belyaev, and H.-P. Seidel. Free-form skeleton-driven mesh deformations. In *Proceedings of the eighth ACM symposium on Solid modeling and applications*, SM '03, pages 247–253, New York, NY, USA, 2003. ACM.

[61] W. Zhao. *Shape and medial axis approximation from samples*. PhD thesis, 2003. AAI3124420.

[62] K. Zhou, X. Wang, Y. Tong, M. Desbrun, B. Guo, and H.-Y. Shum. Texturemontage: Seamless texturing of arbitrary surfaces from multiple images. *ACM Transactions on Graphics*, 24(3):1148–1155, 2005.