Utrecht University

MASTER THESIS

ICA-3644383

REAL-TIME SIMULATION AND VISUALIZATION OF CUTTING WOUNDS

AUTHOR

ing. M.H.J. Lam

SUPERVISORS

dr. ir. J. Egges

dr. ir. A.F. van der Stappen

March 3, 2016

Abstract

Many modern computer games and medical computer simulations feature skin injuries such as cutting wounds. These fields often approach this topic in different ways, where medical simulations commonly have a minimal visualization, and most games rely heavily on artistic influence. As far as we know, no methods currently exist that combine mesh cutting simulation with skin visualization techniques in order to synthesize cutting wounds during runtime. Previous literature fails to describe a complete remeshing scheme that can handle arbitrary cuts while maintaining the topology and parameterization of an input surface mesh. Additionally, the appearance and synthesis of cutting wounds has not been sufficiently addressed. In this thesis, we explore the feasibility of constructing a damage model that simulates and visualizes natural-looking cutting wounds by generating new geometry and textures maps on the fly. We describe a cutting simulation approach which merges a cutting line into the mesh surface that is subsequently opened to reveal interior wound geometry generated at runtime. For visualizing the surface injury we generate a wound texture during runtime and propose an extension to subsurface scattering to locally discolor the skin surface around the cut. Our approach is lightweight: using a mid-range desktop computer, cuts can be created in about 45 milliseconds on average, and a typical frame is rendered in about 2.5 milliseconds. We think that our approach can be attractive for increasing the realism of cutting wounds in real-time applications without having to rely on specific artistic input.

Contents

1. Introduction

Achieving photorealism is one of the pinnacles of computer graphics, particularly in the case of rendering human characters. One the most important aspects when rendering humans is the appearance of skin. This has long been a challenging topic, and high-fidelity skin rendering has only become viable for high-performance real-time applications in recent years.

Because this development is relatively new, realistic rendering of skin is not yet ubiquitous in real-time software. In this thesis we look at how we can leverage these techniques to produce a more realistic representation of skin injuries, in particular cutting wounds. Many surgical simulators and modern video games depict cutting wounds in some manner, although they approach them in very different ways.

Surgical simulators attempt to simulate the interactivity of surgical tools with the geometry of the skin tissue as closely as possible. The visual quality of skin is often sacrificed in lieu of improving the mechanical simulation. See Figure 1.1 (left). However, we assume that in certain cases it can be valuable to account for high-fidelity visual responses as well.

Video games on the other hand are usually focused on conveying an artistic vision. Skin injuries are thus often reproduced artistically rather than physically simulated. See Figure 1.1 (right). Creating wounds this way requires a great deal of artistic input which can be both time-consuming and laborious.

We have discerned various approaches to synthesizing skin injuries in video games. In the most basic approach, an artist creates a number of skin textures with varying degrees of injury, which are swapped out during runtime when a character takes damage. A more sophisticated model applies pregenerated injuries (textures and possibly mesh modifications) to parts of the character model based on criteria determined during runtime. An even more elaborate damage model makes local modifications to a

Figure 1.1: Left: cutting a liver mesh in a surgical simulator (from [WDW13]).
Right: cutting wounds in the video game Max Payne 3 (2012).

character based on precise damage information collected during runtime, where the appropriate textures and mesh modifications are generated on the fly.

Most games visualize character injuries using either the first or second model, where artistic influence is the deciding factor for the appearance of wounds. Only in the third model are injuries synthesized during runtime, and consequently less artistically dependent and thus usable for a broader array of applications.

There is a surprising lack of academic work on the topic of skin injuries in video games and surgical simulators. Any technical details of elaborate damage models in interactive software are often not published in the public domain, making research into this topic both difficult and valuable.

In this thesis we look at the requirements and feasibility of constructing an elaborate damage model for the specific case of cutting wounds. We present a novel solution for simulating and visualizing cutting wounds in real-time. Our approach interacts directly with a three-dimensional polygon mesh and generates new geometry during runtime. This is coupled with a simple but effective texture synthesizer to create natural looking wounds on the fly.

This thesis is organized as follows. Chapter 2 summarizes previous research and provides an overview of our solution. The following three chapters describe the method in detail: Chapter 3 presents the mesh representation used in our solution, Chapter 4 describes our wound cutting simulation in detail, and Chapter 5 describes our approach of visualizing cutting wounds. In Chapter 6 we treat some implementation-specific details. Finally, Chapter 7 summarizes the results and contributions of this thesis and presents suggestions for future research.

2. Related work

Simulating and visualizing cutting wounds involves three areas of research: mesh cutting, skin rendering, and wound visualization. *Mesh cutting* techniques change the topological and geometrical structure of three-dimensional shapes. *Skin rendering* is a topic in computer graphics that is concerned which rendering photo-realistic images of human skin. We introduce the term *wound visualization* for the process of examining and reproducing the appearance of wounds in skin tissue.

2.1 Mesh cutting

Reproducing cutting wounds involves modifying polygonal or volumetric meshes using mesh cutting techniques. Mesh cutting is part of the broader research area of mesh editing that also includes topics such as deformation, simplification, remeshing, merging, and smoothing. The majority of research into mesh cutting is approached from the field of medical simulation and surgery, where three-dimensional scanning and volumetric mesh representations are commonplace. However, most software outside of the medical field exclusively uses polygonal meshes, making techniques that operate on surfaces more suitable. Not only does using surface meshes improve the ease of integration of the presented techniques into existing polygonal-based rendering engines, but it will also benefit the performance, which is an important requirement for real-time applications such as simulators and computer games.

With such a large body of work available, a number of surveys highlight the various techniques that have been developed over time. Delingette [Del98] describes a number of characteristics that surgical simulators should have: 1) reasonably accurate deformability of the manipulated object, 2) a real-time dynamic collision detection mechanism, 3) interactive computation times and low input latency, and 4) the ability to interactively cut tissue. Moreover, Delingette remarks that the visualization of the simulator is important as well: "the quality of the visual rendering greatly influences user immersion and therefore the effectiveness of the simulator." This is one of the motivators for the objective of this thesis.

A surgical simulator can have different deformation models. This field of research is also called *soft body dynamics*. Traditionally, the two main approaches are spring models and finite element methods. Although spring models have the advantage of ease of implementation, they suffer from accuracy problems causing the range of possible dynamic behaviors to be limited. Finite elements methods are accurate and can model complex dynamic behavior, but consume much computational power during runtime, and often require an extensive precomputation phase [Del98, AKR05]. Soft body dynamics are not treated in this thesis, but it should be noted that it is a prominent aspect of surgical cutting simulators.

In a survey by Bruyns et al. [BSM+02] mesh cutting techniques are classified based on a number of criteria: definition of the cut path, creation of new primitives (polygons), and remeshing. They describe that a cut path can be defined by either placing points on the surface of the mesh, by moving a predefined template shape through the mesh, or by directly tracing the path onto the mesh. In both the first and third method successive planes are constructed between either the selected points or between interval points [BS01, BSM+02]; the only difference between these methods is how the cut path is created. The second method requires specific collision detection between the template shape and the mesh primitives.

There are a number of ways to deal with primitive creation and remeshing after creating a new cut. A common strategy is to keep the creation of new primitives to a minimum, because the running time

of dynamic solvers for physics simulation is influenced by the number of primitives in a mesh. This requirement is especially critical when finite element methods are used [MV00]. On the other hand, the topological quality of the mesh influences the stability of dynamic solvers. It is thus important to achieve a balance between the number of primitives and the topological quality of a mesh.

Most works focus on keeping the number of new primitives minimal by creating new vertices at tool-primitive intersection points and then performing additional steps to maintain mesh quality; this process is called *remeshing*. Another approach is to reposition neighboring nodes onto the cutting path in order to maintain the same number of primitives as before the cut; this is called *node-snapping* [SHS01, NvdS01]. Finally, if the quality of the mesh is an extremely important requirement, near-minimal methods can be used to maintain topological symmetry around the cutting path, such as in [BG00].

Local remeshing can occur either during tool-primitive intersection, or when the tool leaves a primitive or changes direction. The first approach is also called *progressive cutting* [MK00], which means that primitives are remeshed along the cut path as soon as the cutting tool intersects with them. In the second, *non-progressive cutting* approach there is a delay before primitives are being remeshed, which may be noticeable depending on the relative size of the primitives.

Various works provide a description of the workflow and features that a common mesh cutting simulator should have [Del98, BSM+02, BG00, BMG99, WZW+05, SSL+07]. Generally, the simulation loop consists of five main stages; see Figure 2.1. The response to *input polling* depends on the definition of the cut path as described above. Whenever a change in input (such as tool motion) is detected, the *collision detection* stage becomes active. This stage is responsible for finding out whether or not the cutting tool intersects with an object, and if so with which primitive. In the case of a positive collision detection the *collision response* stage will modify the topology and geometry of the mesh. Geometry changes are followed by *mesh deformation*, a physically-feasible response to the cutting action. Finally, the *rendering* stage presents the results to the screen. Both the input polling stage and the rendering stage are preferably executing asynchronously from the simulation, so that the system remains interactive at all times.

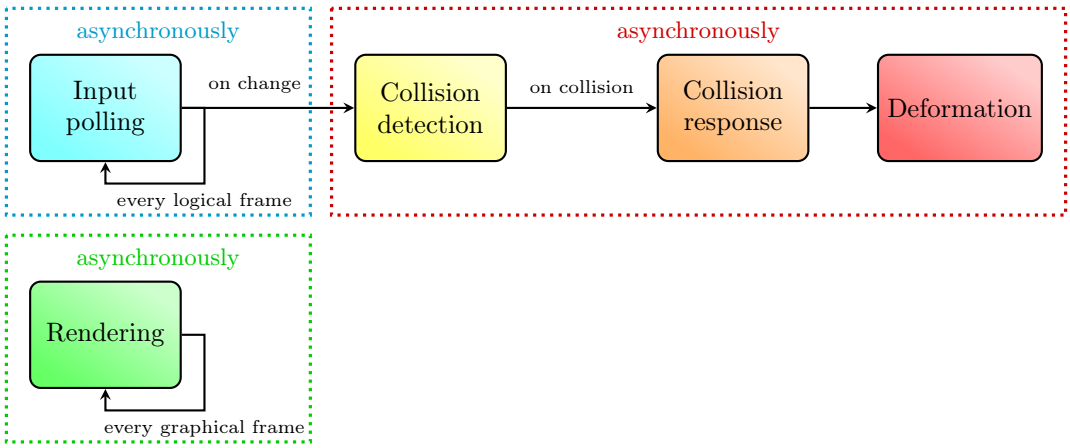


Figure 2.1: Stages of a mesh cutting simulator.

Aside from presenting a survey, Bruyns et al. [BSM+02] also describe a cutting scheme for progressive cutting. In this method tool-primitive collisions are performed by first constructing a sweep surface between consecutive positions of the tool edge. This sweep surface is essentially a plane, bounded by the length of the cutting tool and the positions of two consecutive intervals of the tool motion. Using such a plane allows collision detection to work even for multilayered objects. Testing the sweep surface against a primitive produces edge and face intersection points, where new nodes are connected with newly created edges and faces, thus replacing the original primitive. However, much of the information stored is redundant for non-progressive cutting methods, and the remeshing scheme only handles face-edge and edge-edge intersections. Furthermore, cuts leave the interior of the surface mesh exposed, and thus the method is only useful for a particular set of cases (such as for multi-layered meshes).

Zhang et al. [ZPD02] propose a simple progressive cutting technique for surface models that subdivides primitives and generates an interior structure along a selected cut path. Separate subdivision algorithms are devised for end triangles and midway triangles, where the end triangles only have one edge intersection with the cut path and midway triangles have two. End triangles are subdivided into four new triangles and have an interior edge coinciding with the cut path through the triangle. Additionally, two new nodes are created at location of the intersected edge that are initially coincident but are separated later. This separation leaves a cavity which is filled by four new interior triangles that are oriented perpendicularly to the surface, and their depth is determined by the penetration depth of the tool tip. Midway triangles have a similar algorithm. Further, a progressive cutting algorithm is presented that traces the tool inside triangles by performing temporary subdivisions and constructing groove triangles on the fly. New geometry is made permanent if the cut path stops in the current triangle, or will be replaced by midway geometry if the cut path moves on to another triangle. A method to join two progressive cuts is presented as well, although it is only capable of connecting end triangles. Like the previous method, no complete remeshing scheme nor a description of the mesh representation is presented, making it difficult to reproduce the method. Additionally, the width of the cut opening is based on arbitrary spring forces instead of using physical quantities.

Nienhuys and Van der Stappen [NvdS04] describe a cutting technique that aims to produce a well-shaped mesh that produces as few new elements as possible by enforcing primitives not to have large angles or short edges. This is realized by measuring element quality during remeshing and applying edge-flips to create a valid Delaunay triangulation. Any cutting tool motion moves an attached active node, and after each motion the incident triangles are locally remeshed. Remeshing consists of removal of nodes that are too close to each other, edge flips, and creation of nodes that approximate the cut path. In this way, nodes are removed in front of the tool and inserted behind it. The primary characteristic of this method is that the resolution of the mesh does not change because new nodes always lie on a line that connects existing mesh nodes. This can be an advantage whenever the number of primitives should be kept to a minimum, but can also be a disadvantage because the accuracy of the cut path is determined by the resolution of the starting mesh.

Lim et al. [LJD07] present a progressive cutting algorithm that is based on node snapping. At first contact the nearest node is snapped to the initial collision point between the tool and the object. As the cut progresses the nodes nearest to a tool-edge intersection point are snapped to the cut path, thereby approximating the motion of the tool. Next, each node that lies on the cut path is split into two nodes, displaced perpendicularly to the cut by equal amounts in opposite directions. Finally, a *cutting gutter* is created to give the illusion of volumetric cutting, as in Zhang et al. [ZPD02]. They also describe a physically-based solution for determining the width of the cut opening. A major problem with the node-snapping technique is that the resolution of a cut depends on the level of refinement of the mesh around the cut path. To alleviate this problem, a local subdivision algorithm is applied that subdivides triangles within a certain radius of influence with different levels of detail. This local refinement technique allows the cut path to be more accurately represented, but increases the memory footprint, mitigating the main advantage of using a node snapping method in the first place.

2.2 Skin rendering

Realistically rendering skin is both a traditional and an ongoing research area in computer graphics. In physically based rendering, radiative transfer theory [Cha60] and the rendering equation [Kaj86] provide formulae to compute the appearance of a material. For skin, light propagation consists of surface reflectance, subsurface reflectance, and transmittance [KB04]. Only about 4% to 7% of the incident light is reflected directly at the surface [INN05]. The remaining fraction of incident light particles enter the skin, undergo scattering and absorption effects, and are either fully absorbed or get scattered outwards. Light particles scattered back towards the incident surface account for subsurface reflectance, and those that scatter forward and exit the surface at the other side account for transmittance. Figure 2.2 shows how scattering can produce subsurface reflectance and transmittance at the top and bottom interfaces.

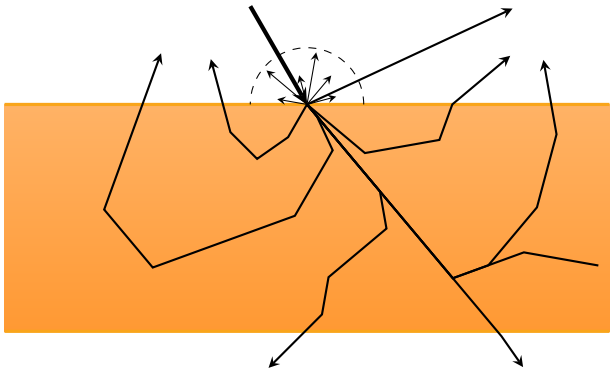


Figure 2.2: Subsurface scattering in a material.

Although subsurface light transport can be accurately simulated by solving the full radiative transfer equation, this is computationally extremely expensive, even for offline rendering [JMLH01]. Hanrahan and Krueger [HK93] presented an analytic first-order approximation to the rendering equation in the form of a bidirectional reflectance distribution function (BRDF), as well as a Monte Carlo method for simulating radiative transfer in layered media. However, this approximation gives very crude results and is still computationally expensive. Some practical models that simulate light transport in tissue include Kubelka-Munk theory, the adding-doubling method, discrete ordinates, path integrals, and especially the diffusion approximation equation [INN05].

Jensen et al. [JMLH01] describe how the diffusion approximation equation can be derived from radiative transfer theory. Although this equation does not generally have an analytic solution, it can be approximated with the dipole method as proposed by Eason et al. [EVNT78] and Farrell et al. [FPW92]. This leads to the formulation of a diffuse bidirectional surface scattering reflectance distribution function (BSSRDF) [NRH+77] that can be used to achieve much faster offline rendering speeds. Jensen's final BSSRDF model is a sum of the diffusion approximation and the single scattering term computed by Hanrahan and Krueger. Donner and Jensen later extend the dipole approximation to the multipole model that can more accurately capture the effects of thin translucent slabs and multi-layered materials [DJ05].

Unfortunately, the diffusion approximation presented by Donner and Jensen is still on the order of several seconds per frame, which means that it is too computationally expensive to use for real-time rendering. However, both Green [Gre04] and Gosselin et al. [GSM04, Gos04] describe methods to simulate diffusion in texture space based on the work by Borshukov and Lewis [BL03]. They proposed to simply collect the incident light on a 3D model into an irradiance texture and then to blur it by convolving it one or multiple times with a rapid falloff kernel whose width varies per color channel. Although fast, the downside of these approaches is that they use ad hoc parameters that are not directly based on the properties of a translucent material, and additionally only use a single Gaussian kernel, resulting in an unrealistic look.

D'Eon et al. [dL07, dLE07] resolve this issue by starting at Donner and Jensen's multipole model and making the key observation that a diffusion profile can be approximated well by a weighted sum of Gaussians. This allows irradiance diffusion to be more efficiently evaluated while still accounting for the physical properties of a material. The Gaussian sum is separated into a hierarchy of irradiance diffusion texture maps, where each texture represents a Gaussian blur. D'Eon notes that six Gaussians can accurately represent the three-layer model for skin presented in [DJ05]. The series of irradiance textures is combined in a final pass to approximate the convolution of irradiance by the original diffusion profile with real-time performance. To account for transmittance effects in thin regions of tissue, d'Eon et al. use modified translucent shadow maps, which were first proposed by Dachsbacher and Stamminger [DS03].

Recent work by Jimenez et al. [JSG09, JG10, JWSG10, JJG12a] translate the texture-space subsurface scattering simulation by d'Eon into a screen-space diffusion approximation that is applied as a post-processing filter. This approach ensures that real-time frame rates are maintained when render-

ing multiple models by alleviating the performance deterioration caused by having to process more and more textures. The idea is to directly apply the diffusion profiles to an already rendered diffuse image in screen-space, instead of to an irradiance texture [JSG09]. Jimenez et al. also describe a method for adding transmittance effects to the screen-space method [JWSG10].

Although there have been extensive investigations into the appearance of skin (such as [INN05]), research about the influence of external factors on skin rendering is scarce at best. There appears to be some research about skin color change due to emotion (and deformation) [JSB+10], but we were unable to locate any works in the field of computer graphics that treat the appearance of skin due to injuries. As such, it is currently unknown how subsurface scattering methods can be leveraged for rendering injured skin.

2.3 Wound visualization

Aside from occasional remarks in academic sources of medical or forensic science, very little information on the topic of the appearance of wounds is available. This is because medical sciences are primarily interested in the mechanics and processes associated with the healing of injured skin. The appearance of wounds is only relevant when its properties can be used for diagnosis. On the other hand, forensic scientists concern themselves with collecting facts from perceptible clues, and although a basic description of skin injuries can be helpful in assessing a crime, a full explanation of how skin visually reacts to trauma is usually not relevant.

In this thesis we are mainly concerned with cutting wounds, so understanding their appearance helps to reproduce them graphically. A cutting wound can be either inflicted or accidental and is always the result of sharp force trauma. A cut made with a sharp surgical tool is also called an incision, but this term does not apply to cuts made with other sharp instruments. Incised wounds are often erroneously used interchangeably with lacerations, but are distinct due to the absence of thin bridges of tissue within the wound, and show little to no abrasion at the edges of the cut [SSF06, BM10]. Another visual aspect of cuts is that the skin surrounding the wound turns red. The medical term for this redness is erythema, and is caused by rupture of small venules and capillaries or by the inflow of blood cells to start the healing process [SBJ07]. Erythema is usually diffuse and does not have a pattern [BM10].

Research in the fields of computer graphics and game technology on the topic of wound visualization or wound models is very sparse. Two related sources that give insight into a practical wound model are by Grimes and Vlachos [Gri10, Vla10]. These two works describe a technique that uses a dynamically placeable cull volume to denote the extent of an open wound, from which an interior model of flesh and bones is visible. A projected texture is used to visualize blood spatter and laceration around the edges of the opening. Although this technique is able to create natural-looking injuries that cut away a part of a mesh, their applicability is limited due to requiring an interior mesh behind the skin surface. Additionally, it only allows cutting with ellipsoids or disks, and is thus not appropriate for small cuts.

Lee et al. present a facial wound synthesis system [LC10, LLC11]. The first of these works describes the practical implications of projecting a wound texture to a three dimensional model. The second work presents techniques for generating a facial depth map and a map for measuring the spatially-varying depth of an input wound image. These two texture maps are then used to modify (the depth of) the local geometry of the facial mesh when applying a wound image to a selected area. Their technique allows a user to interactively choose the location of the wound image which is then applied to the target mesh in real time. However, they use pregenerated wound textures and have taken no action to improve wound blending with the original skin color, making the wounds look very artificial.

2.4 Motivation and overview

We have described three areas of research that have thus far not often been combined in the realms of computer graphics and game technology. To our knowledge no previous solution exists that accounts for both simulation and visualization of cutting wounds in equal fashion. In this thesis we will present a number of techniques for the purpose of simulating and visualizing cutting wounds on skin surface meshes. We also propose a novel mesh representation to support these techniques.

Figure 2.3 gives an overview of the components and stages that our solution consists of. Note the similarities between this diagram and Figure 2.1.

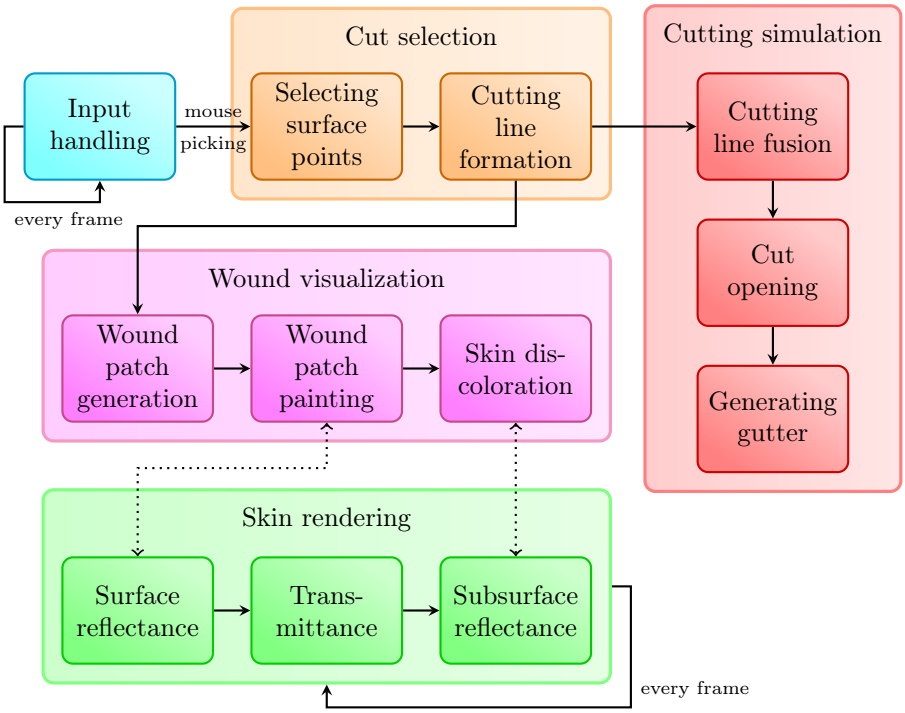


Figure 2.3: Overview of the stages of our method. Solid arrows indicate work-flow between stages and dotted arrows between stages indicate that they indirectly influence each other.

In our method, we allow a user to directly pick two points on the surface of a mesh to define where a new cut should be generated. The input handling stage, which is performed on every frame, triggers collision handling whenever such a mouse picking event occurs. Cut selection combines the collision detection and part of the collision response stages shown in Figure 2.1. A mouse picking event is translated into a ray-surface intersection test to find a surface point. When two valid selections are made on the same mesh, a chain of line segments is formed between the two points on the mesh surface that denotes the cutting wound. The cut selection stage is treated in Section 4.1.

After the cutting line formation stage there are two courses of action: cutting simulation and wound visualization. Cutting simulation merges the cutting line with the mesh and is consequently opened to create a mesh cavity. This is then filled up by newly generated cutting gutter geometry. Cutting line fusion is described in Section 4.2, cut opening in Section 4.3.1, and generating the cutting gutter is discussed in Section 4.3.2.

Our cutting simulation is based on three works. Like Bruyns et al. [BSM+02] we use a sweep surface to define the cut selection and to determine where the cut intersects the mesh surface. Our simulation is augmented by generating interior geometry that represents the groove or gutter of a cutting wound as described in Zhang et al. [ZPD02]. We improve upon these two methods by describing a remeshing scheme that accounts for all possible cutting line segment configurations and handles each case in a

consistent manner. In this thesis, we pay special attention to describing the exact mesh operations performed at every step, something these works neglect to mention. For this purpose we also propose a novel mesh representation (Chapter 3). Finally, we use the definition of the cut opening displacement from Lim et al. [LJD07] to determine the width of a cut.

Note that there are a number of differences between our cutting simulation and other surface-based mesh cutting approaches. Firstly, we do not represent the tool directly. This means that cuts do not necessarily have to be defined by a tool motion. Secondly, our current method does not support progressive cutting, although this is something that could be added in the future. Thirdly, because we assume that creation of new primitives is acceptable, we have not used the node snapping methods as described in [NvdS04] and [LJD07].

After forming the cutting line the other course of action is executing the wound visualization stage. Because this stage is reliant on the pre-modified mesh, it is actually executed before the cutting simulation stage. We use the term wound visualization to change the appearance of the target model around a cut selection. This includes synthesizing and drawing a wound texture onto the model during runtime, as well as locally changing the color of the skin to simulate skin erythema. These novel techniques are presented in Sections 5.3 and 5.4, respectively.

Because wound visualization changes the appearance of skin, it is closely related to the skin rendering stage. This stage, executed on every frame, consists of surface reflectance, subsurface reflectance, and subsurface transmittance. We use the high-performance screen-space skin rendering methods by Jimenez et al. [JWSG10, JJG12a] that are easily integrated as a post-process. Surface reflectance is influenced by wound patch painting due to it modifying the surface color of a mesh. Similarly, skin discoloration changes the appearance of subsurface reflectance. We modify the subsurface reflectance method by Jimenez et al. to account for this local skin discoloration. This process is described in Section 5.4, while the full skin rendering stage is described in Appendix B.

3. Mesh representation

Before we can discuss the mesh cutting process in detail we must first describe our mesh representation. The mesh representation is an essential part of the mesh cutting simulation that is often not treated in the discussion of cutting techniques. However, remeshing and geometry generation techniques – as well as rendering routines – are highly dependent on the exact definition of the mesh representation. One of our main goals is to define a representation that accounts for the interaction between mesh cutting and rendering, an issue most traditional mesh representations neglect to account for. In this chapter we propose a novel mesh representation that is specifically tailored toward simulating cutting wounds.

3.1 Polygon meshes

Making modifications to three-dimensional shapes requires proper organization of its data. At a minimum, a shape consists a collection of points, i.e. a point cloud in $\mathbb{R}^3$ [PGK02]. Such a three-dimensional shape can either be represented as a volume or by a boundary surface [BKP+10].

Because we approach this problem from the field of game technology, where polygon meshes see much more widespread usage than any other type of representation, we have opted to use surface meshes for efficient simulation of cutting wounds on three-dimensional shapes.

Polygon meshes describe a continuous boundary that envelops the three-dimensional shape. These boundaries can be approximated parametrically by connecting the set of points into a polygon mesh. Polygon meshes are piecewise linear approximations of smooth shapes. They generally have a simple but versatile data structure that allows them to be used for almost any type of model, and can be transformed and rendered efficiently [BKP+10].

A polygon mesh $\mathcal{M}$ consists of a geometrical and a topological component contained in three sets of mesh elements: the vertices $\mathcal{V}$, the edges $\mathcal{E}$, and the faces $\mathcal{F}$ [BKP+10]. Although mesh faces can consist of any n-sided polygon (where $n \geq 3$), for this thesis we will assume that the surface has been triangulated. This assumption may appear to be limiting, but there are a number of benefits to exclusively using triangular faces:

1. Graphics hardware is highly optimized to process and render triangles [Mic16, SSKLK13]. Likewise, many geometric algorithms can exploit the fact that all faces are triangular and convex, allowing for simpler and more efficient procedures.
2. Triangles are the smallest possible polygon that can represent a face. This allows the discrete mesh to approximate the continuous surface as closely as possible – which would, for example, not always be the case with quadrilateral faces.
3. Any polygon mesh can be transformed into a triangular mesh. However, transforming an n-polygon mesh to an m-polygon mesh (with $n,\ m > 3$ and $m \neq n$) is a much harder problem and not generally solvable [dBCvKO08].

3.1.1 Mesh elements

A geometric embedding of a polygon mesh into $\mathbb{R}^3$ is specified by associating a Cartesian coordinate to each vertex $v_i \in \mathcal{V}$ [BKP+10]:

$$\mathcal{V} = \{v_1, \ldots, v_V\}, \quad v_i \in \mathbb{R}^3.$$

This set of vertex coordinates is sufficient to extract geometrical properties about the mesh. Examples of geometric queries are finding the Euclidean distance between two vertices, finding the area or volume between a set of vertices, and finding the intersection point between the mesh and a given geometric primitive (e.g. ray, sphere, triangle).

The topological component of the mesh representation encodes the connectivity between the mesh elements. Connectivity can be mathematically described by a graph structure $G = (V, E)$ that is embedded in $\mathbb{R}^3$ [Cor09]. For a triangle mesh the topological space is a simplicial complex. Edge elements are defined as connectors between two vertices that are considered to be adjacent [BKP+10]:

$$\mathcal{E} = \{e_1, \ldots, e_E\}, \quad e_i \in \mathcal{V} \times \mathcal{V}.$$

A face is a closed set of $n \geq 3$ edges, which can be equivalently expressed as a closed set of $n \geq 3$ vertices. A polygon is defined as a set of faces whose vertices are coplanar. Whether polygons and faces coincide partially depends on the number of vertices incident to a face. In a triangular mesh, every face has exactly $n = 3$ vertices, which allows us to define the set of faces as follows [BKP+10]:

$$\mathcal{F} = \{f_1, \ldots, f_F\}, \quad v_i \in \mathcal{V} \times \mathcal{V} \times \mathcal{V}.$$

Faces that share a vertex or edge are called adjacent, incident, or neighbors.

Mesh elements can be optionally augmented with attribute information: specific data required for rendering or simulation. Vertex definitions often include attributes for rendering such as normal vectors and texture coordinates. Note that both geometric information and attribute data can be stored separately from the topology.

3.1.2 Polygon mesh representations

Mesh representations provide an unambiguous approach to accessing the geometry, topology, and attribute information of a mesh. It allows operations to be defined consistently, making it easier to maintain a well-formed mesh that is manifold, triangular, and closed [BKP+10]. The exact representation that is used greatly influences the memory consumption and efficiency of the geometrical and topological algorithms performed on a mesh.

Whether a specific representation is suitable largely depends on the elementary operations that the mesh must efficiently support. Other important factors include memory requirements, algorithmic complexity, mesh maintainability, and other requirements or assumptions. A variety of algorithmic or topological considerations may be relevant as well.

Because geometry and attributes can be stored separately from the topological information, representations mainly differ in how they describe connectivity between mesh elements. Which adjacency relationships to explicitly store depends on which traversal operations are most frequently used. Less frequently used adjacencies can then be stored implicitly. In general, the more explicit relations between vertices, edges, and faces are represented, the faster meshing operations can be executed, and the higher the storage requirements become [FvDFH95].

Often, there is a disparity between the data required for simulation and the data required for rendering. Hardware rendering accelerators expect as input simple and compact triangular mesh data structures such as triangle lists, triangle strips, or triangle fans that include very little adjacency information. Conversely, many mesh processing functions need fast access to topological information which requires more complex data structures that have higher memory footprints.

Although it is not uncommon to design a data structure to specifically support a specific mesh algorithm, there are a number of data structures common to several algorithms in geometry processing [BKP+10]. For convenience we have included some of the most common mesh representations in Appendix A.

3.2 A novel mesh representation for simulating and visualizing cutting wounds

Because we have found that none of the traditional mesh representations is directly suitable for our situation, we propose a novel mesh representation that is specifically tailored for simulating and visualizing cutting wounds on surface meshes.

In order to come to a definition for our new data structure we have to consider the assumptions and requirements of the mesh. We assume the mesh described by the representation is a triangular closed orientable 2-manifold mesh, that can be either convex or concave, and either regular or irregular (see Botsch et al. [BKP+10] for an explanation of these terms).

The primary requirement for our data structure is that it must support dynamic changes to its geometry and topology during runtime. Specifically, we want to support operations relating to the simulation of cutting wounds. Additionally, we wish to have a tight coupling between the topological mesh and what we call the *renderable mesh* to reduce the overhead between simulation and visualization as much as possible.

In this section we will first describe the requirements of our mesh representation, followed by an evaluation of traditional representations with respect to these requirements. We then present our mesh representation formally.

3.2.1 Requirements

For the purpose of simulating and visualizing cutting wounds a mesh representation is needed that supports efficient geometrical modification, topological editing, and real-time rendering. This section provides an overview of the requirements that our mesh representation should satisfy.

3.2.1.1 Simulation requirements

Simulating cutting wounds on a surface mesh consists of a few high-level procedures: testing intersections between geometric primitives, performing local mesh refinement, and carving an incision into the surface of the mesh. The exact algorithms associated with these concepts are presented in the next chapter.

In general, polygon mesh processing algorithms can be reduced to a number of low-level operations by deriving them from the Euler operations defined by Baumgart [Bau72]:

1. Inserting and removing vertices, edges and faces.
2. Vertex merging: merging several vertices into a single one.
3. Vertex duplication: creating a copy of a vertex.
4. Edge contraction: removing an edge and merging its incident vertices.
5. Edge subdivision: inserting a vertex on an edge, splitting it in two.
6. Face subdivision: inserting an edge or vertex that splits a face.

For our mesh cutting simulation we will make extensive use of insertion and removal operations, and edge and face subdivision operations. Vertex removal, vertex merging, and edge contraction procedures are never used. Note that this means that the complexity of the mesh always increases due to the creation of a new cut.

We impose a number of conditions on the mesh modification operations. First, mesh consistency must be maintained by ensuring that the assumptions made about the mesh are not violated. Second, updating

connectivity information after an operation must be rapid. Third, mesh elements must be able to be inserted and removed without expensive memory handling (such as manual memory shuffling).

Mesh traversal queries used by the high-level operations are predominantly face-based. Particularly queries that acquire the vertices and edges incident to a face are extensively used. We also want to use edge-based queries that access the endpoints and incident faces of an edge. These four elementary traversal operations are sufficient to allow efficient access to neighboring faces of a given face by following a face-edge-face association. Note that we have no need for vertex-based adjacency associations because no operation starts at a vertex. Based on these aspects, it makes sense for our representation to be primarily face-based, but also represent edges explicitly to support fast face-edge traversal.

A notable difference between face-based and edge-based representations is that the former provides atomic access to incident elements of a face, while the latter provides faster access to one-ring neighborhoods [BKP+10] of vertices, which is more cumbersome in face-based structures. Because we traverse face-based connectivity much more often than one-ring neighborhood connectivity, it makes sense to use a face-based data structure.

3.2.1.2 Rendering requirements

Because we want the the visualization to mirror the state of the mesh, it must be updated as soon as the mesh undergoes geometric modification. However, there is usually a conflict of interest between topological elegance and the data structure sent to the rendering pipeline: the former must be easily navigable while the latter must be compact. It is therefore desirable to make the translation process from the topological mesh to a renderable mesh as efficient as possible, while avoiding redundancy.

This can be achieved by maintaining a tight coupling between the topological and renderable mesh. This way we can prevent converting from one representation to another during runtime, thus improving performance. We define three constraints that help define coupling. First, we express the topology as a collection of triangle primitives, avoiding the need to perform an expensive triangulation before rendering. Second, the vertex winding order is adapted accordingly to the expected input by the rendering system (in this case clockwise). Third, a list of unique vertices and a list of vertex indices is maintained side by side with the topological representation.

Note that we are constricted by data structures supported by graphics hardware. Of the possible data structures used as input for graphics hardware, the indexed triangle list (another term for the indexed face set, see Appendix A) is both most efficient for general shapes [Sup10] and most widely used. Although this data structure enjoys accelerated rendering capabilities, its lack of topological information – besides direct access to the vertices of each face – makes traversal and modification operations laborious and inefficient. In general we want to avoid traversing the full array of vertices, edges, or faces during a query at all costs. Hence, we want to define a mesh topology that is efficient at accessing local connectivity information, and simultaneously supports efficient extraction of an indexed triangle list used for rendering.

3.2.1.3 Storage and performance requirements

The design of the representation is also influenced by memory consumption, querying performance, and updating performance [SSF11]. Storing relationships between any two elements requires more memory and makes the updating process more time consuming, but speeds up querying operations. Conversely, computing relationships on demand requires less memory and reduces the cost of updating connectivity information, but querying operations become more time consuming. Although in this thesis no strict rules upon memory consumption are imposed, it is desirable to only explicitly store relationships that are most often accessed and to derive those that are less often required.

3.2.2 Evaluation of traditional representations

In this section we evaluate the traditional mesh representations described in Appendix A with respect to our requirements.

As we have discussed above, our mesh cutting simulation does not use all adjacency queries. Thus comprehensive representations that store a large amount of local adjacency information like full adjacency lists or corner-tables are redundant and needlessly complex to maintain. We want topology to be represented in a concise and efficient manner while retaining all the information required to perform the desired processing operations.

Because the connectivity of a mesh is primarily encoded in edges [BKP+10], data structures for general polygon meshes are often edge-based. For triangle meshes however, face-based structures can be more convenient. This is because face-based structures allow easy access to adjacent edges and vertices for a particular face, while this requires more work in edge-based structures.

Assuming that the mesh is manifold, traditional edge-based structures like winged-edges and quad-edges can be replaced by halfedge-based data structures to reduce the memory footprint and algorithmic complexity. Since we only consider triangle meshes, the directed edge structure – a specialized form of the standard halfedge structure – seems appealing. However, because adjacency information is referenced by index, maintaining a dynamically changing mesh is a laborious and complex process. Since much of the storage reduction of a directed edge structure comes from its index-based references, replacing indexes with pointers essentially brings us back to the original halfedge data structure. Although an indexed face set representation can be extracted from a halfedge structure with minimal overhead, we think it is more desirable to avoid extraction altogether and instead store a renderable mesh side by side with the topology.

The mesh representation by Tobler and Maierhofer [TM06] is an example of a face-based structure that explicitly stores edges and has a close coupling between rendering and topology. However, it suffers from a number of drawbacks that are inconvenient when dealing with a dynamic mesh that changes often. In particular having to store the orientation of incidence relationships and the difficulty of modifying the topology due to the use of indexed arrays are cumbersome. Nevertheless, we borrow the idea of storing and maintaining a topological and a renderable mesh simultaneously.

3.2.3 Definition

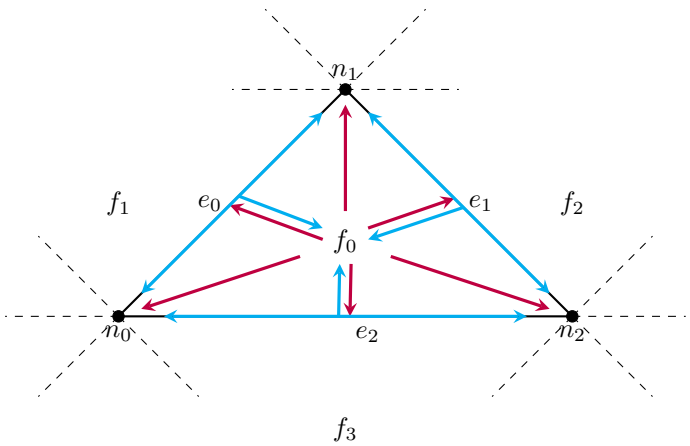
A compromise between efficient topology traversal and efficient extraction of render data can be achieved by representing the mesh with a face-based connectivity structure (or face-based partial adjacency list). This is realized by augmenting the indexed face set to include connectivity information. By explicitly storing vertices, edges, and faces, mesh traversal queries can be performed as efficiently as in the edge-based structures, while rendering can be as efficient as in face-based structures.

Formally, a triangular mesh $\mathcal{M}$ is defined as a pair $\{\mathcal{P}, \mathcal{K}\}$, where $\mathcal{P} = \{p_0, \ldots, p_n \mid p_i \in \mathbb{R}^3\}$ is the set of vertex positions that define the shape of the mesh in three-dimensional space, and $\mathcal{K}$ is a simplicial complex specifying the connectivity of the mesh simplices [HDD+93]. A simplex of dimension k is the convex hull of $k+1$ points in general position, and is connected, compact, and closed. In this notation, 0-simplexes $\{i\} \in \mathcal{K}$ correspond to the vertices of the mesh, 1-simplexes $\{i, j\} \in \mathcal{K}$ are edges, and 2-simplices $\{i, j, k\} \in \mathcal{K}$ compose the faces of $\mathcal{M}$ [LLM+10].

A simplicial complex $\mathcal{K}$ is a finite set of simplices together with all its subsimplices such that if simplices σ and τ belong to $\mathcal{K}$, then either $\sigma \cap \tau \neq \emptyset$ (they intersect at a subsimplex), or σ and τ are independent [Ede01]. A topology can be formed on $\mathcal{K}$ by identifying its vertices with standard basis vectors $\{\boldsymbol{e}_1, \ldots, \boldsymbol{e}_m\}$ in affine space $\mathbb{R}^m$ and defining a linear mapping $\mathbb{R}^m \to \mathbb{R}^3$ that maps the i-th standard basis vector $\boldsymbol{e}_i \in \mathbb{R}^m$ to geometric point $p_i \in \mathcal{P}$. For each simplex $\sigma \in \mathcal{K}$, let $|\sigma|$ denote the convex hull of its vertices in $\mathbb{R}^m$. We can then say that an arbitrary union $|\mathcal{K}|$ of all closed simplices describes the topological realization of $\mathcal{K}$ [HDD+93]: $|\mathcal{K}| = \cup_{\sigma \in \mathcal{K}} |\sigma|$.

Different embeddings may be imposed on any given mesh topology to obtain different connectivity relations. For a mesh of topological dimension D, there are $(D+1)^2$ different classes of incidence relations (connectivities) to consider [Log09]. For a triangular mesh ($D = 2$), this means there are nine different incidence relations between the mesh simplices. The set of mesh elements and the incidence relations between them define the topology of the mesh. In truth, a complete mapping of connectivity can be derived from the incidence relation between faces and vertices. Other incidence relations are merely maintained to increase the efficiency of traversing the topology.

We will now describe the topological embedding of our mesh representation. Our embedding $\mathcal{K} = \{\mathcal{V}, \mathcal{N}, \mathcal{E}, \mathcal{F}\}$ consists of vertex set $\mathcal{V}$, node set $\mathcal{N}$, edge set $\mathcal{E}$, and face set $\mathcal{F}$. Figure 3.1 shows the the topological embedding for an example polygon mesh.



$$
\begin{aligned}
e_0 &= \{\{n_0, n_1\}, \{f_0, f_1\}\} \\
e_1 &= \{\{n_1, n_2\}, \{f_0, f_2\}\} \\
e_2 &= \{\{n_2, n_0\}, \{f_0, f_3\}\} \\
f_0 &= \{\langle n_0, n_1, n_2\rangle, \langle e_0, e_1, e_2\rangle\}
\end{aligned}
$$

Figure 3.1: An illustration of the connectivity information for a single face in the mesh representation described in this section. Red lines denote incidence relationships recorded by a face, and blue lines denote incidence relationships recorded by an edge. Although not explicitly shown here, face f_0 also references the vertices defined at node positions n_{0_p}, n_{1_p}, and n_{2_p}.

3.2.3.1 Vertices

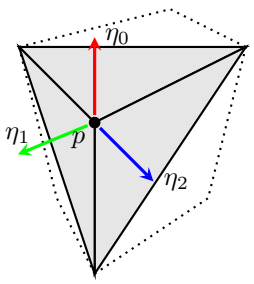
The set of mesh vertices $\mathcal{V}$ is defined as a combination of the geometrical set $\mathcal{P}$ and an additional set $\mathcal{A}$ of attribute information. Each element $a_i \in \mathcal{A}$ is composed of texture coordinates x_i, a normal vector η_i, a tangent vector t_i, and a bitangent vector b_i that are parametrized over the mesh surface. As such, a vertex $v \in \mathcal{V}$ is defined not only by position, but also by texture coordinates and tangent-space basis vectors (normal, tangent, bitangent):

$$v := \{p, \langle x, \eta, t, b\rangle \mid p \in \mathcal{P} \text{ and } \langle x, \eta, t, b\rangle \in \mathcal{A}\},$$

where tuple $\langle x, \eta, t, b\rangle$ describes the subset of scalar attributes that is associated with vertex v. This definition implies that any two vertices are different if either their position differs, their attributes differ, or both. Discontinuities in the scalar fields of the attributes – such as along creases and seams of the mesh – are handled by defining multiple vertices at a surface point that lies on the border of such a discontinuity in the mesh parameterization. Figure 3.2 shows the implications of discontinuities in the tangent space, and Figure 4.6 in Section 4.1.2 describes how texture seams are handled.

3.2.3.2 Nodes

The vertex definition precludes topological information: set $\mathcal{V}$ describes a disconnected mesh if a given position p occurs more than once in $\mathcal{V}$. Since attribute information is not used for purely topological



$$v_0 = \langle p, x_p, \eta_0, t_0, b_0 \rangle$$
$$v_1 = \langle p, x_p, \eta_1, t_1, b_1 \rangle$$
$$v_2 = \langle p, x_p, \eta_2, t_2, b_2 \rangle$$

Figure 3.2: A discontinuity in the tangent space leads to the definition of three distinct vertices at point p. Here we assume three discontinuous tangent bases ($\langle \eta_i, t_i, b_i \rangle$) are defined at p (only normal vectors are shown). However, the texture coordinate x_p at p is continuous over the three faces adjacent to p, allowing this quantity to be reused. Note that normalizing the tangent bases over v_0, v_1 and v_2 would reduce the three vertices into a single one.

operations, we can define a topological counterpart to the vertex called a *node* that is merely defined by its geometric location:

$$n := \{p \mid p \in \mathcal{P}\}\,.$$

Set $\mathcal{N}$ of all nodes thus corresponds to the set of geometric points $\mathcal{P}$, and is used in the description of the mesh connectivity. This distinction between vertices and nodes becomes more apparent when discussing the algorithms performed on the mesh.

3.2.3.3 Edges

Mesh edges are described by the set $\mathcal{E}$. In a topological embedding based on simplexes, edges describe the incidence relation between two adjacent nodes and between two adjacent faces. As such, edge $e \in \mathcal{E}$ is defined as a set of two *unordered* pairs of references to its adjacent nodes and faces

$$e := \{\{n_0, n_1\}, \{f_0, f_1\} \mid n_i \in \mathcal{N},\ f_i \in \mathcal{F}\} \text{ for } i = 0, 1\,,$$

where $\mathcal{F}$ is the set of mesh faces defined below. We store face adjacency information in edges rather than in faces as a convenience for the mesh subdivision algorithms presented in the next chapter. Furthermore, although an edge definition can be derived from two incident nodes of a face, explicitly storing them allows us to encode adjacency between face pairs and node pairs in a simple topological construct.

Note that node pair $\{n_0, n_1\}$ does not have a particular ordering: n_0 and n_1 do not necessarily describe the origin and target nodes of the edge. Therefore, e describes an undirected full edge (as opposed to a half-edge) that is shared between faces f_0 and f_1, where it is unclear which one one is on the left or right side of the edge. This may seem like a significant hindrance, especially considering that the mesh was assumed to be orientable (the vertices of a face have a particular winding order), but because each face is composed of exactly three edges and nodes, the orientability property can be preserved in the face definition instead.

3.2.3.4 Faces

The set $\mathcal{F}$ of faces is the cornerstone of the topological realization of the mesh, where a face f is described by three *ordered* tuples of incidence relations:

$$f := \{\langle n_0, n_1, n_2 \rangle, \langle v_0, v_1, v_2 \rangle, \langle e_0, e_1, e_2 \rangle \mid n_i \in \mathcal{N},\ v_i \in \mathcal{V},\ e_i \in \mathcal{E}\} \text{ for } i = 0, 1, 2\,.$$

This definition allows all incident elements to be accessed either directly or via an incident edge. Both vertices and nodes are referenced by the face, where the nodes are shared between adjacent faces and can thus be used in topological queries, and the vertices may or may not be shared between two neighboring

faces depending on their exact attributes. Edges $\langle e_0, e_1, e_2 \rangle$ exclusively reference nodes $\langle n_0, n_1, n_2 \rangle$ incident to face f and no others:

$$e_i \mapsto n_j \in \{n_0, n_1, n_2\} \quad \text{for } i = 0, 1, 2 \text{ and } j = 0, 1,$$

where $\mapsto$ expresses an adjacency relation from one mesh element to an incident mesh element.

Face f specifies an ordering of the nodes, vertices, and edges. This winding order can be either clockwise or counter-clockwise, and must be the same for all faces in order to satisfy the orientability property. In this case it was chosen to use a clockwise ordering in compliance with Direct3D that uses a left-handed coordinate system [Mic14], thus avoiding any conversion overhead. Moreover, the geometric position of nodes $\langle n_0, n_1, n_2 \rangle$ and vertices $\langle v_0, v_1, v_2 \rangle$ are synchronized:

$$n_i.p = v_i.p \quad \text{for } i = 0, 1, 2,$$

where $p \in \mathcal{P}$ denotes the geometric position of node n_i and vertex v_i. We use the dot notation to indicate attribute associations. Although edges are undirected, their ordering in the face definition also corresponds to the left-handed ordering of the nodes (and vertices), where e_0 is incident to n_0 and n_1, e_1 is incident to n_1 and n_2, and edge e_2 is incident to n_2 and n_0 in arbitrary order.

3.2.3.5 Representation of incidence relationships

One consideration that has not been addressed is whether incidence relationships should be represented as indexes or as pointers in the implementation. Since triangle lists used for rendering require indices to an array of vertex declarations it makes sense to store face-vertex references ($f_i \to v_j$) as indexes.

However, other incidence relations do not benefit from index-based references as the removal of edges and faces invalidates indices elsewhere in the mesh, necessitating an update to all references in the mesh. This has a time complexity of $\mathcal{O}(n+m)$, where n and m are the number of faces and edges, respectively.

Instead we store the sets of nodes, edges, and faces in dynamic arrays and describe references by pointers that point to elements of those arrays. Using pointers not only eliminates the need to update the list of faces and edges, but the complexity of removing a face or edge is only linear on the number of elements positioned after the removed element, so $\mathcal{O}(n)$ *or* $\mathcal{O}(m)$ in the *worst-case* scenario.

3.2.3.6 Storage complexity

Memory consumption of this representation is $3 \times 4 = 12$ bytes for each vertex (excluding attribute data) and for each node. Here we ignore the fact that there may be duplicate vertex declarations at each node as it is unclear how to measure this duplication. Edges store four pointer references, which is $4 \times 4 = 16$ bytes, and faces store three indices and six pointers, or $3 \times 4 + 6 \times 4 = 36$ bytes. Following the Euler-Poincaré characteristic, the total memory consumption amounts to $12 + 12 + 2 \times 36 + 3 \times 16 = 144$ bytes, which is equal to the non-optimized halfedge data structure [BKP+10].

3.3 Mesh construction and extraction

Constructing the mesh representation consists of two parts: loading it from a file or memory, and generating the topological realization. Additionally, during runtime we need to extract a renderable mesh from the topological mesh so that it can be displayed on screen.

3.3.1 Mesh loading

We have written a custom mesh loader implementation that reads and parses ASCII Wavefront object files given as input. These files list the positions, normals (one per face), and texture coordinates of each

vertex, and describe triangular mesh faces by triples of indexes. Although the normal vectors and texture coordinates are optional, and many more structures are described in the official specification [Wav95], we only support files that contain the basic information described above.

Algorithm 3.1 Producing vertex and index array from Wavefront object file

Input: Wavefront object file O.
Output: Vertex set $\mathcal{V}$ and index set $\mathcal{I}$ (indexed triangle list).

```
1:  i ← 0                                  ▷ Vertex indexer.
2:  P ← {P_0, ..., P_n}                    ▷ Vertex positions.
3:  N ← {N_0, ..., N_n}                    ▷ Vertex normals (non-smoothed).
4:  X ← {X_0, ..., X_m}                    ▷ Vertex texture coordinates.
5:  J ← ∅                                  ▷ Vertex indexers (indexes to the other sets).
6:  for each vertex definition J in O do   ▷ There are three vertex definitions per face.
7:      if J ∉ J then                      ▷ If J was not encountered before.
8:          J_i ← i                        ▷ Store vertex index in J.
9:          V ← MakeVertex(J)              ▷ Create vertex from indexes to other data defined
                                             by J.
10:         J ← J + J                      ▷ Add J to J to detect duplicates.
11:         I ← I + i                      ▷ Add index to set of indexes.
12:         V ← V + V                      ▷ Add vertex to set of vertexes.
13:         i ← i + 1                      ▷ Increment index (new vertex was added).
14:     else
15:         I ← I + J_i                    ▷ Retrieve vertex index for J and add to I.
16: return Set of vertexes V and indexes I.
```

Algorithm 3.1 gives an overview of the mesh loading process. While parsing the file, the mesh loader constructs three arrays that store the positions, face normals, and texture coordinates, respectively. For each face defined in the file three indexers – consisting of triples of indexes to position, face normal, and texture coordinate data – are created and given a unique identifier. Since indexers may be shared between faces, a new identifier is appointed only if its exact configuration of indexes has not been stored before, and otherwise a previously stored identifier is reused.

Each of the indexers is then processed to produce arrays of normals, tangents, and bitangents. Tangent and bitangent vectors are computed as described in [Len12]. Note that the normals specified in the object file are face normals, and as such are shared between adjacent faces. To make sure that the vector field of normals is smooth instead of dicontinuous, each normal is computed as an average over all adjacent faces of a vertex. Alternatively, if no normals were specified in the input file, they can be computed from the vertex positions of a face assuming a counter-clockwise ordering.

After computing the arrays of positions, texture coordinates, vertex normals, tangents, and bitangents, the list of indexers is processed one more time to construct the actual vertex list. Finally, the mesh loader produces an index list and a vertex list that are sent to the rendering pipeline, and are used to build the mesh representation with.

Since mesh loading can take tens of seconds for reasonably sized meshes (with thousands of vertexes), a binary representation of the mesh is written to a file once a mesh has been successfully loaded. This file contains the lists of index data and vertex data created during mesh loading, as well as the sizes of these lists. Reading from this binary data file simplifies the mesh building, greatly reduces the start-up time of the application, and has the added benefit of requiring less storage space than the ASCII formatted OBJ file.

Using the index and vertex arrays as input, nodes, vertices, edges, and a face structure is constructed for each index triple. The index and vertex data is read from a file in advance, and must be correctly formatted for rendering. This means that the vertex array contains unique vertices, and that the index array contains triples of indices to vertices in the vertex array, each one forming a face.

3.3.2 Generating the topological mesh

After constructing the vertex and index arrays used for rendering, the topology defined earlier will be generated from this indexed face set. Algorithm 3.2 describes the general steps.

Algorithm 3.2 Generating the topology from indexed face set.

Input: Vertex set $\mathcal{V}$ and index set $\mathcal{I}$.
Output: Topological realization $|\mathcal{K}|$ of mesh $\mathcal{M}$.

1: **for each** Index triple $\langle i_0, i_1, i_2 \rangle \in \mathcal{I}$ **do** ▷ Iterate over face definitions.
2: $n_0 \leftarrow \textsc{MakeNode}(\mathcal{V}(i_0))$ ▷ Make a node using vertex position.
3: $n_1 \leftarrow \textsc{MakeNode}(\mathcal{V}(i_1))$
4: $n_2 \leftarrow \textsc{MakeNode}(\mathcal{V}(i_2))$
5: $e_0 \leftarrow \textsc{MakeEdge}(n_0, n_1)$ ▷ Make an edge between two incident nodes.
6: $e_1 \leftarrow \textsc{MakeEdge}(n_1, n_2)$
7: $e_2 \leftarrow \textsc{MakeEdge}(n_2, n_0)$
8: $f \leftarrow \textsc{MakeFace}(i_0, i_1, i_2, n_0, n_1, n_2)$ ▷ Make a face from three vertexes/nodes.
9: $\textsc{RegisterFace}(f, e_0, e_1, e_2)$ ▷ Set incidence relations between face and its three edges.
10: $\mathcal{F} \leftarrow \mathcal{F} + f$
11: $\mathcal{E} \leftarrow \mathcal{E} + e_i$ for $i = 0, 1, 2$
12: $\mathcal{N} \leftarrow \mathcal{N} + n_i$ for $i = 0, 1, 2$
13: **return** Set of faces $\mathcal{F}$, edges $\mathcal{E}$, and nodes $\mathcal{N}$.

First of all, note that the index array consists of sequential triples of indexes that point to vertexes incident to the same face. By iterating over each triple and following the indexes we can first construct the three nodes of the face. Recall that the nodes only store a position, so this is simply extracted from each vertex object created earlier. Then, three edges between the three nodes are created, where the incidence relations are constructed as described above. Finally the face defined by the three nodes and three edges is constructed. This face object also contains indexes to vertexes in the vertex array which are used during extraction of the renderable mesh. Incidence relations between the face and its three edges are completed afterward since a circular dependency is defined on the face and edges: the face references three edges which each reference that same face.

Internally the three `Make` functions test if the element that is attempted to be added was already defined before. For nodes this is determined by using a hash function on the position they store. This function first hashes each of the individual coordinates (`floats`) of the geometric position, and then combines each of the hash values with the `hash_combine` function from the Boost library [Jam08].

Similarly, edges are hashed based on the two nodes references n_0 and n_1, but care must be taken that for node reference pair $\langle n_0, n_1 \rangle$ the inverse pair $\langle n_1, n_0 \rangle$ is mapped to the same hash value. This is realized by imposing an invariant on the node references that orders them by geometric position, where the floating-point coordinates of the positions nodes n_0 and n_1 are located at are lexicographically compared with the less than ($<$) relation. Afterward, the two hashed nodes are again combined into a single hash value for the edge with `hash_combine`.

Finally, each face is hashed based on the three nodes it is composed of, and thus the process of hashing each node and combining their hash values is repeated.

3.3.3 Extracting a renderable mesh

Extraction of a renderable mesh from the topological mesh (the sets of faces, edges and nodes; and vertex data) occurs every time the mesh geometry and topology is modified. To have the rendering pipeline display the modified mesh the vertex and index buffers must be reconstructed, which means updating the vertex and index arrays created during mesh loading. Note that vertexes are never removed, but new vertexes can be added and their data can be modified. This makes it fairly easy to update the vertexes during mesh processing algorithms. However, for the index array it is much more difficult and inefficient

to access specific local areas of the mesh. Instead of attempting to create a complex structure that will make this possible, the whole problem can be circumvented by simply rebuilding the indexed face set from the set of faces $\mathcal{F}$. This results in a simple algorithm shown in Algorithm 3.3.

Algorithm 3.3 Rebuilding the indexed face set for rendering.

Input: Set of faces $\mathcal{F}$, index array `indexes`.
Output: Updated index array `indexes`.

```
index ← 0
for each face f ∈ F do                          ▷ Iterate over each face.
    for each vertex index j ∈ f ↦ ⟨i_0, i_1, i_2⟩ do   ▷ Iterate over triple of indexes.
        indexes[index] ← j
        index ← index + 1
return indexes
```

Closing remarks

In this chapter we proposed a data structure that is capable of representing the geometry and topology of a dynamically changing mesh. It is primarily face-based and augmented by the explicit representation of edges in order to facilitate efficient face-edge and edge-face traversal. In order to allow managing the geometry and mesh parameterization separately from the topology we make a distinction between vertices and nodes. Simultaneously we maintain a renderable mesh in the form of an indexed face set that is updated after geometrical or topological modifications.

We use our novel mesh representation in the following two chapters where we discuss simulating and visualizing cutting wounds, respectively.

4. Simulating cutting wounds

In this chapter we describe our approach of simulating cutting wounds on a three-dimensional surface mesh. In reality, incisions on human skin are produced when a sharp object collides with the tissue, tearing the skin open and leaving the subsurface tissue visible. We reproduce this phenomenon by using mesh editing techniques to locally modify the mesh area that is affected by the cut. In this chapter we discuss the algorithms and techniques associated with the cut selection and cutting simulation components from Figure 2.3. This consists of interactive selection of the location of a cut, formation of the cutting line on the surface of the mesh, fusing this cutting line into the mesh topology, opening the cutting line, and generating the cutting gutter. These stages are repeated in Figure 4.1 below.

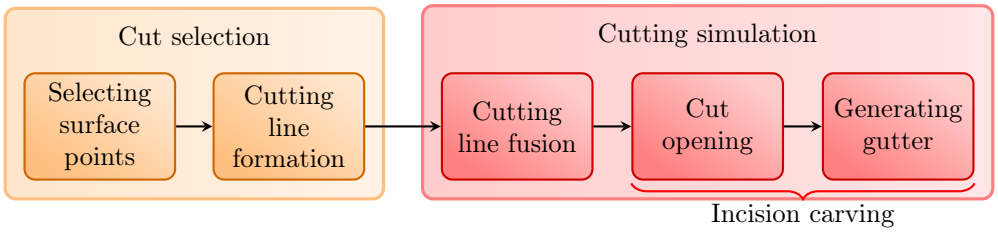


Figure 4.1: Overview of our cut selection and mesh cutting simulation. Cut selection is described in Section 4.1. The cutting simulation stages are treated in Sections 4.2, 4.3.1, and 4.3.2, respectively. We use the umbrella term "incision carving" for the latter two stages.

4.1 Cut selection

In the first stage of our mesh cutting approach we allow a user to select the location of a new cutting wound. In this section we describe how a cut is defined by two points in screen space and how this is translated into a cutting line on the surface of the mesh. In section 4.3 we will then describe how this cutting line is separated to create an open wound.

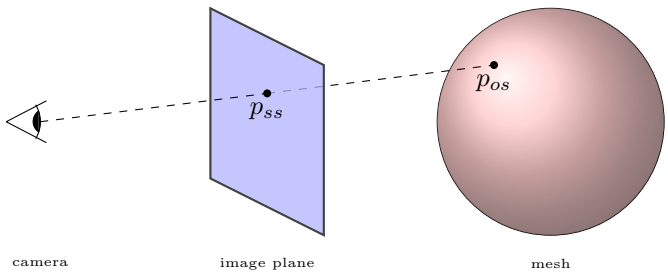


Figure 4.2: Using ray casting to transform a screen-space point (p_{ss}) into an object-space point p_{os} on the surface of a mesh.

4.1.1 Selecting surface points using ray casting

New incisions are defined by selecting a starting and ending location of a line segment that represents the cutting line on the mesh surface. Although several input options are possible, in our test application we allow a user to select points on the screen, which are translated into points on the mesh surface. Such a translation requires shooting a ray into the scene, followed by a number of collision tests to find the exact point of intersection on the mesh surface. See Figure 4.2.

4.1.1.1 Ray construction

Note that a point in two-dimensional screen space does not have a direct mapping to a three-dimensional point in world space, but rather translates into a ray in world space. A ray (or half-line) R is described by an origin and direction vector: $R(t) = O + t\boldsymbol{D}$.

In this case the origin is located on the image plane, and the ray travels in the direction of the viewing frustum. Obtaining a ray in object space from a selected point in screen space involves reversing the transformation process performed by the rendering pipeline. See Figure 4.3. Below we summarize a process that can be found in many computer graphics textbooks or online resources.

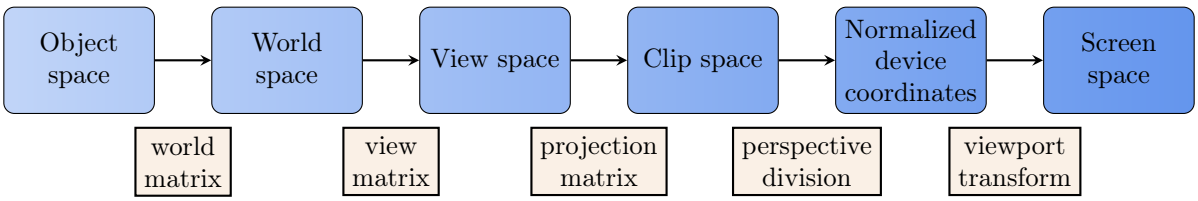


Figure 4.3: Coordinate spaces and transformations in the rendering pipeline.

We first manually reverse the viewport transform by multiplying screen-space point p_{ss} with the reciprocal matrix of the viewport scaling and accounting for the viewport offset. This gives us point p_{ndc} in normalized device coordinates.

A point in normalized device coordinates has to be unprojected to get back to view space. Note that the perspective division can be skipped – we merely have to obtain two points that lie on the ray. This can be done by unprojecting p_{ndc} once at the near plane and once at the far plane of the viewing frustum.

Normalized device coordinates are simply transformed directly into object-space coordinates by multiplying with the inverse world-view-projection matrix. Doing so gives us the origin O and direction $\boldsymbol{D}$ of our ray in object space.

4.1.1.2 Ray intersection

To find out whether or not a selected point in screen space corresponds with a point on the mesh surface, we use the constructed ray to perform intersection tests. For simplicity we will assume that the scene only contains one mesh. If the ray does not hit the mesh the intersection test will report a negative answer. In the case that more than one intersection was detected, which often happens due to the front and the back of a mesh being hit, the intersection closest to the viewing point is returned.

Algorithm 4.1 describes a naive solution for finding the intersection point. It is naive because space partitioning methods would eliminate the need to iterate over all faces of the mesh. For our application a naive solution suffices because intersections only have to be computed when a user selects a point on the mesh.

For the actual ray-triangle intersection test we use the efficient Möller-Trumbore intersection algorithm [MT97]. Our intersection test returns a tuple containing distance t along the ray where the intersection occurs, as well as the barycentric coordinates u and v with respect to the intersected triangle.

Algorithm 4.1 Finding the closest intersection point (if any) of a given ray and a mesh.

Input: Ray $R(t) = O + t\boldsymbol{D}$ and set of faces $\mathcal{F}$ of the target mesh.
Output: A tuple $\mathcal{I}$ that describes the intersection, or `nil` if there was no intersection.

```
p ← nil
t_min ← ∞
for each face f ∈ F do
    ⟨t, u, v⟩ ← IntersectionTest(O, D, f)   ▷ Computes distance and barycentric coordinates.
    if t < t_min then                       ▷ If no intersection detected, then t = ∞.
        t_min ← t                           ▷ Update minimum detected distance.
        I ← ⟨p, x, R, f⟩                    ▷ World-space coordinate and texture coordinates of
                                              intersection point, ray, and intersected face.
return p
```

In the case of a successful intersection we store position p and texture coordinate x along with ray R and a reference to intersected face f in a tuple $\mathcal{I} = \langle p, x, R, f\rangle$ that is returned by the algorithm. These quantities are used during the incision carving stages. Note that texture coordinates can be computed from the barycentric coordinates (u, v) computed earlier.

4.1.2 Cutting line formation

After the start and end points of the cutting line have been defined, we form a chain of piecewise linear segments whose lengths are delimited by the edges of the mesh. This chain will then allow us to merge the cutting line into the mesh representation. Note that we cannot simply construct a straight line segment between the two intersection points due to the mesh surface not generally being flat. Also, since each point on the cutting line in screen space translates into a ray in object space (and thus have an undetermined depth value), we cannot determine edge crossings with a two-dimensional screen-space segment. Therefore, we determine the intersections of each mesh edge with the cutting line with respect to the viewing frustum active when defining the cut.

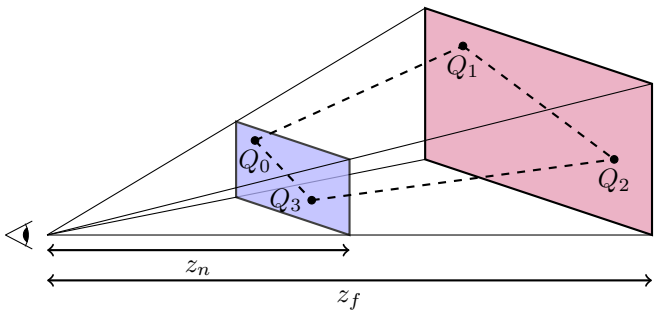


Figure 4.4: Expansion of a 2D cutting line to a 3D cutting quadrilateral. Note the clockwise winding order of its vertexes; Q_0 and Q_3 are located on the near clipping plane of the viewing frustum, and Q_1 and Q_2 on the far plane.

Edge crossings in three-dimensional space can be computed by representing the cutting line as a simple (non-intersecting) convex planar quadrilateral whose sides are constructed from the object-space rays at the selected start and end points of the cutting line. We define a quadrilateral $Q = \square Q_0 Q_1 Q_2 Q_3$ by specifying its four vertexes (in clockwise winding order because we use left-handed coordinate system):

$$
\begin{aligned}
Q_0 &= O_0 + z_n \boldsymbol{D}_0 \\
Q_1 &= O_0 + z_f \boldsymbol{D}_0 \\
Q_2 &= O_1 + z_f \boldsymbol{D}_1 \\
Q_3 &= O_1 + z_n \boldsymbol{D}_1,
\end{aligned}
$$

where O_0 and O_1 are the origins of the rays at the endpoints of the cutting line, and $\boldsymbol{D}_0$ and $\boldsymbol{D}_1$ are their directions. Furthermore, z_n and z_f denote the distances from the viewpoint to the near and far clipping planes of the viewing frustum, respectively. See Figure 4.4.

The cutting quadrilateral is used to find the piecewise linear line segments on the surface of the mesh that denote the cutting line. These segments are ordered from the starting intersection point to the ending intersection point computed above. Algorithm 4.2 describes this cut line formation process.

Algorithm 4.2 Forming a chain of segments that describes the cutting line on the surface of the mesh.

Input: Quadrilateral Q, starting face F_0, and surface intersection points P_0 and P_1 at the start and end of the cutting line, respectively.

Output: Ordered set of segments $\mathcal{C}$ that describes the cutting line.

```
f ← F_0                                   ▷ Current face.
p_0 ← P_0                                 ▷ First edge crossing of segment.
p_1 ← nil                                 ▷ Second edge crossing of segment.
T ← ∅                                     ▷ Hash table for edges.
C ← ∅                                     ▷ Cutting line chain.
while f ≠ nil do                          ▷ Stop looping if no more intersections.
    f ← nil
    for each edge e ∈ f_E do              ▷ Iterate over three edges incident to f.
        if e ∉ T then                     ▷ Edge was not visited yet.
            T ← T + e                     ▷ Mark edge as visited.
            R ← CreateRay(e)              ▷ Create ray from endpoints of e.
            t ← RayQuadIntersection(R, Q) ▷ Compute distance t to edge crossing.
            if t ≠ nil and 0 ≤ t ≤ 1 then ▷ Crossing lies on edge e.
                p_1 ← R(t) = O + tD       ▷ Compute intersection point.
                C ← C + CreateLink(f, p_0, p_1) ▷ Add segment p_0p_1 to C.
                p_0 ← p_1
                if f = e_{F_0} then       ▷ Continue with face opposite to e.
                    f ← e_{F_1}
                else
                    f ← e_{F_0}
                break                     ▷ Skip testing remaining edges of f.
C ← C + CreateLink(f, p_1, P_1)           ▷ Add last segment to C.
return C
```

The algorithm starts at the face that contains the first intersection point (recall that this was stored earlier), and performs an intersection test with the quadrilateral for each of its incident edges. There is only one such crossing per face, and whenever it is found the algorithm moves on to the face that is incident to that edge for new intersection tests. If none of the edges intersect the quadrilateral, then the penultimate segment was encountered and only the last segment remains to be added. Note that for this to work each visited edge must be marked so that it is not tested a second time when transitioning to a neighboring face.

Function `CreateLink` creates a structure $\ell = \langle f, p_0, p_1, x_0, x_1 \rangle$ that represents a link in the cutting line chain: i.e. the line segment whose endpoints p_0 and p_1 correspond to the previous edge crossing and newly found edge crossing, respectively. This structure stores additional information used later: the face f in which the segment lies, and the texture coordinates x_0 and x_1 at the locations of its endpoints.

Edge crossings are determined with a ray-quadrilateral intersection algorithm that takes a ray representing the edge and the cutting quadrilateral as input, and returns the distance t along this ray where the intersection point is located, or `nil` if no such intersection is detected. A ray can be created from edge e by using one of its endpoints $e \mapsto n_0$ as its origin, and the vector from that endpoint to the other as its direction: $(e \mapsto n_1) - (e \mapsto n_0)$. Note that we use the $\mapsto$ symbol to indicate an association between mesh elements.

Internally, the ray-quadrilateral intersection algorithm is based on the work by Lagae and Dutré [LD05]. They describe a simple scheme of decomposing the input quadrilateral into two ray-triangle intersection tests that avoids testing both triangles unnecessarily. Since input quadrilateral $Q = \langle Q_0, Q_1, Q_2, Q_3 \rangle$ is convex and planar, its diagonal Q_1Q_3 determines two coplanar triangles, $T = \langle Q_0, Q_1, Q_3 \rangle$ and $T' = \langle Q_1, Q_2, Q_3 \rangle$. Each point in the plane of the quadrilateral can be expressed with barycentric coordinates with respect to either triangle. After computing barycentric coordinates (α, β) of the intersection point with respect to T (using the Möller-Trumbore algorithm), we can discern a number of ray rejection cases.

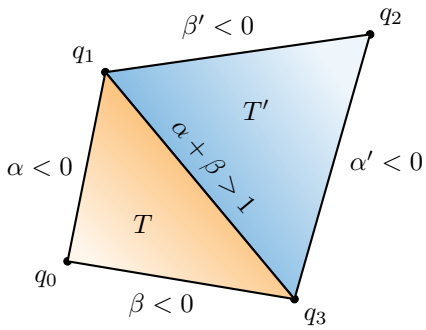


Figure 4.5: Rays are rejected based on the barycentric coordinates (α, β) with respect to triangle T, and (α', β') with respect to triangle T'. Color gradients approximate the increase of $\alpha + \beta$ and $\alpha' + \beta'$.

As we can see from Figure 4.5, if $\alpha < 0$ or $\beta < 0$ the intersection point lies outside of the borders of Q. If $\alpha \geq 0$, $\beta \geq 0$ and $\alpha + \beta \leq 1$, then the ray intersects T (and thus Q), and no further testing is necessary. However, if $\alpha \geq 0$, $\beta \geq 0$ and $\alpha + \beta > 1$, then the intersection point is not in T, but could potentially lie in Q. This can be tested by computing the barycentric coordinates (α', β') of the intersection point with respect to T', and if $\alpha' \geq 0$ and $\beta' \geq 0$, the ray hits T' and thus Q, otherwise the ray misses them both. Finally, if the barycentric coordinates of the intersection point are known to be bound by Q, then t is computed in the usual manner as described in [MT97]. If $0 \leq t \leq 1$ then the edge intersects with the quadrilateral.

Texture coordinates with respect to the mesh parameterization are computed by linearly interpolating with the t value between the texture coordinates at the endpoints of edge e. Note that texture coordinates have to be acquired from the face because the edge only stores node references. Unlike position coordinates, which can be transferred between adjacent segments ($p_0 = p_1$, line 16 in Algorithm 4.2), texture coordinates are not always identical for two successive segments. This is due to texture seams causing discontinuities in the parameterization (texture mapping) whenever the cutting line crosses an edge that lies on a seam. Figure 4.6 shows the situation for a cut that crosses a texture seam.

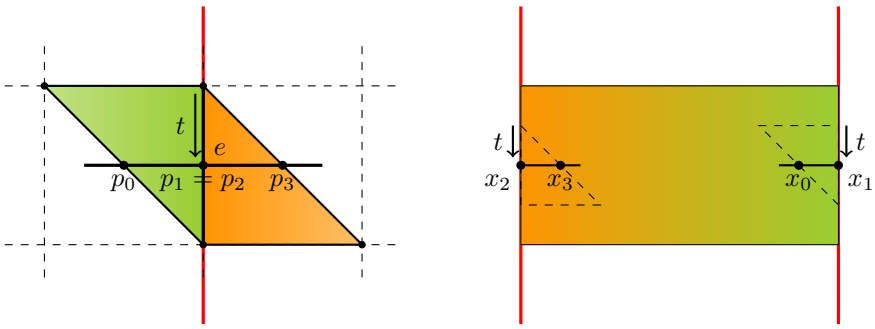


Figure 4.6: Handling cutting lines that cross texture seams (red lines). Left: the geometric situation around edge e. Right: the corresponding mesh parameterization of the two triangles on either side of the seam. Although edge crossings p_1 and p_2 on either side of e are coincident, the texture coordinates x_1 and x_2 at those vertexes are not. Distance t from one edge endpoint to the crossing can be used to compute the appropriate texture coordinates at x_1 and x_2.

Because we cannot generally predict the occurrence of parameterization discontinuities, texture coordinates must be explicitly computed twice for each edge crossing. An elegant solution to this issue is to compute the first texture coordinate (x_0) for next segment after line 20 in Algorithm 4.2 by finding out which of the vertexes of the next face correspond to the endpoints of e, and performing linear interpolation on those vertexes using t. Texture coordinates are thus computed twice per segment; once for the endpoints of e, and once for the endpoints of the corresponding edge in its adjacent face.

4.2 Cutting line fusion

Now that the cutting line chain has been constructed, its segments must be topologically merged into the mesh representation. Doing so will then enable us to open the mesh along the cutting line. Note that naively inserting the segments of the cutting line as edges into the mesh would violate its manifold property. By first refining the local geometry around the cutting line before inserting the new cutting line edges we can avoid this issue. This two-part process is realized by applying elementary face subdivision operations. Figure 4.7 illustrates the process of fusing the cutting line into a section of a mesh.

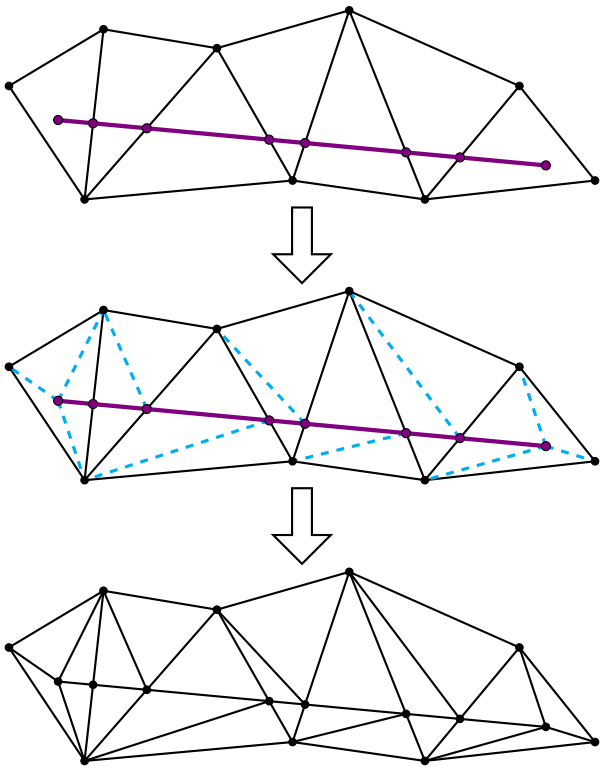
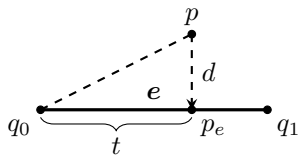


Figure 4.7: An overview of the cutting line fusion process. Top: a cutting line and its intersections with the faces it lies on. Middle: faces are first subdivided along the dashed blue lines, and then along the segments of the cutting line. Bottom: the resulting mesh after fusing the cutting line chain into the mesh.

For every link $\ell = \langle f, p_0, p_1, x_0, x_1 \rangle$ of the cutting line we can discern a number of cases concerning the location of endpoints p_0 and p_1 with respect to face f. Calculated in the previous stage, these two points are guaranteed to be contained inside f, and may either coincide with one of its nodes, lie on one of its edges, or lie in the interior of f. We use the following notation for these three situations:

1. $N(p)$: Point p lies on one of the nodes of face f.
2. $E(p)$: Point p lies on one of the edges of face f.
3. $F(p)$: Point p lies in the interior of face f.

Which situation applies is determined through deduction by testing for node incidence first, then for edge incidence and finally for face incidence, thus delaying the more expensive tests and maximizing the efficiency of the test. Note that whenever a point is not incident to an edge we can automatically assume that it lies in the interior of face f. This follows from the definition of the cutting line formation algorithm.



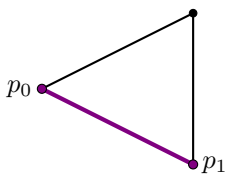
$$\hat{\boldsymbol{e}} = q_1 - q_0 / \|q_1 - q_0\|$$
$$t = \hat{\boldsymbol{e}} \cdot (p - q_0)$$
$$p_e = q_0 + t\hat{\boldsymbol{e}}$$
$$d = \sqrt{(p - p_e)^2}$$

Figure 4.8: Using projection to determine whether a point lies on a line segment. Projected point p_e lies on $\boldsymbol{e}$ if $0 < t < \|q_1 - q_0\|$. If distance d is within a certain threshold ϵ then p is considered to be incident to segment $\overline{q_0 q_1}$.

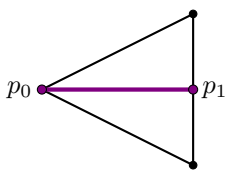
There are various methods of testing whether a point lies on an edge (or line segment). We have found that an approach where p is projected onto each edge outperforms an approach that computes the barycentric coordinates of p in face f. The projection approach is outlined in Figure 4.8.

4.2.1 Cutting line segment configurations

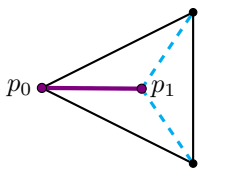
Considering cutting line segment $\overline{p_0 p_1}$ that is contained in face f, we can discern a total of nine configurations, where each one demands a different approach of merging $\overline{p_0 p_1}$ with the mesh. For each configuration the face is subdivided in accordance with the cutting line definition and the mesh properties. Newly inserted edges e_c with p_0 and p_1 as its endpoints are added to an ordered set of edge $\mathcal{E}_\mathcal{C}$ that is used later. The following list describes the fusion process for each of the configurations, where cutting line segments (and corresponding inserted edges e_c) are denoted with purple lines, and additional face subdivisions are indicated by blue lines.



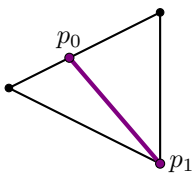
1. $N(p_0)$ and $N(p_1)$
In the case that both p_0 and p_1 lie on a node of f, we have two possibilities: either $p_0 = p_1$ (both points lie on the same node), or they lie on different points and thus corresponds to an edge of f. There are six combinations in the latter case; just one of them is shown here. If $p_0 = p_1$ then no edge between p_0 and p_1 exists and nothing needs to be done. If $p_0 \neq p_1$ then e_c coincides with an existing edge e that connects them, and e is added to $\mathcal{E}_\mathcal{C}$.



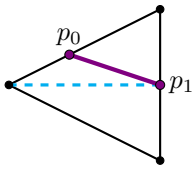
2. $N(p_0)$ and $E(p_1)$
If p_0 lies at a node of f and p_1 lies on its opposite edge then e_c subdivides f into two new triangle faces. Face subdivision algorithm 2-SPLIT is used to split face f into child faces fc_0 and fc_1 at point p_1 and to update the direct neighborhood of f.



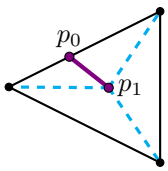
3. $N(p_0)$ and $F(p_1)$
The last configuration that has p_0 lying on a node occurs when p_1 lies in the interior of face f. In this case f is subdivided into three parts, with p_1 being shared between new child faces fc_0, fc_1 and fc_2. Subdivision operation 3-SPLIT creates three new faces and edges in the interior of f that are all incident to p_1. One of the new edges coincides with e_c, and is added to $\mathcal{E}_\mathcal{C}$.



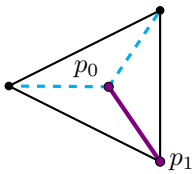
4. $E(p_0)$ and $N(p_1)$
This is the reverse case of configuration 2. Instead of splitting the face at point p_1 we perform a 2-SPLIT operation at point p_0 and add the resulting splitting edge to $\mathcal{E}_C$.



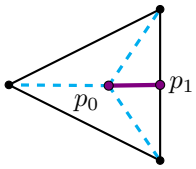
5. $E(p_0)$ and $E(p_1)$
This configuration occurs the most often, where p_0 and p_1 lie on opposite edges. Naively subdividing f here would create one triangle and one quadrilateral, so in order to maintain a triangulation we first 2-SPLIT face f at p_1 (blue line), and then 2-SPLIT one of the new child faces (fc_0 or fc_1) at p_0. Which of the two faces p_0 is incident to can be determined by comparing their edge references with edge $E(p_0)$ that p_0 lies on.



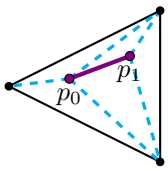
6. $E(p_0)$ and $F(p_1)$
This configuration occurs almost always at the last segment of the cutting line. As in configuration 3 we 3-SPLIT face f at p_1, but then determine in which child face p_0 lies and proceed to 2-SPLIT that face at p_0.



7. $F(p_0)$ and $N(p_1)$
The reverse case of configuration 3: a 3-SPLIT at p_0 creates three new edges, one of which coincides with e_c.



8. $F(p_0)$ and $E(p_1)$
This is the mirrored version of configuration 6 and occurs almost always at the starting segment of the cutting line. Face f is 3-SPLIT at p_0 and one of its child faces is then 2-SPLIT at p_1.



9. $F(p_0)$ and $F(p_1)$
The final configuration has both points in the interior of f. Edge e_c can be inserted by first performing a 3-SPLIT at p_0, and then again at p_1 in one of the newly created child faces. However, due to our assumption that such configurations yield cuts that are too small, cutting lines that lie fully in the interior of a single face are currently discarded.

After determining which of the nine configurations applies to link ℓ, the appropriate 2-SPLIT and 3-SPLIT operations are executed. Operation 2-SPLIT subdivides face f along a splitting edge e_c from point p lying on edge e_s to its opposite node in f, creating two new child faces. Likewise, operation 3-SPLIT subdivides face f in three parts using given point p as its center. These operations are described below.

4.2.2 Two-split

Algorithm 4.3 describes the 2-SPLIT operation in detail. First, to be able conceptualize the relative positioning of the elements of f, we apply the scheme illustrated on the left side of Figure 4.9. Note that it is unknown which of the nodes referenced by f correspond to n_0, n_1, or n_2, but we do know that

Algorithm 4.3 2-Split

Input: Face f, point p, and edge e_s that p lies on ($e_s = E(p)$).
Output: Splitting edge e_c that subdivides f from p to its opposite node n_1.

1: Acquire nodes $\langle n_0, n_1, n_2 \rangle$, vertexes $\langle v_0, v_1, v_2 \rangle$, and edges $\langle e_0, e_1, e_2 \rangle$ of face f according to Figure 4.9 (left).
2: $n_m \leftarrow$ MakeNode(p) ▷ Create midpoint node at p.
3: $v_m \leftarrow$ MakeVertex(v_0, v_2, p) ▷ Create midpoint vertex at p.
4: $e_c \leftarrow$ MakeEdge(n_m, n_1, v_m, v_1) ▷ Create splitting edge between n_m and n_1.
5: $e_{x0} \leftarrow$ MakeEdge(n_0, n_m) ▷ Create exterior edge between n_0 and n_m.
6: $e_{x1} \leftarrow$ MakeEdge(n_m, n_2) ▷ Create exterior edge between n_m and n_2.
7: $f_{c0} \leftarrow$ MakeFace($n_m, n_0, n_1, v_m, v_0, v_1$) ▷ Create child face between n_m, n_0 and n_1.
8: $f_{c1} \leftarrow$ MakeFace($n_m, n_1, n_2, v_m, v_1, v_2$) ▷ Create child face between n_m, n_1 and n_2.
9: RegisterEdge(e_c, f_{c0}, f_{c1}) ▷ Set incident faces of edge e_c.
10: RegisterEdge(e_{x0}, f_{c0}) ▷ Set one incident face of edge e_{x0}
11: RegisterEdge(e_{x1}, f_{c1}) ▷ Set one incident face of edge e_{x1}
12: RegisterFace(f_{c0}, e_{x0}, e_0, e_c) ▷ Set incident edges of face f_{c0}.
13: RegisterFace(f_{c1}, e_c, e_1, e_{x1}) ▷ Set incident edges of face f_{c0}.
14: UpdateEdge(e_0, f, f_{c0}) ▷ Replace face adjacency f of edge e_0 with f_{c0}.
15: UpdateEdge(e_1, f, f_{c1}) ▷ Replace face adjacency f of edge e_1 with f_{c1}.
16: $\mathcal{F} \leftarrow \mathcal{F} - f$ ▷ Remove face f from mesh.
17: $\mathcal{E} \leftarrow \mathcal{E} - e_s$ ▷ Remove edge e_s from mesh.
18: **return** Splitting edge e_c.

they are ordered in clockwise manner. The same applies to edges e_0, e_1 and e_2, which connect nodes n_0 and n_1, n_1 and n_2, and n_2 and n_0, respectively. Additionally, recall that the vertex references of f are indexed in accordance with the indexing of the nodes such that vertex v_i and node n_i are located at the same position ($v_i.p = n_i.p$). As shown by Figure 4.9, edge e_s connects nodes n_0 and n_2 (and vertexes v_0 and v_2). We can thus find n_0 and n_2 with the following statements:

$$\begin{aligned} n_0 &= f \mapsto n_i \quad \text{if } f \mapsto n_i = e_s \mapsto n_0 \quad \text{for } i = 0, 1, 2 \\ n_2 &= f \mapsto n_i \quad \text{if } f \mapsto n_i = e_s \mapsto n_1 \quad \text{for } i = 0, 1, 2. \end{aligned}$$

Node n_1 is then the remaining node of f that does not coincide with one of the nodes referenced by e_s. Since edges are undirected, there is no guarantee that nodes n_0 and n_2 are respectively to the left and to the right of e_c as depicted in Figure 4.9. Therefore, we determine the orientation of $\triangle n_{0_p} n_{1_p} n_{2_p}$ by using the cross product to find its normal:

$$N = (n_{1_p} - n_{0_p}) \times (n_{2_p} - n_{0_p}).$$

If the angle between N and actual face normal N_f is more than 90 degrees, we swap nodes n_0 and n_2.

Vertexes v_0, v_1 and v_2 are acquired in similar fashion. To acquire edges e_0, e_1, and e_2 in accordance to Figure 4.9 we observe that one of the edges is already known: $e_s = e_2$. Moreover, the fact that the edges of f are ordered clockwise allows us to iterate over edges $\langle f \mapsto e_0, f \mapsto e_1, f \mapsto e_2 \rangle$ of face f, and apply the following scheme as soon as edge e_i of f coincides with e_s:

$$\begin{aligned} e_2 &= f \mapsto e_i \\ e_0 &= f \mapsto e_{((i+1) \bmod 3)} \\ e_1 &= f \mapsto e_{((i+2) \bmod 3)}, \end{aligned}$$

where $i \in \{0, 1, 2\}$. This causes e_0 to correspond to the edge that follows e_2 in the clockwise ordering, and e_1 to correspond to the edge that follows e_0 in the clockwise ordering of the edges of f.

The remainder of Algorithm 4.4 consists of creating new geometry and topology, and updating the adjacency references of the mesh elements involved in the splitting. Node n_m is created at position p located along edge e_2. The Make functions return a previously inserted node, vertex, edge, or face if

it already exists in sets $\mathcal{N}$, $\mathcal{V}$, $\mathcal{E}$, or $\mathcal{F}$ respectively, or add a new mesh element to the corresponding set otherwise. On line 3, a vertex is created at p by linearly interpolating the vertex attributes (texture coordinate and normal, tangent, bitangent vectors) between v_0 and v_2. First the relative (parametric) distance t from $v_0.p$ to p is computed:

$$t = \frac{\sqrt{(p - v_0.p)^2}}{\sqrt{(v_2.p - v_0.p)^2}}.$$

The texture coordinate for vertex v_m is then computed by linearly interpolating between texture coordinates x_0 and x_2 at vertexes v_0 and v_2 respectively: $x = (1-t)x_0 + tx_2$, with similar equations for the normal, tangent, and bitangent vectors.

Splitting edge e_c is constructed on line 4. Note that there are two variations of the MakeEdge function. The regular version takes two nodes as input and simply constructs an edge that references those two nodes. Referred nodes of new edge e are geometrically ordered such that $(e \mapsto n_0) < (e \mapsto n_1)$, that is, the individual (x, y, z) coordinates of the nodes are lexicographically compared. This invariant is the reason why edges are undirected, allowing them to be shared between incident faces. However, in this particular case it is more convenient to let e_c be a directed edge from p_0 to p_1, which is helpful during the incision carving process. Therefore an alternative version of MakeEdge is defined to return an edge that stores two pairs of node/vertex pairs located at p_0 and p_1 respectively. This gives us the following definition for edge e_c:

$$e_c = \{\{n_m, n_1\}, \{v_m, v_1\}, \langle\langle n_m, v_m\rangle, \langle n_1, v_1\rangle\rangle\}.$$

Note that the given direction must be reversed in configurations 2 and 8 due to the splitting occurring at p_1 instead of p_0.

Next, the two new edges that split e_s are created, as well as the two new child faces that f is subdivided into. MakeFace simply constructs a face whose cyclic order of nodes and vertexes corresponds to the order of the given arguments. This is also true for the cyclic order of the edges. Since edges and faces are mutually recursive (they store references to each other), some adjacency references cannot be set immediately. Lines 9 to 13 correct this by setting the references to edges e_c, e_{x0}, e_{x1} and faces f_{c0} and f_{c1}. Note that for edges e_{x0} and e_{x1} only one incident face is currently known; the other face reference will be set in the next iteration of the fusion algorithm where edges e_{x0} and e_{x1} are encountered again.

Lines 14 and 15 replace the face reference to f of edges e_0 and e_1 with new faces f_{c0} and f_{c1} respectively. Finally, face f and edge e_s are removed from the mesh. Edge e_c is returned so that it can be added to an an ordered set of directed edges $\mathcal{E}_\mathcal{C}$. This set is used as input for the incision carving process.

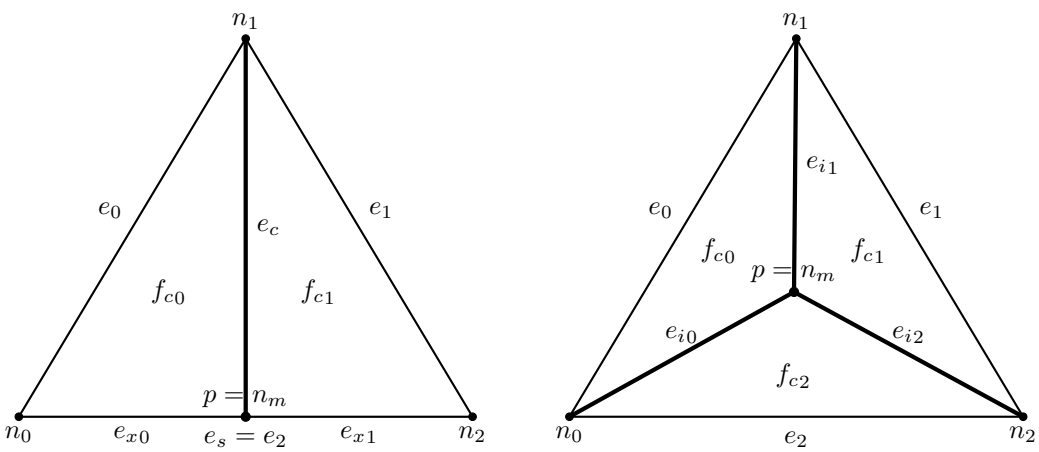


Figure 4.9: Topological situation of face f during the 2-split operation (left) and the 3-split operation (right).

4.2.3 Three-split

The 3-Split operation is shown in Algorithm 4.4 and Figure 4.9. It works similarly to 2-Split but has two important differences: 1) acquisition of nodes, vertices, and edges is straightforward because only their order matters, not their relative positioning; and 2) the operation delivers a well-formed local triangular topology (it does not leave dangling nodes).

Like in 2-Split we first define a new node n_m and a new vertex v_m at given position p. For determining the vertex attributes, instead of performing a linear interpolation between two vertexes we must interpolate between the three vertexes of a triangle. This can be done by expressing point p located in the interior of triangle $\triangle abc$ in terms of its barycentric coordinates (α, β, γ):

$$p = \alpha a + \beta b + \gamma c,$$

with $0 \leq \alpha, \beta, \gamma \leq 1$ and $\alpha+\beta+\gamma = 1$. Since the position at p is already known, but not its corresponding attribute values, we must first compute the barycentric coordinates at p. There are various ways to do this. One way is to compute the ratios of areas between triangle $\triangle abc$ and its subtriangles $\triangle bcp$, $\triangle cap$ and $\triangle abp$ as described in Section 2.7 of [SAM09]. Another approach that proved to be less computationally expensive is the one described in Section 3.4 of [Eri04], where the expression above is reformulated as

$$p = a + \beta(b - a) + \gamma(c - a) \quad \Rightarrow \quad \beta(\boldsymbol{v}_0) + \gamma(\boldsymbol{v}_1) = \boldsymbol{v}_2,$$

where $\boldsymbol{v}_0 = b - a$, $\boldsymbol{v}_1 = c - a$, and $\boldsymbol{v}_2 = p - a$. By taking the dot product of both sides with both $\boldsymbol{v}_0$ and $\boldsymbol{v}_1$, a system of linear equations is formed:

$$\begin{aligned} \beta(\boldsymbol{v}_0 \cdot \boldsymbol{v}_0) + \gamma(\boldsymbol{v}_1 \cdot \boldsymbol{v}_0) &= \boldsymbol{v}_2 \cdot \boldsymbol{v}_0 \\ \beta(\boldsymbol{v}_0 \cdot \boldsymbol{v}_1) + \gamma(\boldsymbol{v}_1 \cdot \boldsymbol{v}_1) &= \boldsymbol{v}_2 \cdot \boldsymbol{v}_1, \end{aligned}$$

which can be solved with Cramer's rule, yielding two formulas for β and γ, and $\alpha = 1 - \beta - \gamma$. These barycentric coordinates are then used to interpolate the vertex attributes between the three vertexes of face f to determine their values at p.

Algorithm 4.4 3-Split

Input: Face f and point p.
Output: Interior splitting edges e_{i0}, e_{i1}, e_{i2} incident to p that subdivide f.

1: Acquire nodes $\langle n_0, n_1, n_2 \rangle$, vertexes $\langle v_0, v_1, v_2 \rangle$, and edges $\langle e_0, e_1, e_2 \rangle$ of face f according to Figure 4.9 (right).
2: $n_m \leftarrow$ MakeNode(p) ▷ Create midpoint node at p.
3: $v_m \leftarrow$ MakeVertex(v_0, v_1, v_2, p) ▷ Create midpoint vertex at p.
4: $e_{i0} \leftarrow$ MakeEdge(n_m, n_0) ▷ Create interior edge between n_m and n_0.
5: $e_{i1} \leftarrow$ MakeEdge(n_m, n_1) ▷ Create interior edge between n_m and n_1.
6: $e_{i2} \leftarrow$ MakeEdge(n_m, n_2) ▷ Create interior edge between n_m and n_2.
7: $f_{c0} \leftarrow$ MakeFace$(n_m, n_0, n_1, v_m, v_0, v_1)$ ▷ Create child face between n_m, n_0 and n_1.
8: $f_{c1} \leftarrow$ MakeFace$(n_m, n_1, n_2, v_m, v_1, v_2)$ ▷ Create child face between n_m, n_1 and n_2.
9: $f_{c2} \leftarrow$ MakeFace$(n_m, n_2, n_0, v_m, v_2, v_0)$ ▷ Create child face between n_m, n_2 and n_0.
10: RegisterEdge(e_{i0}, f_{c0}, f_{c2}) ▷ Set incident faces of edge e_{i0}.
11: RegisterEdge(e_{i1}, f_{c1}, f_{c0}) ▷ Set incident faces of edge e_{i1}.
12: RegisterEdge(e_{i2}, f_{c2}, f_{c1}) ▷ Set incident faces of edge e_{i2}.
13: RegisterFace$(c_{f_0}, e_{i0}, e_0, e_{i1})$ ▷ Set incident edges of face f_{c0}.
14: RegisterFace$(c_{f_1}, e_{i1}, e_1, e_{i2})$ ▷ Set incident edges of face f_{c1}.
15: RegisterFace$(c_{f_2}, e_{i2}, e_2, e_{i0})$ ▷ Set incident edges of face f_{c2}.
16: UpdateEdge(e_1, f, f_{c0}) ▷ Update face adjacency of edge e_0.
17: UpdateEdge(e_2, f, f_{c1}) ▷ Update face adjacency of edge e_1.
18: UpdateEdge(e_3, f, f_{c2}) ▷ Update face adjacency of edge e_2.
19: $\mathcal{F} \leftarrow \mathcal{F} - f$ ▷ Remove face f from mesh.
20: **return** Splitting edges e_{i0}, e_{i1}, e_{i2}.

The rest of the 3-Split algorithm is straightforward. Lines 4 to 15 create the three new edges and faces that p splits the face into, and sets their incidence relationships. The original exterior edges e_0, e_1, and e_2 must again be updated to reference the appropriate new child face instead of referencing f which is removed from the mesh. The three splitting edges e_{i0}, e_{i1}, e_{i2} are returned. These are used to determine which one of them coincides with e_c (in configurations 3, 7, and 9), and are used in configurations 6 and 8 to find out whether p_0 or p_1 lies in one of the child faces referenced by those edges.

4.3 Incision carving

Incision carving consists of two parts: opening the cut and generating the interior of the incision; the *cutting gutter*. Using the ordered set of cutting line edges $\mathcal{E}_\mathcal{C}$ constructed in the previous stage as input, the local geometry on either side of the cut is separated, simulating a cleft in the tissue. This action leaves a cavity in the mesh – temporarily violating the manifold property – which is then connected again with new topology that lies at a certain depth below the surface of the local geometry. Figure 4.10 illustrates the separated geometry and generated interior structure for the cutting line from Figure 4.7. The rest of this chapter describes these two processes in detail.

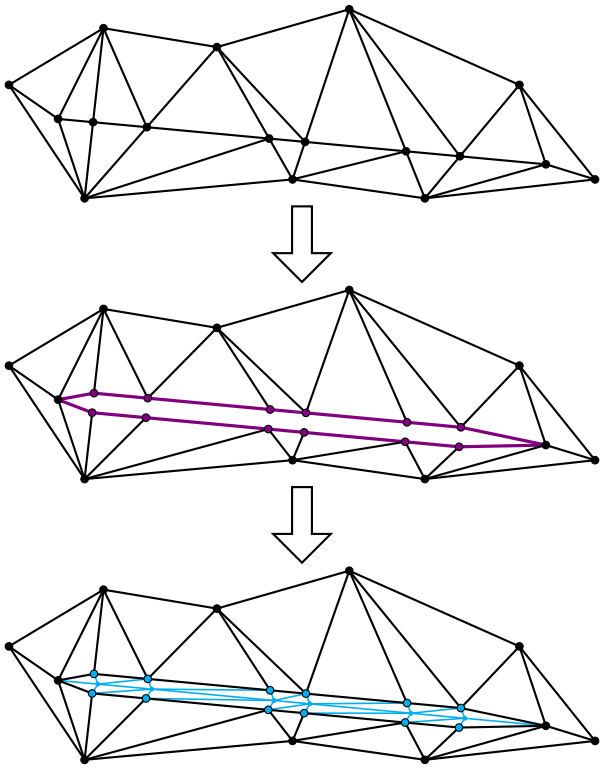


Figure 4.10: The incision carving process consists of opening the cutting line (middle) and generating the cutting gutter (bottom). New mesh elements that result from separating the upper geometry from the lower geometry have a purple color. Cutting gutter geometry located below the surface of the mesh is shown in blue.

4.3.1 Opening the cutting line

In order to cleave the incision, nodes and vertexes that lie on the cutting line must be duplicated and moved apart from each other. Connecting these elements with edges will form a border around the incision (shown in purple in Figure 4.10) that must be further connected to the topology above and

below the cutting line. The original nodes, vertices and edges that lie on the cutting line are then no longer referenced and can be removed.

4.3.1.1 Forming the border of the cut

Because the work presented in this thesis neglects soft body dynamics, the new positions of the nodes and vertexes on the border of the cut are displaced instantaneously instead of over time. This displacement is perpendicular to the cut in opposite directions and by equal amounts. Lim et al. [LJD07] describe that the width of the *cut opening displacement* – the maximum extent of the opening at the center of the cut – is influenced by the length and depth of the incision, and the pretension of the tissue. The results of their cut opening experiments suggest that the latter is much less influential than length and depth, and we will therefore ignore the pre-stress factor in our model.

The plot on the left-hand side of Figure 4.11 shows the measurements by Lim et al. of the cut opening displacement (COD) as a function of the length L and depth d of the cut. We have recorded interpreted discrete values for the lengths and cut opening displacements (in centimeters) for depths $d = 0.06cm$ and $d = 0.08cm$ in the table on the right.

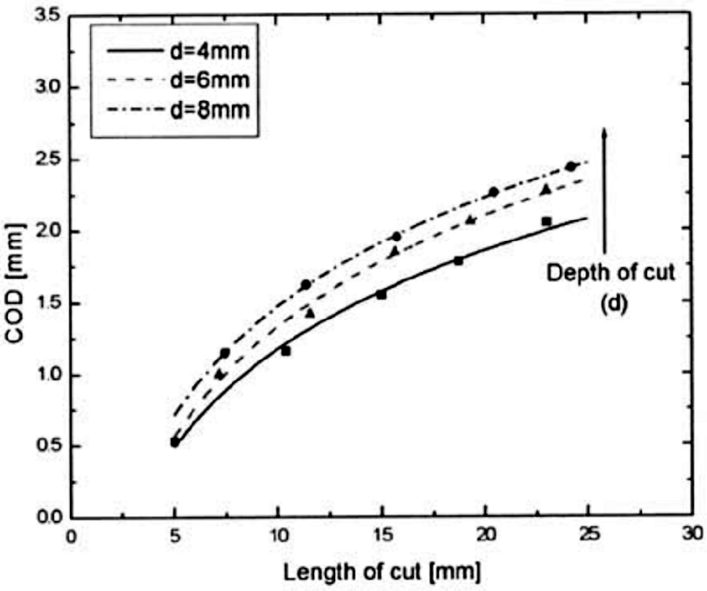
Because we want to support cutting lengths that are much longer than $0.3cm$, a function must be found that exceeds this range while still approximating the discrete data set. Using curve fitting techniques (least squares regression), it was found that a logarithmic function best approximates the shape of the curve. The following logarithmic functions approximate the maximum extent of the cut opening for cuts with respective depths of $0.06cm$ and $0.08cm$:

$$COD(L) = 0.0107\ln(L) + 0.0385 \quad \text{for } d = 0.06cm$$
$$COD(L) = 0.0109\ln(L) + 0.0400 \quad \text{for } d = 0.08cm,$$

where L is again the length of the cut in centimeters.

The shapes of these logarithmic functions suggest that a linear trend can be constructed for every additional $0.02cm$ of depth, yielding the following expression for $d = 0.10cm$:

$$COD(L) = 0.0111\ln(L) + 0.0415 \quad \text{for } d = 0.10cm.$$



L	COD (d=0.06cm)	COD (d=0.08cm)
0.025	0.0000	0.0000
0.050	0.0058	0.0073
0.075	0.0104	0.0116
0.100	0.0136	0.0150
0.125	0.0161	0.0175
0.150	0.0182	0.0193
0.175	0.0198	0.0209
0.200	0.0212	0.0224
0.225	0.0225	0.0238
0.250	0.0238	0.0250
0.275	0.0249	0.0261
0.300	0.0261	0.0272

Figure 4.11: Left: cut opening displacement measured as a function of the length and depth of the cut (from [LJD07]). Right: interpreted discrete values from the plot for length L and cut opening displacement for depths $d = 0.06cm$ and $d = 0.08cm$.

A general formula can be constructed that continues this same step for each 0.02cm, allowing us to compute the COD for any length L and depth d:

$$COD(L,d) = (0.0111 + 0.0002\left(\frac{d-0.1}{0.02}\right))\ln(L) + (0.0415 + (0.0015\left(\frac{d-0.1}{0.02}\right)).$$

The depth of the cut would usually be dependent on the cutting tool, but since this is emulated in our solution, d is arbitrarily computed as one-fifth of L and constrained to range $[0.1cm, 1.0cm]$.

Cut opening displacement COD_{p_i} at any point p_i on the cutting line is computed by modulating COD by the parabolic curve shown in Figure 4.12. Note that $|\mathcal{E}_\mathcal{C}|$ is the cardinality of the set of cutting line edges $\mathcal{E}_\mathcal{C}$. This yields the following equation for COD_{p_i}:

$$COD_{p_i} = m\ COD(L,d).$$

For every directed edge $e_c \in \mathcal{E}_\mathcal{C}$ the cut opening displacement is computed at two points: the origin p_i and the destination p_{i+1}, with $i = 0, \ldots, |\mathcal{E}_\mathcal{C}|$. Note that each destination p_{i+1} coincides with the origin p_i of the next edge e_c on the cutting line.

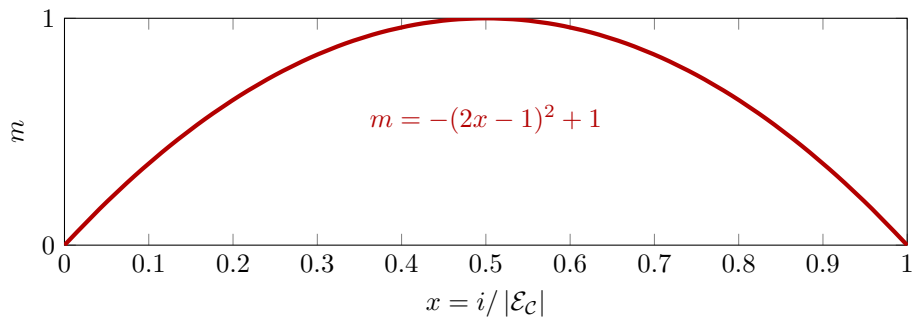


Figure 4.12: Modulating the cut opening displacement (COD) for all points of the cut. This parabola ensures that the COD is (near) maximal at the center point(s), zero at the first and last points, and smoothed out in-between. Variable i represents the index of point p_i on the cutting line with $i = 0, \ldots, |\mathcal{E}_\mathcal{C}|$.

One additional quantity is necessary to compute the positions of the new nodes and vertexes located around the border of the cut: a vector perpendicular to the direction of the cut. It can be constructed by calculating the cross product between two sides of the cutting quad Q. For quadrilateral $Q = \square Q_0Q_1Q_2Q_3$ with points given in clockwise order in accordance with Figure 4.4, we define its (normalized) upward vector as:

$$\hat{\boldsymbol{u}} = \frac{(Q_1 - Q_0) \times (Q_3 - Q_0)}{\|(Q_1 - Q_0) \times (Q_3 - Q_0)\|}.$$

Figure 4.13 illustrates the creation of a border around the cut by constructing new points at either side of the cutting line. Note that the first and last points of the cutting line must be kept unaltered and thus we have three cases for cutting line edge $e_c \in \mathcal{E}_\mathcal{C}$. For each intermediate point p_i on the cutting line two new points p_{0_u} and p_{0_l} are defined from the cut opening displacement and the upward vector defined above:

$$p_{i_u} = p_i + \frac{COD_{p_i}}{2}\hat{\boldsymbol{u}}$$
$$p_{i_l} = p_i - \frac{COD_{p_i}}{2}\hat{\boldsymbol{u}}$$

where p_{0_u} denotes the point located on the upper side of the e_c (with respect to $\hat{\boldsymbol{u}}$), and p_{0_l} denotes the lower point (see Figure 4.13). Note that COD_{p_i} describes the total width of the cut opening at point p_i; half this value corresponds to the displacement distance from p_i.

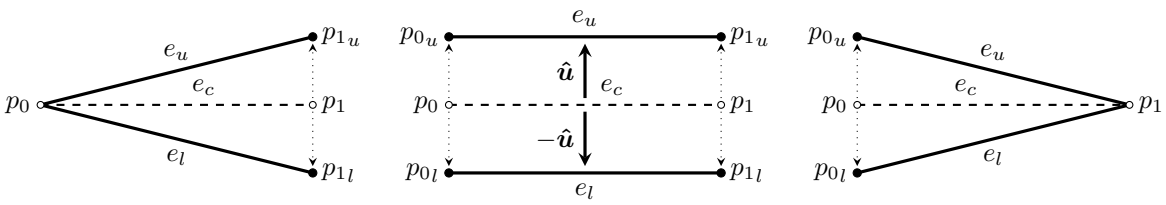


Figure 4.13: Opening (or cleaving) the cutting line. Open dots and dashed lines denote points and edges lying on the cutting line, and solid dots and lines indicate the new border created due to the cleaving process. Each point p_i on the cutting line is copied twice and displaced perpendicularly in direction $\hat{\boldsymbol{u}}$ and $-\hat{\boldsymbol{u}}$ at a distance of $COD_{p_i}/2$. The first and last edges of the cutting line (left and right) are handled separately from the intermediate edges (center).

The new geometric points that define the border of the cut opening are used to construct the mesh topology that divides the cutting line into two parts. Algorithm 4.5 describes how this is done for intermediate edges of the cutting line. The process is similar for the first and last edges, where the nodes and vertexes located at respectively p_0 and p_1 are reused instead of being duplicated and displaced (left and right image of Figure 4.13).

Algorithm 4.5 Constructing the geometry and topology of the cutting border.

Input: Ordered set of cutting line edges $\mathcal{E}_\mathcal{C}$.

1: **for each** intermediate edge $e_c \in \mathcal{E}_\mathcal{C}$ **do**
2: $v_0 \leftarrow \mathcal{V}[c_e \mapsto v_0]$ ▷ Vertex at the origin of e_c.
3: $v_1 \leftarrow \mathcal{V}[c_e \mapsto v_1]$ ▷ Vertex at the destination of e_c.
4: $p_{0_u} \leftarrow v_{0_p} + (COD_{p_0}/2)\hat{\boldsymbol{u}}$ ▷ Left upper border position.
5: $p_{1_u} \leftarrow v_{1_p} + (COD_{p_1}/2)\hat{\boldsymbol{u}}$ ▷ Left lower border position.
6: $p_{0_l} \leftarrow v_{0_p} - (COD_{p_0}/2)\hat{\boldsymbol{u}}$ ▷ Right upper border position.
7: $p_{1_l} \leftarrow v_{1_p} - (COD_{p_1}/2)\hat{\boldsymbol{u}}$ ▷ Right lower border position.
8: $n_{0_u} \leftarrow \textsc{MakeNode}(p_{0_u})$ ▷ Left lower border node.
9: $n_{0_l} \leftarrow \textsc{MakeNode}(p_{0_l})$ ▷ Left lower border node.
10: $n_{1_u} \leftarrow \textsc{MakeNode}(p_{1_u})$ ▷ Right upper border node.
11: $n_{1_l} \leftarrow \textsc{MakeNode}(p_{1_l})$ ▷ Right lower border node.
12: $v_{0_u} \leftarrow \textsc{MakeVertex}(p_{0_u}, v_{0_x}, v_{0_\eta}, v_{0_t}, v_{0_b})$ ▷ Left upper border vertex.
13: $v_{0_l} \leftarrow \textsc{MakeVertex}(p_{0_l}, v_{0_x}, v_{0_\eta}, v_{0_t}, v_{0_b})$ ▷ Left lower border vertex.
14: $v_{1_u} \leftarrow \textsc{MakeVertex}(p_{1_u}, v_{1_x}, v_{1_\eta}, v_{1_t}, v_{1_b})$ ▷ Right upper border vertex.
15: $v_{1_l} \leftarrow \textsc{MakeVertex}(p_{1_l}, v_{1_x}, v_{1_\eta}, v_{1_t}, v_{1_b})$ ▷ Right lower border vertex.
16: $e_u \leftarrow \textsc{MakeEdge}(n_{0_u}, n_{1_u})$ ▷ Upper border edge for e_c.
17: $e_l \leftarrow \textsc{MakeEdge}(n_{0_l}, n_{1_l})$ ▷ Lower border edge for e_c.
18: Add n_{0_u} and n_{1_u} to $\mathcal{N}_\mathcal{U}$, and n_{0_l} and n_{1_l} to $\mathcal{N}_\mathcal{L}$.
19: Add v_{0_u} and v_{1_u} to $\mathcal{V}_\mathcal{U}$, and v_{0_l} and v_{1_l} to $\mathcal{V}_\mathcal{L}$.
20: Add e_u to $\mathcal{E}_\mathcal{U}$ and e_l to $\mathcal{E}_\mathcal{L}$.
21: Add $[c_e \mapsto f_0]$ to $\mathcal{F}_\mathcal{U}$ and $[c_e \mapsto f_1]$ to $\mathcal{F}_\mathcal{L}$.

For each edge e_c the vertices located at the origin and destination of that edge are acquired first. Recall that these vertex references were stored in e_c during the cutting line fusion process. Next, in lines 4 to 7 we compute the four new positions located above and below e_c due to the the cut opening displacement. These are consequently used in the definition of the upper and lower nodes and vertexes that lie on the border of the cut (lines 8 to 15). The vertex definitions reuse the texture coordinates, normals, tangents, and bitangents of vertexes v_0 and v_1. This will ensure that the mesh parameterization is unaffected by the splitting operation. Upper edge e_u and lower edge e_l are constructed between the upper nodes and the lower nodes, respectively.

Finally, in addition to adding the new mesh elements to the mesh, they are also added to ordered collections for easy retrieval during the gutter generation process. These ordered sets are defined as

follows: upper nodes $\mathcal{N}_\mathcal{U}$, lower nodes $\mathcal{N}_\mathcal{L}$, upper vertexes $\mathcal{V}_\mathcal{U}$, lower vertexes $\mathcal{V}_\mathcal{L}$, upper edges $\mathcal{E}_\mathcal{U}$, lower edges $\mathcal{E}_\mathcal{L}$, upper faces $\mathcal{F}_\mathcal{U}$, and lower faces $\mathcal{F}_\mathcal{L}$. Note that faces f_0 and f_1 of edge e_c were already defined as being located respectively above and below e_c in the fusion process, and can thus be directly added to either $\mathcal{F}_\mathcal{U}$ or $\mathcal{F}_\mathcal{L}$ on line 21.

4.3.1.2 Updating local mesh topology

Creating the border topology is only half the work; the new topology remains to be connected with the rest of the mesh. This amounts to updating the references of the upper and lower faces and any edge that references a node of the cutting line. We must therefore find all the upper and lower faces that are incident to the cutting line. Note that the sets $\mathcal{F}_\mathcal{U}$ and $\mathcal{F}_\mathcal{L}$ defined earlier only contain those faces that reference any cutting line edge $e_c \in \mathcal{E}_\mathcal{C}$, while there may also be adjacent faces that merely reference a node of the cutting line. Figure 4.14 shows how these directly and indirectly incident faces are defined for the example surface mesh seen at the start of this section.

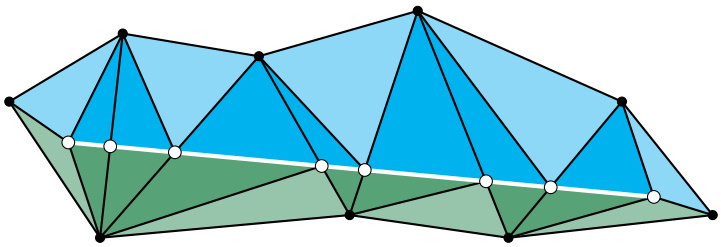


Figure 4.14: Faces directly and indirectly incident to the cutting line. Edges and points of the cutting line are highlighted in white. Deep blue and green: faces part of respectively the set of upper faces $\mathcal{F}_\mathcal{U}$ and the set of lower faces $\mathcal{F}_\mathcal{L}$ considered to be directly incident to the cutting line that store a reference to one of its edges. Light blue and green: upper and lower faces considered to be indirectly incident to the cutting line that store a reference to one of its nodes.

Starting with sets $\mathcal{F}_\mathcal{U}$ and $\mathcal{F}_\mathcal{L}$ of the upper and lower faces that are incident to the edges of the cutting line, we want to construct two new sets $\mathcal{F}_\mathcal{U}\prime$ and $\mathcal{F}_\mathcal{L}\prime$ that also include all the faces that are indirectly adjacent to the cutting line, that is, they reference a node-vertex pair that lies on the cutting line. The construction of $\mathcal{F}_\mathcal{U}\prime$ is outlined in Algorithm 4.6, and $\mathcal{F}_\mathcal{L}\prime$ is constructed in a similar manner.

Algorithm 4.6 Finding all upper and lower faces adjacent to the cutting line.

Input: Sets of direct upper faces $\mathcal{F}_\mathcal{U}$ and lower faces $\mathcal{F}_\mathcal{L}$.
Output: Sets of all (direct and indirect) upper faces $\mathcal{F}_\mathcal{U}\prime$ and lower faces $\mathcal{F}_\mathcal{L}\prime$.

```
 1: F_U' ← F_U                                  ▷ Add all elements from F_U to F_U'.
 2: F_L' ← F_L                                  ▷ Add all elements from F_L to F_L'.
 3: for each face f_u ∈ F_U do
 4:     AddUpperFace([f_u ↦ f_nb0])             ▷ Face neighboring f_u as referenced by e_0.
 5:     AddUpperFace([f_u ↦ f_nb1])             ▷ Face neighboring f_u as referenced by e_1.
 6:     AddUpperFace([f_u ↦ f_nb2])             ▷ Face neighboring f_u as referenced by e_2.
 7: function AddUpperFace(f)
 8:     if (f ∉ F_U') and (f ∉ F_L') then       ▷ Continue if face f is not in F_U' or F_L'.
 9:         for each edge e_c ∈ E_C do          ▷ Iterate over the edges of the cutting line.
10:             for each vertex v ∈ [f ↦ v_1,2,3] do   ▷ Iterate over the vertexes referenced by f.
11:                 if v = [e_c ↦ v_0] or v = [e_c ↦ v_1] then   ▷ Test if vertex v is referenced by e_c.
12:                     F_U ← F_U + f           ▷ Add face f to set of upper faces.
13:                     AddUpperFace([f ↦ f_nb0])   ▷ Continue with the neighbors of f.
14:                     AddUpperFace([f ↦ f_nb1])
15:                     AddUpperFace([f ↦ f_nb2])
16:                     break                   ▷ Discard testing remaining vertexes.
```

First the elements of $\mathcal{F}_\mathcal{U}$ are copied to $\mathcal{F}_\mathcal{U}\prime$, and likewise for the lower faces. Then, for each face $f_u \in \mathcal{F}_\mathcal{U}$ we acquire its three neighboring faces by following the edges of f_u. Each of these edges contain a reference to f_u and a reference to another face f_{nb} which is its neighbor. If either one of these neighboring faces is already stored in $\mathcal{F}_\mathcal{U}$, the rest of the procedure can be skipped. Otherwise we iterate over each edge e_c of the cutting line to determine whether one its vertex references corresponds to one of the vertex references of face f. If this is the case then f is added to $\mathcal{F}_\mathcal{U}\prime$, and the process is repeated for the neighboring faces of f. Knowing that at least one vertex of f lies on the cutting line is sufficient; the break operation prevents unnecessary further tests.

Note that if the vertex references of face f and edge e_c correspond, their nodes do as well, while this does not always hold vice versa. This follows from the definition of a vertex that is uniquely defined by its position as well as its attributes, compared to just the position of a node. One of the main reasons for including vertex references in the *edges of the cutting line* was to be able to handle this distinction.

One final remark about the faces indirectly incident to the first and last points of the cutting line. Although Figure 4.14 shows these as both blue and green, in truth the presented algorithm will add all of these as in the set of upper faces. This is simply a result of computing $\mathcal{F}_\mathcal{L}\prime$ after $\mathcal{F}_\mathcal{U}\prime$, at which point these faces have already been added to the set of upper faces. In practice this makes no difference because the nodes and vertices at the first and last points remain untouched.

Now that the faces adjacent to the cutting line have been accumulated, the cut is opened by replacing any references to the cutting line topology with references to the cutting border topology. This is relatively straightforward due to the data sets that we have constructed earlier. The reference updating process is described in Algorithm 4.7, where for each edge on the cutting line we determine whether any of the references of faces $f \in \mathcal{F}_\mathcal{U}\prime$ require updating.

Algorithm 4.7 Updating cutting line references with cutting border references for incident faces.

Input: Cutting line edges $\mathcal{E}_\mathcal{C}$, incident faces $\mathcal{F}_\mathcal{U}\prime, \mathcal{F}_\mathcal{L}\prime$, and border topology $\mathcal{N}_\mathcal{U}, \mathcal{N}_\mathcal{L}, \mathcal{V}_\mathcal{U}, \mathcal{V}_\mathcal{L}, \mathcal{E}_\mathcal{U}, \mathcal{E}_\mathcal{L}$.

```
i ← 0,   j ← 0                                          ▷ Indexes for border topology data.
for each e_c ∈ E_C do                                   ▷ Iterate over cutting line edges.
    n_0c ← [e_c ↦ n_0]                                  ▷ Acquire cutting line nodes.
    n_1c ← [e_c ↦ n_1]
    v_0c ← [e_c ↦ v_0]                                  ▷ Acquire cutting line vertexes.
    v_1c ← [e_c ↦ v_1]
    for each face f ∈ F_U' do                           ▷ Iterate over incident upper faces.
        for each k ∈ {0, 1, 2} do                       ▷ Iterate over references of f.
            if [f ↦ n_k] = n_0c then [f ↦ n_k] ← N_Uj   ▷ Update face-node references.
            if [f ↦ n_k] = n_1c then [f ↦ n_k] ← N_Uj+1
            if [f ↦ v_k] = v_0c then [f ↦ v_k] ← V_Uj   ▷ Update face-vertex references.
            if [f ↦ v_k] = v_1c then [f ↦ v_k] ← V_Uj+1
            if [f ↦ e_k] = e_c then                     ▷ Update face-edge reference.
                [E_Ui ↦ f_0] ← f                        ▷ Set edge-face reference.
                [f ↦ e_k] ← E_Ui                        ▷ Set face-edge reference.
            if [f ↦ e_k ↦ n_0] = n_0c then [f ↦ e_k ↦ n_0] ← N_Uj   ▷ Update edge-node references.
            if [f ↦ e_k ↦ n_1] = n_0c then [f ↦ e_k ↦ n_1] ← N_Uj
            if [f ↦ e_k ↦ n_0] = n_1c then [f ↦ e_k ↦ n_0] ← N_Uj+1
            if [f ↦ e_k ↦ n_1] = n_1c then [f ↦ e_k ↦ n_1] ← N_Uj+1
    i ← i + 1,   j ← j + 2                              ▷ Increment indexes.
```

By iterating over each edge e_c of the cutting line, its nodes and vertexes can be easily accessed. Indexes i and j maintained during the loop are used to access the border edges and node-vertex pairs respectively. Because we stored two nodes and vertexes per edge, j is incremented by two in each iteration. It is unclear which of the references of a face correspond to the topology of the cutting line (nodes, vertices, and edges), and thus we iterate over all of them and replace each reference as soon as a match is found. For nodes and vertices, either one or two references need to be updated, and these correspond to the node and vertex located at either p_0 or at p_1 of e_c.

In the case of a directly incident face, there is one edge reference that corresponds to e_c (lines 13-15). We must replace this reference by the new upper border edge defined in set $\mathcal{E}_{\mathcal{U}}$. This border edge did not have any face references assigned to it in Algorithm 4.5 because they were still unknown at that time. This is rectified here by registering face f as one of its adjacencies (line 14). The other reference will be registered during the creation of the cutting gutter. Furthermore, for directly or indirectly incident faces we have at most three edges that still store reference to nodes of the cutting line (lines 16-19). Like the face-node references each one is replaced by the nodes of the (upper) border topology.

Note that the process of updating the lower topology of the cutting border has been omitted in Algorithm 4.7. This is very similar to the loop starting on line 7, but instead we iterate over $\mathcal{F}_{\mathcal{L}'}$ and replace references with the lower border topology stored in sets $\mathcal{N}_{\mathcal{L}}$, $\mathcal{V}_{\mathcal{L}}$, and $\mathcal{E}_{\mathcal{L}}$.

4.3.2 Generating the cutting gutter

Constructing the cutting gutter – the inner geometry of the cut – is done similarly to how the border geometry was created, but now the new points are created below the surface of the cutting line. Again the process is divided into two phases: defining the geometry of the cutting gutter (the positions and attributes of its points), and constructing the gutter topology and properly connecting it to the mesh.

4.3.2.1 Gutter geometry and attributes

As previously mentioned, the depth d of the cut is computed as

$$d = \max(0.1, \min(1.0, L/5)),$$

where L is the length of the cut. It is used in combination with the inward direction of the cut to compute the positions of the inner geometry. Like the upward direction, the inward direction $\hat{\boldsymbol{\imath}}$ is computed from cutting quadrilateral $Q = \square Q_0 Q_1 Q_2 Q_3$:

$$\hat{\boldsymbol{\imath}} = \frac{Q_1 - Q_0}{\|Q_1 - Q_0\|}.$$

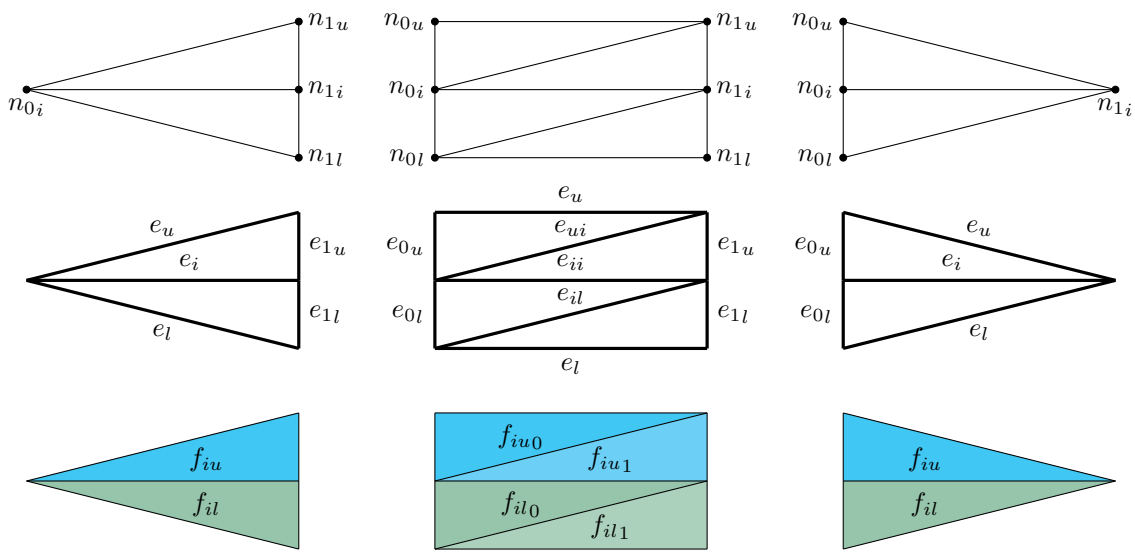


Figure 4.15: Upper, inner, and lower mesh elements for the cutting gutter topology. Left to right: the first edge, intermediate edges, and last edge of the cutting line. Top to bottom: node (and vertex), edge, and face elements of the cutting gutter.

Positions p_{0_i} and p_{1_i} of the interior of the cut for a given cutting edge e_c are then computed as

$$p_{0_i} = p_0 + d\hat{\boldsymbol{\imath}}$$
$$p_{1_i} = p_1 + d\hat{\boldsymbol{\imath}},$$

where p_0 and p_1 are the positions of the origin and destination of e_c. Like previously, we do not define new points at the first and last points of the cutting line, which again means that the first and last edges are handled separately from the interior edges of $\mathcal{E}_\mathcal{C}$. Both this and the invariable depth of the cut govern the geometric appearance of the cutting gutter, which is chosen arbitrarily as a proof of concept.

Figure 4.15 (top) shows the nodes associated with the cutting gutter. Note that only the two inner nodes n_{0_i} and n_{1_i} located at respectively p_{0_i} and p_{1_i} have to be defined. For the upper and lower nodes of the gutter topology we can reuse the corresponding upper and lower nodes of the border topology.

For the vertices of the cutting gutter, located at the upper positions p_{0_u}, p_{1_u} and lower positions p_{0_l}, p_{1_l}, we cannot simply reuse the cutting border vertex definitions because of deviating texture coordinates. This arises due to the parameterization of the cutting gutter, which is disjoint from that of the surface mesh. Concretely, this means that the texture for the cutting gutter is packed into the color map side by side with the skin surface texture. See also Figure 5.2.

As the other texture maps use the same texture mapping, they too must reserve the same (u, v)-space for the cutting gutter. Because (u, v) coordinates of a texture map range from 0 to 1, a parameterization of any rectangular area of a texture can be acquired by dividing the column and row indexes of its top-left and bottom-right corners by the width W and height H (in pixels) of the texture map. See Figure 4.16. The minimum and maximum (u, v) coordinates of the gutter texture are therefore given by:

$$u_{min} = \frac{i_0}{W}, \qquad v_{min} = \frac{j_0}{H}, \qquad u_{max} = \frac{i_1}{W}, \qquad v_{max} = \frac{j_1}{H}.$$

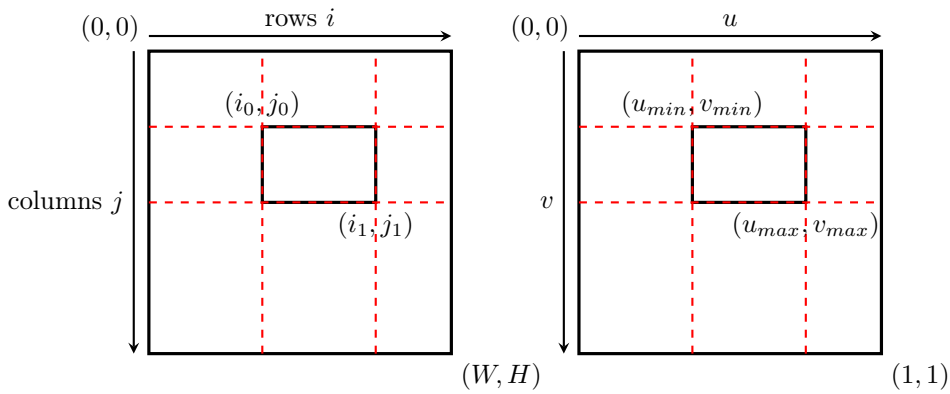


Figure 4.16: A rectangular subarea of a texture map can either be expressed by the i-th row and j-th column of its top-left and bottom-right corners (left image), or by the (u, v)-coordinates of those corners (right image).

In our test application, the gutter texture is a patch of 256×128 texels with its top-left corner at $(i_0 = 0, j_0 = 0)$ in a 4096px $\times$ 4096px texture map. We map the top of the gutter texture patch to the vertices located around the border of the cut, and the bottom of this patch to the inner vertices located at p_{0_i} and p_{1_i}. The span of the gutter patch (with length $u_{max} - u_{min}$) will be mapped across the vertices of the cutting line, from its first to last point. Because the amount of points of the cutting line is variable, we split the total span of the u-coordinate into equal steps:

$$u_{step} = (u_{max} - u_{min}) / |\mathcal{E}_\mathcal{C}|,$$

where $|\mathcal{E}_\mathcal{C}|$ is equal to the index of the last point of the cutting line. This allows us to compute, for each edge e_c, the border texture coordinates x_{0_b} and x_{1_b}, and the inner texture coordinates x_{0_i} and x_{1_i}.

With this missing piece of information we can finally construct the upper, lower, and inner vertices of the cutting gutter. Algorithm 4.8 shows how Algorithm 4.5 is extended to account for the nodes and vertices of the cutting gutter. Once again only the process for intermediate edges is shown; the process for the first and last edges differs only in that their respective first and last points are not displaced.

Algorithm 4.8 Constructing the nodes and vertexes of the cutting gutter.

1: $i \leftarrow 0$ ▷ Edge indexer.
2: **for each** intermediate edge $e_c \in \mathcal{E}_\mathcal{C}$ **do**
3: ▷ Execute lines 2-21 of Algorithm 4.5.
4: $p_{0_i} \leftarrow p_0 + d\hat{\boldsymbol{\imath}}$ ▷ Left inner gutter position.
5: $p_{1_i} \leftarrow p_1 + d\hat{\boldsymbol{\imath}}$ ▷ Right inner gutter position.
6: $x_{0b} \leftarrow (u_{min} + (i)u_{step}, v_{min})$ ▷ Left border texture coordinate.
7: $x_{1b} \leftarrow (u_{min} + (i+1)u_{step}, v_{min})$ ▷ Right border texture coordinate.
8: $x_{0i} \leftarrow (u_{min} + (i)u_{step}, v_{max})$ ▷ Left inner texture coordinate.
9: $x_{1i} \leftarrow (u_{min} + (i+1)u_{step}, v_{max})$ ▷ Right inner texture coordinate.
10: $n_{0i} \leftarrow \textsc{MakeNode}(p_{0_i})$ ▷ Left inner gutter node.
11: $n_{1i} \leftarrow \textsc{MakeNode}(p_{1_i})$ ▷ Right inner gutter node.
12: $w_{0u} \leftarrow \textsc{MakeVertex}(p_{0_u}, x_{0b}, v_{0\eta}, v_{0t}, v_{0b})$ ▷ Left upper gutter vertex.
13: $w_{0i} \leftarrow \textsc{MakeVertex}(p_{0_i}, x_{0i}, v_{0\eta}, v_{0t}, v_{0b})$ ▷ Left inner gutter vertex.
14: $w_{0l} \leftarrow \textsc{MakeVertex}(p_{0l}, x_{0b}, v_{0\eta}, v_{0t}, v_{0b})$ ▷ Left lower gutter vertex.
15: $w_{1u} \leftarrow \textsc{MakeVertex}(p_{0_u}, x_{1b}, v_{1\eta}, v_{1t}, v_{1b})$ ▷ Right upper gutter vertex.
16: $w_{1i} \leftarrow \textsc{MakeVertex}(p_{0_i}, x_{1i}, v_{1\eta}, v_{1t}, v_{1b})$ ▷ Right inner gutter vertex.
17: $w_{1l} \leftarrow \textsc{MakeVertex}(p_{0l}, x_{1b}, v_{1\eta}, v_{1t}, v_{1b})$ ▷ Right lower gutter vertex.
18: Add n_{0i} and n_{1i} to $\mathcal{N}_\mathcal{I}$.
19: Add w_{0u} and w_{1u} to $\mathcal{W}_\mathcal{U}$.
20: Add w_{0i} and w_{1i} to $\mathcal{W}_\mathcal{I}$.
21: Add w_{0l} and w_{1l} to $\mathcal{W}_\mathcal{L}$.
22: $i \leftarrow i + 1$

Texture coordinates for the border and inner vertices are computed on lines 6-9. Note how the index of edge e_c in $\mathcal{E}_\mathcal{C}$ is used to determine the u-coordinate of the cutting gutter texture patch for the vertices corresponding to p_0 and p_1 of e_c. These gutter vertices are defined on lines 12-17. Note that each vertex definition reuses the normal, tangent, and bitangent vectors defined for the original cutting line vertices. Although this is not entirely correct if we want to maintain a smooth vector field of surface normals (for Phong interpolation [Pho75]), the visual impact is minimal and we will therefore ignore this shortsightedness.

4.3.2.2 Gutter topology

The final part of generating the cutting gutter consists of creating the remaining edge and face elements illustrated in Figure 4.15 (middle and bottom), configuring their incidence relationships, and connecting the gutter topology to the rest of the surface mesh. Again we will only discuss the process for intermediate cutting line edges, which is the most complex of the three cases.

Algorithm 4.9 describes the process of generating an intermediate gutter structure. For each edge of the cutting line its associated gutter nodes and vertices that were constructed in Algorithm 4.8 are retrieved. Their naming conventions follow those used in Figure 4.15. Edges e_u and e_l located at the top and bottom of the gutter structure have already been created as part of the border topology. For the remainder we use functions MakeEdge and MakeFace to add a new edge or face definition to $\mathcal{E}$ and $\mathcal{F}$ respectively, or retrieve an existing one if the definition already existed.

Edge e_{ii} between the two inner nodes n_{0i} and n_{1i} delimits the maximum depth of the gutter. To conform with the triangulation property of the mesh, two diagonal edges are also defined: one from the upper edge to the inner edge, e_{ui}, and one from the inner edge to the lower edge, e_{il}. For the sides of the intermediate gutter structure we have two edges connecting the inner points to the upper points, and two edges connecting the inner points to the lower points (lines 6-9). We also have four faces connecting

Algorithm 4.9 Generating the cutting gutter.

Input: Sets of cutting line edges $\mathcal{E}_\mathcal{C}$, upper edges $\mathcal{E}_\mathcal{U}$, lower edges $\mathcal{E}_\mathcal{L}$, upper nodes $\mathcal{N}_\mathcal{U}$, inner nodes $\mathcal{N}_\mathcal{I}$, lower nodes $\mathcal{N}_\mathcal{L}$, upper gutter vertexes $\mathcal{W}_\mathcal{U}$, inner gutter vertexes $\mathcal{W}_\mathcal{I}$, and lower gutter vertexes $\mathcal{W}_\mathcal{L}$.

1: **for each** intermediate edge $e_c \in \mathcal{E}_\mathcal{C}$ **do**
2: ▷ Acquire nodes and vertexes according to Figure 4.15.
3: $e_{ui} \leftarrow \textsc{MakeEdge}(n_{1u}, n_{0i})$ ▷ Upper-to-inner cross edge.
4: $e_{ii} \leftarrow \textsc{MakeEdge}(n_{0i}, n_{1i})$ ▷ Inner-to-inner cross edge.
5: $e_{il} \leftarrow \textsc{MakeEdge}(n_{1i}, n_{0l})$ ▷ Lower-to-inner cross edge.
6: $e_{0u} \leftarrow \textsc{MakeEdge}(n_{0i}, n_{0u})$ ▷ Left upper inner-to-border edge.
7: $e_{0l} \leftarrow \textsc{MakeEdge}(n_{0i}, n_{0l})$ ▷ Left lower inner-to-border edge.
8: $e_{1u} \leftarrow \textsc{MakeEdge}(n_{1i}, n_{1u})$ ▷ Right upper inner-to-border edge.
9: $e_{1l} \leftarrow \textsc{MakeEdge}(n_{1i}, n_{1l})$ ▷ Right lower inner-to-border edge.
10: $f_{iu0} \leftarrow \textsc{MakeFace}(n_{0u}, n_{1u}, n_{0i}, w_{0u}, w_{1u}, w_{0i})$ ▷ Top-left upper inner face.
11: $f_{iu1} \leftarrow \textsc{MakeFace}(n_{0i}, n_{1u}, n_{1i}, w_{0i}, w_{1u}, w_{1i})$ ▷ Bottom-right upper inner face.
12: $f_{il0} \leftarrow \textsc{MakeFace}(n_{0i}, n_{1i}, n_{0l}, w_{0i}, w_{1i}, w_{0l})$ ▷ Top-left lower inner face.
13: $f_{il1} \leftarrow \textsc{MakeFace}(n_{0l}, n_{1i}, n_{1l}, w_{0l}, w_{1i}, w_{1l})$ ▷ Bottom-right lower inner face.
14: $[e_u \mapsto f_1] \leftarrow f_{iu0}$ ▷ Set second edge-face reference for upper edge.
15: $[e_l \mapsto f_1] \leftarrow f_{il1}$ ▷ Set second edge-face reference for lower edge.
16: $\textsc{RegisterEdge}(e_{0u}, f_{iu0})$ ▷ Edge-face references for inner-to-border edges.
17: $\textsc{RegisterEdge}(e_{0l}, f_{il0})$
18: $\textsc{RegisterEdge}(e_{1u}, f_{iu1})$
19: $\textsc{RegisterEdge}(e_{1l}, f_{il1})$
20: $\textsc{RegisterEdge}(e_{ui}, f_{iu0}, f_{iu1})$ ▷ Edge-face references for cross edges.
21: $\textsc{RegisterEdge}(e_{ii}, f_{iu1}, f_{il0})$
22: $\textsc{RegisterEdge}(e_{il}, f_{il0}, f_{il1})$
23: $\textsc{RegisterFace}(f_{iu0}, e_u, e_{ui}, e_{0u})$ ▷ Face-edge references for gutter faces.
24: $\textsc{RegisterFace}(f_{iu1}, e_{ui}, e_{1u}, e_{ii})$
25: $\textsc{RegisterFace}(f_{il0}, e_{ii}, e_{il}, e_{0l})$
26: $\textsc{RegisterFace}(f_{il1}, e_{il}, e_{1l}, e_l)$

the nodes and vertices of the gutter structure (lines 10-13). Once again their node and vertex references are stored in clockwise winding order to preserve the mesh orientation.

Registering the incidence references between edges and faces is relatively straightforward. For upper edge e_u and lower edge e_l, whose first face reference was set on line 14 of Algorithm 4.7, we set respectively f_{iu0} and f_{il1} as their second face reference (lines 14-15). Moreover, each of the four side edges references one of the gutter faces. Because these edges are shared between adjacent gutter structures, we use function RegisterEdge to ensure that a given face reference is set as $[e \mapsto f_0]$ on the first encounter and as $[e \mapsto f_1]$ the second time any given edge e is encountered (lines 16-19). For the inner and diagonal edges both face references are directly available, and thus set simultaneously (lines 20-22). For face-edge references care must be taken that the winding order of its edge references corresponds with that of its node references (lines 23-26).

Finally, at the end of this process, we can remove the nodes, vertices, and edges of the cutting line, with the exception of the first and last points. The incision carving process has thus transformed this chain of edges into a mesh cavity whose border is subsequently connected to a subsurface gutter structure, thus creating an incision in the mesh.

Closing remarks

In this chapter we presented our mesh cutting simulation from start to finish. First we used ray casting to translate a point on the viewing window into a mesh surface point. A cutting line was formed by constructing a quadrilateral from two selected points and performing intersection tests with the mesh edges in order to find intersection points. Accounting for mesh parameterization, the cutting line was merged into the mesh topology by identifying all possible edge-face configurations and applying elementary 2-split and 3-split operations. We then opened the cutting line by creating new mesh elements that define its border, and connecting them with the existing mesh. Finally, an interior geometry that represents the groove of the cut was generated and connected to the cut border.

Our main contribution is a detailed description of the techniques required to create a mesh cut, giving special attention to keeping the mesh topology and parameterization intact. Our cutting scheme is able to handle all configurations that occur when working with a triangular mesh by performing basic remeshing operations. Combined with the previous chapter we have given a complete overview of the data structures and algorithms required to synthesize cutting wounds into surface meshes.

An advantage of our solution is that it is minimally intrusive with respect to the mesh topology due to only modifying the faces incident to the cutting line and those directly surrounding it. Additionally, because we use fundamental data structures and operations we expect that the method can easily be adjusted to support different types of wounds. Similarly, the method of defining a cut can be effortlessly changed to a different approach as long as it produces a cutting line on the surface of the mesh.

Of course our method has a number of drawbacks as well. Most crucially, accuracy problems when determining whether two points (nodes or vertices) are coincident can lead to topological errors, producing an invalid mesh. This operation, used internally in the MakeNode and MakeVertex functions and elsewhere, uses a threshold value ϵ to determine whether the distance between points is small enough for them to be considered to be coincident. The problem is with choosing the threshold value: if ϵ is too large two points that should be separate are considered to be coinciding, and if ϵ is too small two points may not be considered to be coinciding due to floating-point precision issues. The threshold value must thus be adapted based on the vertex density of the target mesh and other such factors.

Another drawback is that our current solution does not support progressive cutting or cutting over previous cuts. Doing so would require us to account for the current location of a cutting tool, and mesh elements would require additional quantities to trigger remeshing actions. The same can be said for the addition of soft body dynamics for the purpose of simulating mesh deformation. Another drawback that occurs due to the omission of a cutting tool is that the width and depth of a cut cannot be based on its dimensions and are thus somewhat arbitrary.

Finally, besides remeshing the faces incident to the cutting line, no additional mesh refinement is performed. This may positively affect the performance of the method, but will cause the appearance of the cut geometry to be governed by the underlying mesh geometry.

5. Visualizing cutting wounds

Aside from modifying the geometry of the mesh, synthesizing skin injuries also involves visualizing the wound. We visualize the uninjured skin surface with the skin renderer described in Appendix B. Cutting wounds are then visualized by making local modifications to the texture maps of the mesh that are used by the renderer, as well as introducing a new texture map that alters the appearance of subsurface scattering.

A difficulty arises due to the limited research into the appearance of injured skin in computer graphics. In game technology and computer graphics it is common to base the appearance of natural phenomena on reference images in cases where more rigorous (mathematical or physical) models are unavailable [Ebe03]. As such, the visualization of wounds is currently highly subjective to artistic input. We will therefore put more emphasis on the visualization techniques than on the aesthetics of the wounds.

In this chapter, we will first discuss the appearance of cutting wounds and how we attempt to reproduce them (Section 5.1). We will then go on to describe the visualization of the cutting gutter (Section 5.2), followed by the visualization of the skin surface bordering the cut (Section 5.3). Finally, we describe our adjustments to the skin renderer to simulate local skin discoloration (Section 5.4).

5.1 Reproducing the appearance of cutting wounds

For most types of injuries and materials the appearance of the surface immediately surrounding the wound is significantly changed. Although cutting wounds like scratches and incisions are generally cleaner than other types of wounds such as bruises and lacerations, they do feature skin discoloration [SSF06]. In the absence of any tangible research of the appearance of these discolorations, we have chosen to determine the visualization of injured skin based on the appearance of photographic material.

Figure 5.1 shows a number of reference images. Some common visual features of cutting wounds can be discerned:

1. The center of the cut exhibits a deep red; the color of the subdermal tissue.
2. The skin immediately surrounding the ridge exhibits a pinkish hue due to inflammation.
3. The skin further from the wound is reddish due to the dilation of blood vessels (erythema). This reddish color fades off with distance from the cut in a highly irregular pattern, blending more and more into the natural color of the skin.

Although the physical nature of the cut and the physiological nature of the affected area appear to have an influence on the exact appearance of the discoloration, we consider this to be outside the scope of this thesis.

We describe an approach to procedurally generate a texture patch at runtime whose appearance resembles those of reference photos. This so-called wound patch is then mapped onto the color map of the mesh. A blending approach is used in an attempt to reduce the synthetic appearance of the generated texture. Note that both the generation process and the appearance of the discoloration are constrained by a specific artistic view, that is, a different view would require a different generation process.

Figure 5.1: Various photographic references of cutting wounds.

An alternative method would be to directly map reference photographs or pregenerated images onto the mesh. There are a number of reasons why we have avoided this approach:

1. There is a severe lack of suitable reference photographs in the public domain, especially for modeling cuts without bloodied tissue. Also, many images are extremely graphic and thus not appropriate for use in games or surgical simulators.
2. As we have seen in the work by Lee et al. [LC10, LLC11], directly mapping reference material onto an arbitrary mesh leads to incredibly artificial appearances. We expect that this could be alleviated by using blending techniques, but not for all cases. There is an additional mapping issue to consider when the dimensions of a reference image do not directly line up with the dimensions of the cut.
3. We want to be able to support a large range of variation and flexibility in the visualization, which we believe can be better achieved by using procedural generation techniques.
4. We want to allow wounds to be visualized programmatically based on a mathematical model. Although to our knowledge such a model does not yet exist, it can only be supported by eschewing pregenerated content.

Concretely, in our wound visualization we visualizing the cutting gutter and the mesh surface surrounding a cut. The latter process consists of a number of stages. In the first stage a shader generates a texture of varying size and appearance that visualizes the cutting wound. This texture is then mapped onto the mesh by using a texture painting technique. To do this we first have to accumulate the mesh faces that lie in a certain radius around the cutting line. A shader will then render the wound patch to the appropriate mesh faces.

5.2 Visualizing the cutting gutter

The visualization of the interior injury, or the cutting gutter, was briefly mentioned in the previous chapter. A rectangular area of 256×128 pixels used to render the interior of the wound is packed into the color map of the target mesh. The texture coordinates corresponding to this rectangle are assigned to the vertices of the cutting gutter. The left side of Figure 5.2 shows how the cutting gutter patch is packed into the color map of our test mesh.

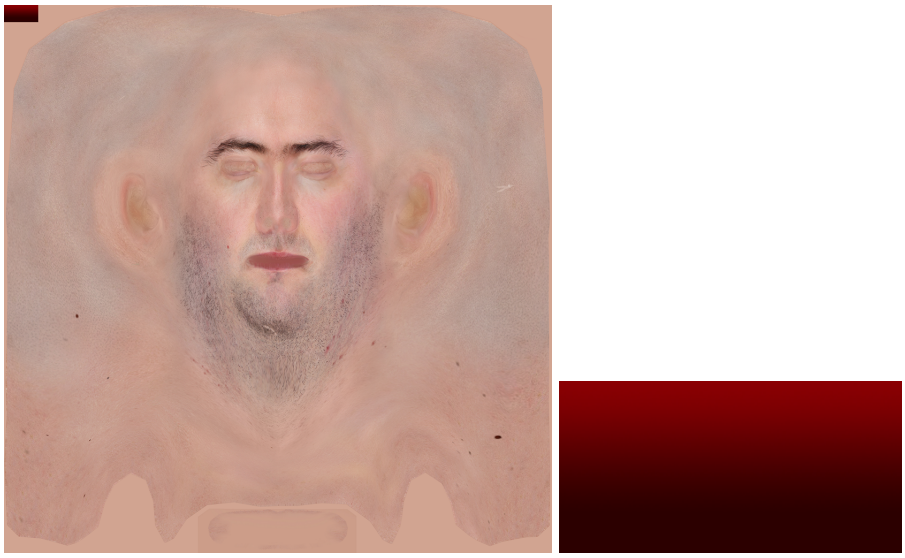


Figure 5.2: Left: we pack the cutting gutter patch into the color texture map. Right: the subsurface tissue is visualized with a simple color gradient.

Pre-packaging the texture patch inside the color map will improve the runtime performance during the creation of a wound, but the drawback is that the same texture is used for the interior of every cut. Depending on the material of the mesh (human skin or otherwise) and desired graphical fidelity, a pre-generated interior texture may or may not be sufficient.

Alternative methods include creating a sheet of interior texture patches stored separately from the color map and randomly choosing one of them during wound creation, or procedurally generating a single interior wound patch during runtime and storing it in memory. These solutions however require either rendering the interior geometry separately from the surface geometry or storing an additional per-vertex value that indicates which texture map will be used to render the vertex.

An issue that arises when procedurally generating an inner wound texture is how to determine the color of the tissue of the material. Two dimensional surface maps generally do not provide the information about the subsurface material that is required to be able to generate such a texture. For example, the color of subsurface human tissue (dermis and subcutis) cannot be extracted from a surface color map. This kind of information must thus either be inherent to the procedural function, or stored separately from the color map and read out by the procedural algorithm.

In our test implementation we use a single pre-generated color gradient from lighter to darker red to emulate the color of bloody skin tissue (Figure 5.2, right). This method is sufficient for an application where the inner geometry is not in full view. More specialized applications such as surgery simulations or educational simulations that require a more detailed representation of the subsurface tissue could benefit from using more realistic texture patches.

5.3 Visualizing the surface wound

The primary visualization of the surface wound around the border of the cut is handled by a procedurally generated texture called the *wound patch*. This texture aims to resemble the photographic references of cutting wounds. We present a relatively straightforward technique to generate a two-layered wound patch here. However, as this generation phase is an independent stage in the visualization process, it can be altered to achieve different visual results. The wound patch is consequently mapped onto the color map associated with the mesh.

5.3.1 Wound patch generation

In our two-layered wound patch generation we separate the darker and bloodied inner layer from the lighter erythematic outer layer. Figure 5.3 illustrates the composition of the two-layered wound patch. As we can see from the first two images, the inner and outer layers exhibit a fair amount of jaggies, which is exacerbated in their composition shown in the third image. The fourth and fifth images introduce a fade-out between the inner and outer layers and around the outer layer to improve blending.

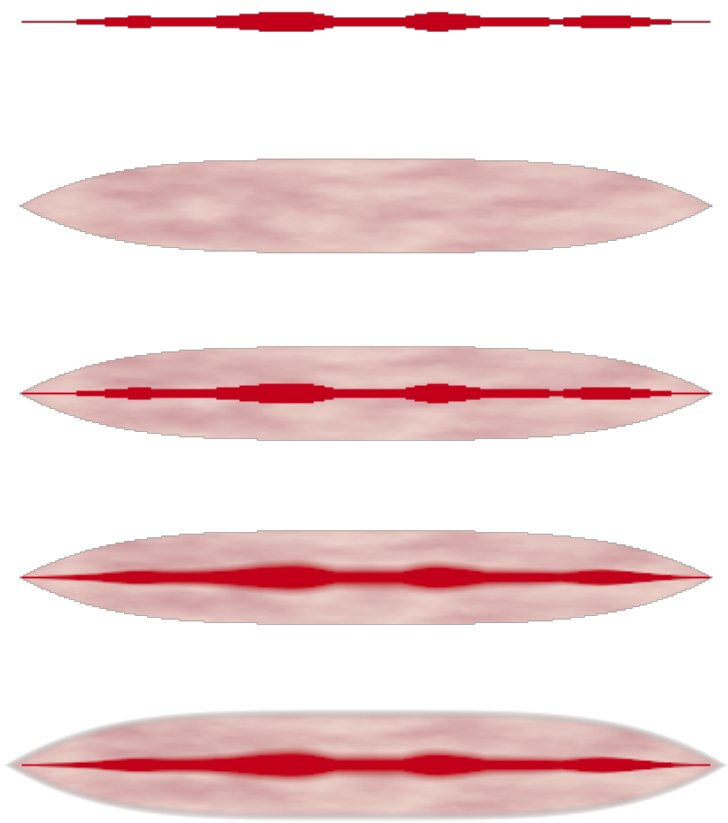


Figure 5.3: Various phases of the two-layered wound patch generation process. From top to bottom: inner layer, outer layer, two-layered combination, two-layered combination with inner fade, two-layered combination with inner and outer fade.

Both layers are procedurally generated using Perlin noise [Per02, Per04] in an effort to reduce the artificiality of the wound patch as well as the similarity between any two wound patches. We render the wound patch to a texture using a pixel shader. The width and height of the wound patch are dependent

on the length of the cut, which is determined in texture space by using the first and last points of the cutting line p_0 and p_1. Pixel width p_w of the wound patch texture is equal to the product of this distance and the width c_w of the target model color map in pixels:

$$p_w = c_w \sqrt{(p_{0_x} - p_{1_x})^2 + (p_{0_y} - p_{1_y})^2},$$

where p_0 and p_1 are two-dimensional vectors in texture-space. The pixel height p_h of the texture is defined as a function of its width:

$$p_h = 2 \log_{10}(p_w)\sqrt{p_w}.$$

This is an empirically obtained function that will – based on an average wound patch texture width of 100 to 500 pixels – limit height p_h to about one third to about one fourth of the width. Figure 5.4 illustrates the effect of the function: its decreasing slope yields a diminishing ratio of height over width p_h/p_w, and thus wound patches with larger widths will have relatively smaller heights.

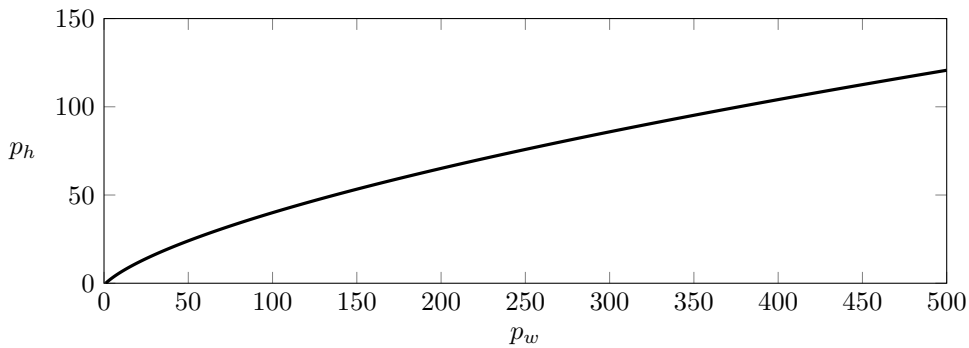


Figure 5.4: Wound patch pixel height p_h as a function of its pixel width p_w. This exponentiality causes the diameter of the wound to be more pronounced for smaller cuts and less pronounced for longer cuts.

With the dimensions of the wound patch texture known, we now use a shader to determine the color of every pixel of that texture. A number of shader parameters influence the appearance of the wound patch. These include the color of the inner layer, a darker and a lighter color that are mixed to create the noisy outer layer, and parameters that configure the noise functions. The pixel shader is shown in Snippet 5.1.

First the ranges and colors of the inner and outer layers are determined. The range is the maximum distance from the vertical center of the texture that a layer will cover. For the inner layer this will follow a parabolic shape modulated with a noise function so that it will have a variable thickness while always being wider in the center.

For noise function `pnoise` we use textureless Perlin noise by Stefan Gustavson and Ian McEwan [Gus05, GM13]. This function typically takes the texture coordinate as input, but here we use an offset `Offset.x` – which is a random value between 0 and 100 given as shader parameter – to change the location at where the noise is sampled. We also multiply the x-value by a factor of 8 to reduce the frequency of the noise, and keep the y-value at the texture center 0.5. The resulting value of the noise function is furthermore divided by a factor of 6 in order to reduce its amplitude. These two factors were empirically obtained and modifying them drastically changes the shape of the inner layer.

The `max` operation ensures that the inner layer covers at least 0.05 units of texture-space in both vertical directions away from the center. Finally, we modulated the noise function by parabolic function $y = -0.5x^2 + 0.45$ to widen the inner layer at the horizontal center and to thin it at the edges of the cut.

```
float4 woundpatch(float4 position : SV_POSITION, float2 texcoord : TEXCOORD) : SV_TARGET
{
    // ranges of color layers
    float2 offset_inner = float2((texcoord.x + Offset.x) * 4.0, 0.5);
    float range_inner = max(pnoise(offset_inner) / 4.0, 0.05) *
                        parabola(texcoord.x, -0.5, 0.45);
    float range_inner_fade = (range_inner + 0.05) * parabola(texcoord.x, -1.0, 1.0);
    float range_outer = parabola(texcoord.x, -0.3, 0.25, 4);
    float range_outer_fade = (range_outer + 0.05);

    // colors of layers
    float4 color_inner = InnerColor;
    float4 color_outer = lerp(DarkColor, LightColor, (fbm(texcoord + Offset));

    // horizontal and vertical distance from center
    float distx = abs(texcoord.x - 0.5); // horizontal distance to center
    float disty = abs(texcoord.y - 0.5); // vertical distance to center

    // texcoords.x = 0.025 to texcoords.x = 0.975
    if (distx < 0.475)
    {
        // inner layer
        if (disty < range_inner) return color_inner;

        // inner layer fade-off
        if (disty < range_inner_fade)
        {
            // color of noise at outer layer
            float offsety = range_inner_fade;
            if (texcoord.y > 0.5) offsety = -offsety;
            float2 offset = (float2(texcoord.x, 0.5 - offsety)) + Offset;
            float4 samplecolor = lerp(DarkColor, LightColor, fbm(offset));

            // lerp from inner color to outer color
            return lerp(color_inner, samplecolor,
                (disty - range_inner) / (range_inner_fade - range_inner));
        }
    }

    // outer layer
    if (disty < range_outer) return color_outer;

    // outer layer fade-off
    if (disty < range_outer_fade)
    {
        float offsety = range_outer;
        if (texcoord.y > 0.5) offsety = -offsety;
        float2 offset = (float2(texcoord.x, 0.5 - offsety)) + Offset;
        float4 samplecolor = lerp(DarkColor, LightColor, fbm(offset));

        // lerp from outer color to transparency
        return lerp(samplecolor, float4(0,0,0,0),
            (disty - range_outer) / (range_outer_fade - range_outer));
    }

    return float4(0,0,0,0);
}
```

Snippet 5.1: Pixel shader that generates a wound patch.

A fade-off range determines the maximum extent of blurring from the inner layer to the outer layer. This will reduce the aliasing issues visible in Figure 5.3. Due to parabolic function $y = -x^2 + 1$ this is 0.05 units away from the inner layer range at the horizontal center and 0.00 units away at the horizontal edges. This will create a soft appearance in the center while keeping the edges sharp.

For the outer range the calculations are fairly straightforward; parabolic function $y = -0.3x^4 + 0.25$ determines the shape of the outer layer, and its fade-off range is 0.05 units away from that in the vertical direction. Figure 5.5 graphs the aforementioned parabolic functions.

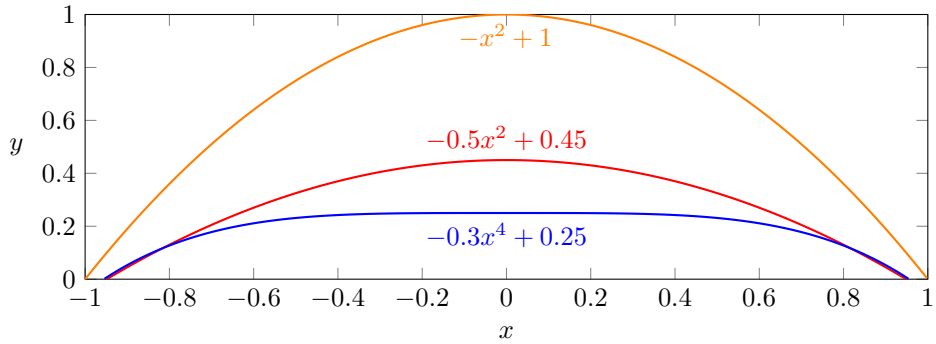


Figure 5.5: Parabolic functions used in determining layer thickness. Red: limits the thickness of the inner layer. Orange: limits the thickness of the inner layer fade-off. Blue: determines the shape of the outer layer. All of these functions were experimentally obtained. Note how the red and blue parabolas stop short of -1 and 1. This is to leave space in the texture for the outer layer fade-off.

Next we define the colors of the inner and outer layers. For the inner layer we use a uniform color to represent bloodied tissue. Conversely, for the outer layer we interpolate between a darker and a lighter skin tissue color. The interpolation factor is determined via fractal Brownian motion (fBm) [MVN68], a technique that adds detail to noise by overlapping noise functions with varying shapes – essentially making the noise fractal. We use this technique to reduce the artificial appearance of the wound texture. The general algorithm, shown in Snippet 5.2, returns a summation of successive *octaves* of noise, where each octave has a higher frequency and lower amplitude.

Shader parameters `Octaves`, `Frequency`, `Amplitude`, `Lacunarity` and `Persistance` directly control the shape of the noise. The number of octaves determines the number of layers the noise will be composed of. Frequency is inversely proportional to the feature size of the noise: higher frequencies correspond to smaller details in the noise. Amplitude controls the intensity, or height, of the features. The other parameters influence the frequency and amplitude per octave: lacunarity increases the frequency (typically by a factor of two), and persistence decreases the amplitude (typically by one half).

```
float fbm(float2 p)
{
    float sum = 0;
    float frequency = Frequency;
    float amplitude = Amplitude;

    for (int i = 0; i < Octaves; i++)
    {
        sum += snoise(p * frequency) * amplitude;
        frequency *= Lacunarity;
        amplitude *= Persistence;
    }
    return sum;
}
```

Snippet 5.2: Fractional Brownian motion.

Note that instead of using Perlin noise we use simplex noise [Per01, Gus05] in the `fbm` function. Although two-dimensional Perlin noise and simplex noise are very similar, simplex noise has somewhat more pronounced features (as seen on the last page of [Gus05]), which we deemed to be more appropriate for the color of the outer layer. Also note that on line 13 of Snippet 5.1 we shift the x and y positions of the input texture coordinate by a random offset (between 0 and 100). This is done so that no two outer layers have the same appearance.

We will now discuss how the inner layer is drawn. Firstly, the inner layer is only applicable to texture coordinates between 0.025 and 0.975. The reason is that some space must be reserved for the outer layer

fade-off at the horizontal edges of the texture (see last image of Figure 5.3). Without this the texture would abruptly end, resulting in very noticeable artifacts when the wound patch is applied to the target color map. The texture coordinate that the pixel shader is operating on must also be inside the range of the inner layer. If both conditions are met the target pixel simply takes on the inner layer color.

If the texture coordinate is within the inner fade-off distance, but not within the inner layer distance, the color of the target pixel is somewhere between the inner and outer color. Since it is unknown what the actual color of the outer layer is at the inner layer fade-off distance (this differs between each wound patch), it must be explicitly computed for that specific location. Although this causes a bit of overhead, the alternative – a two-pass wound patch generator where the outer layer is rendered first – causes overhead as well due to expensive texture lookups. The fade-off color can then simply be computed as a linear interpolation between the inner color and the sampled outer color, where the interpolation factor is the relative distance from the inner layer range to the inner layer fade-off range.

For rendering the outer layer we repeat the previous concepts and mechanisms. The previously computed outer color is returned if the texture coordinate is inside of the outer layer range. For the fade-off we sample the outer layer color at its border and fade into transparency. Drawing a transparent border around the wound patch allows us to easily blend it into the target color map.

5.3.2 Wound patch mapping

Before the wound patch texture can be mapped onto the target color map we have to know which mesh faces it covers. Although the faces directly incident to the cutting line are already known, the entirety of the wound patch usually covers a larger area in all cases but for very small cuts. The objective is thus to find all mesh faces in a certain radius from the cutting line. In order to facilitate texture painting (Section 5.3.2.2) these faces are also grouped based on the cutting line segment they are closest to.

Using cutting line length c_l in texture space, we can compute its corresponding cutting line height c_h in texture space by using the ratio of pixel height p_h to pixel width p_w of the wound patch

$$c_h = c_l \frac{p_h}{p_w},$$

where c_l itself is computed as

$$c_l = \sum_{i=1}^{\|\mathcal{C}\|} \text{dist}(\ell_i.x_0, \ell_i.x_1),$$

where $\|\mathcal{C}\|$ is the cardinality (size) of the set of cutting line segments $\mathcal{C}$, $\text{dist}(v, w)$ denotes the distance between vectors v and w, and ℓ_i represents a line segment of $\mathcal{C}$.

We can now easily determine that faces that lie inside a radius of $c_h/2$ away from the cutting line will cover the wound patch. The following section describes the algorithm that accumulates all of these faces and groups them based on their closest line segment. Another shader will then paint the generated wound patch texture onto the appropriate faces.

5.3.2.1 Accumulating cutting line faces

Recall that the cutting line formation process described in Section 4.1.2 returns a chain of cutting line segments $\mathcal{C}$, where each segment or link $\ell \in \mathcal{C}$ is defined as: $\ell = \langle f, p_0, p_1, x_0, x_1 \rangle$. This structure stores face f the segment lies in, and the world-space and texture-space locations of the endpoints of the segment. Section 4.2 described how this set was used to fuse the cutting line segment into a mesh.

This same set is now used to collect all the polygonal faces that lie within a certain distance from the cutting line on the surface mesh. Although this stage could technically run simultaneously with the fusion process, the latter alters the state of the mesh, and thus concurrent execution of these two phases would require a full mesh copy, a process which is both too time-consuming and memory intensive to outweigh the benefits of using concurrency. Additionally, as we will see momentarily, it is more convenient and efficient to have access to the original mesh faces before any local mesh subdivision occurs.

Figure 5.6 illustrates the desired output of the algorithm. Starting with set $\mathcal{C}$, an ordered set $\mathcal{C}_{\mathcal{F}}$ is constructed that maps links (or segments) $\ell_i \in \mathcal{C}$ to an unordered set $\mathcal{F}_{\ell_i}$ of their nearest faces, all of which lie inside a given radius ρ away from the cutting line. Set $\mathcal{C}_{\mathcal{F}}$ can then be used to correctly paint the wound patch onto the mesh (Section 5.3.2.2). Algorithm 5.1 first accumulates all of the faces that lie inside this radius into set $\mathcal{F}_\rho$, and afterward Algorithm 5.2 determines the mapping for set $\mathcal{C}_{\mathcal{F}}$.

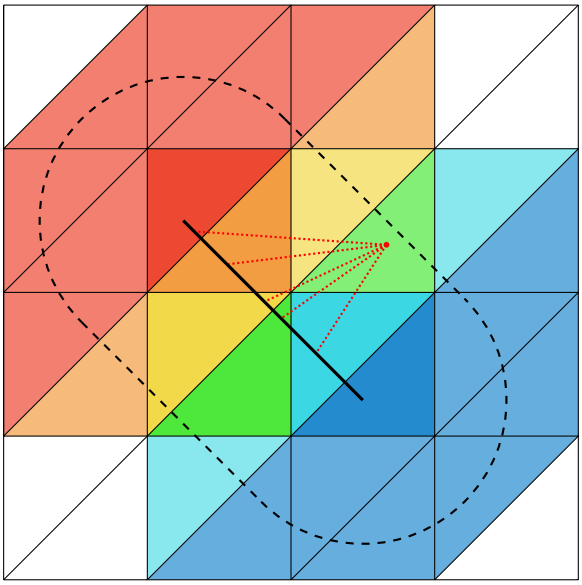


Figure 5.6: Faces located inside a given radius (dashed black line) around the cutting line (black), color coded based on the segment they are closest to. Faces incident to cutting line segments are shaded with higher saturation than those further away. White triangles denote faces located outside of the radius. We determine the nearest segment for a given face by computing the distance between the center of that face and the center of each of the segments of the cutting line (dotted red lines).

Algorithm 5.1 Accumulating faces that surround the cutting line within radius ρ.

Input: Ordered set $\mathcal{C}$ of cutting line segments, and radius ρ.
Output: Unordered set $\mathcal{F}_\rho$ of faces that lie within radius ρ from the cutting line.

```
1:  F_ρ ← {}
2:  for each link ℓ_i ∈ C do
3:      NeighborWalk(ℓ_i, ℓ_i.f)
4:      procedure NeighborWalk(ℓ, f)
5:          F_nbr ← Neighbors(f)
6:          for each face f_nbr ∈ F_nbr do
7:              if f_nbr ∉ F_ρ and f_nbr ∉ C then                    ▷ ∀ℓ ∈ C, f_nbr ∉ ℓ
8:                  Get texture coordinates ⟨x_0, x_1, x_2⟩ at vertices of face f_nbr.
9:                  for each texcoord x_i ∈ ⟨x_0, x_1, x_2⟩ do
10:                     Compute distance d between x_i and center of ℓ.x_0 and ℓ.x_1.
11:                     if d ≤ ρ then
12:                         F_ρ ← F_ρ + f_nbr
13:                         NeighborWalk(ℓ, f_nbr)
14:                         break
15: return F_ρ
```

Algorithm NeighborWalk walks over the neighboring faces of a given face recursively, starting at face $\ell_i.f$ that is incident to given cutting line segment ℓ_i. For each of the neighboring faces we check if that face is incident to the cutting line or already added to $\mathcal{F}_\rho$. If it is not, we compute the distance between the vertices of neighboring face f_{nbr} and the center of cutting segment ℓ_i.

This calculation is made in texture space due to world-space positions not being able to provide good approximations of distance in case of concave geometry. Although texture-space distances are subject to uv stretching, if texture mapping is relatively uniform over all polygons (the area of the uv-map that a polygon covers is about the same over the entire mesh), then distances in texture space provide a decent approximation. However, issues occur in cases of considerate amounts of uv-stretching, where relative distances are no longer accurate. In those cases it may be necessary to adopt a solution that generates and samples stretch maps to compensate for stretching (see for example [dL07]).

Algorithm 5.1 concludes with checking whether the texture-space distance is equal or smaller than the radius, and adds face f_{nbr} to set $\mathcal{F}_\rho$. Procedure NeighborWalk recurses here to continue with the neighbors of f_{nbr} in order to find all the faces that lie within distance ρ from $\mathcal{C}$. Note that only one vertex of a face has to lie within this threshold to be added to $\mathcal{F}_\rho$; as soon as one is detected any other vertex of that face can be skipped.

Algorithm 5.2 Create mapping between links and their nearest sets of faces.

Input: Cutting line chain $\mathcal{C}$ and set of faces $\mathcal{F}_\rho$ located within radius ρ of $\mathcal{C}$.
Output: Ordered set $\mathcal{C}_\mathcal{F}$ that maps links ℓ_i to their corresponding set of nearest faces $\mathcal{F}_{\ell_i}$.

```
1:  C_F ← {}
2:  for each link ℓ_i ∈ C do
3:      F_{ℓ_i} ← ℓ_i.f
4:      C_F ← ⟨ℓ_i, F_{ℓ_i}⟩
5:  for each face f ∈ F_ρ do
6:      d_min ← ∞
7:      ℓ_min ← nil
8:      for each link ℓ_i ∈ C do
9:          Compute distance d between center of face f and center of segment ℓ_i.
10:         if d ≤ d_min then
11:             d_min ← d
12:             ℓ_min ← ℓ_i
13:     Add face f to set F_{ℓ_min} ∈ C_F.
14: return C_F
```

With the faces surrounding the cutting line accumulated in set $\mathcal{F}_\rho$, we now use Algorithm 5.2 to assign each face to its nearest cutting line segment. We start out by taking advantage of the fact that for all links $\ell \in \mathcal{C}$ faces $\ell_i.f$ intrinsically lie on the cutting line by the definition of $\mathcal{C}$. A new unordered set $\mathcal{F}_{\ell_i}$ is assigned to each link ℓ_i and stored in $\mathcal{C}_\mathcal{F}$. We now want to assign the remaining faces previously accumulated into set $\mathcal{F}_\rho$ to their appropriate nearest segments. For each face $f \in \mathcal{F}_\rho$ the texture-space distance d between the face and each of the segments is computed, and is added to set $\mathcal{F}_{\ell_{min}}$, where ℓ_{min} represents the segment nearest to face f. Note that for computing distances the center of a segment is used, as this makes it more likely that sets $\mathcal{F}_{\ell_i}$ are orientated perpendicularly along the cutting line, which is exploited during the texture painting process.

The entire process described in this section is actually performed twice: once for a smaller radius and once for a larger radius. Because doing this naively causes overhead, a combined approach that yields two sets of $\mathcal{C}_\mathcal{F}$ for their respective radii is preferable.

5.3.2.2 Wound patch painting

With the use of a shader, we can now directly draw the wound patch onto the faces surrounding the cutting line stored in set $\mathcal{C}_\mathcal{F}$. This approach is called *texture painting*, where texels of the source color map are overwritten with colors sampled from the wound patch.

Since each of the faces surrounding the cutting line is mapped to their nearest segment, painting the wound patch onto the faces can be done in vertical slices. This concept is illustrated in Figure 5.7. Each of the groups of faces mapped to a segment ℓ_i receives a vertical slice of the wound patch texture.

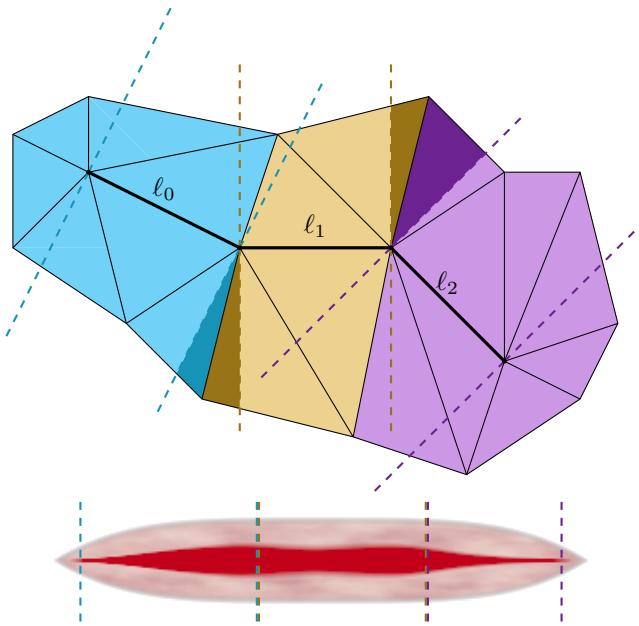


Figure 5.7: Painting the wound patch onto faces surrounding the cutting line. Top: a cutting line consisting of three segments with their nearest faces. Bottom: the corresponding vertical slices of the wound patch texture for this situation.

The dashed lines perpendicular to each cutting line segment indicate the boundaries and the orientation in which the wound patch is painted onto each face group. These boundaries also align with the vertical slices that the wound patch is divided into, where their relative length is based on the relative length of a segment as a part of the cutting line. In some cases these boundaries have to be crossed in order to draw to areas of faces that lie outside of the region perpendicular to a cutting line segment. Figure 5.7 shows these areas with a darker shade.

Finally, note how the first and last boundaries line up with the edges of the inner layer of the wound patch. This ensures that the inner layer is drawn between the first and last points of the cutting line.

Algorithm 5.3 Painting wound patch onto faces that surround the cutting line.

Input: Ordered set $\mathcal{C}_{\mathcal{F}}$ that maps links ℓ_i to their corresponding set of nearest faces $\mathcal{F}_{\ell_i}$.
Output: Updated color map with wound patch painted onto it.

1: $x_{offset} \leftarrow 0.025\, c_l$
2: **for each** $\ell_i \in \mathcal{C}_{\mathcal{F}}$ **do**
3: **for each** face $f_i \in \mathcal{F}_{\ell_i}$ **do**
4: Set shader variable `P0` as $\ell_i.x_0$.
5: Set shader variable `P1` as $\ell_i.x_1$.
6: Set shader variable `CutLength` as c_l.
7: Set shader variable `CutHeight` as c_h.
8: Set shader variable `Offset` as x_{offset}.
9: $\triangle_i \leftarrow$ SetupVertexBuffer(f_i)
10: Execute wound patch painting pixel shader (Snippet 5.3) for $\triangle_i$.
11: $x_{offset} \leftarrow x_{offset} + \text{dist}(\ell_i.x_0, \ell_i.x_1)$

Algorithm 5.3 describes our wound patch painting method. Variable x_{offset} represents the x-coordinate from where texture sampling starts for a group of faces. The first intersection point of the first cutting line segment should align with the left side of the inner-layer of the wound patch. Recall that in Snippet 5.1 a distance of 0.025 units was reserved for the outer layer, and thus the sampling should start at this distance along the length of the cutting line: $0.025\, c_l$. For each group of faces, the offset shifts by the length of corresponding nearest cutting line segment ℓ_i.

A number of variables must be sent to the shader, including the two intersection points of the cutting line in texture-space, the x-offset, and the texture-space length and width of the cutting line. For the last segment, shader variable `CutLength` is increased by $0.05\,c_l$ to ensure that the last slice of the wound patch is correctly aligned with the last point of the cutting line.

Before we can invoke the pixel shader we must configured the vertex buffer to render to a triangle of the color map that corresponds to face f_i. Function SetupVertexBuffer(f_i) converts the texture coordinates of f_i from texture space (range $x = [0, 1], y = [1, 0]$) to clip space (range $x = [-1, 1], y = [-1, 1]$), thus making it possible to directly draw to the area that face f_i covers in the color map.

```
float4 paint(float4 position : SV_POSITION, float2 texcoord : TEXCOORD) : SV_TARGET
{
    // parallel and orthogonal vectors to cutline segment
    float2 vx = normalize(P1 - P0);
    float2 vy = float2(-vx.y, vx.x);

    // direction vector from origin to texcoord
    float2 vt = (texcoord - P0);

    // x and y components of the direction vector
    float dx = dot(vt, vx);
    float dy = dot(vt, vy);

    float2 sample;
    sample.x = (Offset + dx) / CutLength;
    sample.y = 0.5 - (dy / CutHeight);

    return Texture.Sample(LinearSampler, sample);
}
```

Snippet 5.3: Wound patch painting pixel shader.

Snippet 5.3 shows the pixel shader that does the actual texture painting. This shader samples the wound patch texture for a given location on the mesh and returns the color value of the corresponding texel. When executed for an entire triangle, each pixel of that triangle will take on the color value of a texel of the wound patch, blended with its original color.

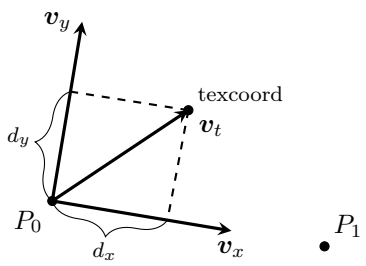


Figure 5.8: Using the dot product to compute horizontal distance d_x and vertical distance d_y for a texture coordinate relative to a cutting line segment $\overline{P_0P_1}$.

Three vectors are constructed: one between the given start and end points (`P0` and `P1`) of the cutting line segment, one vector that is perpendicular to this, and one vector from `P0` to the texture coordinate `texcoord` of the current pixel the shader is operating on. We project the texture coordinate onto the vectors parallel and perpendicular to the cutting line segment to obtain the distance along each vector that the current pixel lies. See Figure 5.8.

Determining the coordinates at which to sample the wound patch texture is now relatively straightforward. The texture coordinate for the x-axis is equal to the distance traveled along the cutting line relative to the total length of cutting line c_l:

$$\texttt{sample.x} = \frac{x_{offset} + d_x}{c_l}.$$

Because we assume that the wound patch is vertically aligned with the cutting line segment (at $y = 0.5$), we can subtract the relative distance that the current texture coordinate is away from this center to obtain the y-coordinate:

$$\texttt{sample.y} = 0.5 - \frac{d_y}{c_h},$$

where c_h is the height of the cutting line. In these equations, denominators c_l and c_h convert the given distances from color map texture-space into wound patch texture-space.

Every time that the pixel shader is executed on a face surrounding the cutting line, the appropriate texels of the color map of the model are updated. Figure 5.9 shows the final result of painting the wound patch onto the faces in the situation of Figure 5.7.

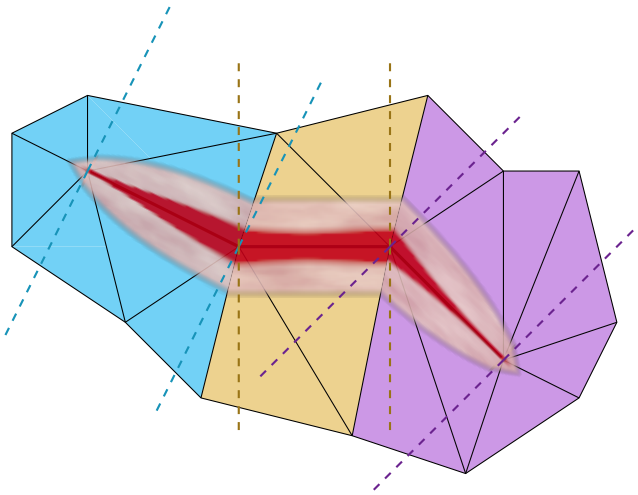


Figure 5.9: Painting the wound patch onto the faces of Figure 5.7.

5.4 Local skin discoloration

The final stage of the wound visualization is discoloration of the skin around a cut. This simulates the reddening of the skin (erythema) that occurs due to dilatation of the blood capillaries around the injured tissue. We modify the subsurface scattering technique of Jimenez et al. [JJG12a, JJG+12b] (see Appendix B) to account for this type of skin discoloring.

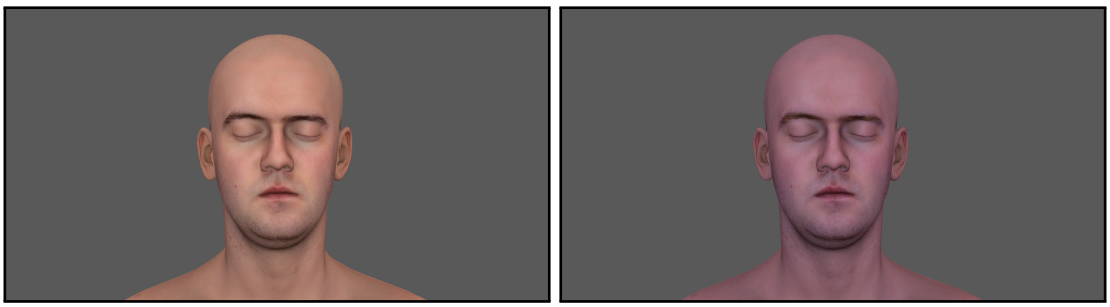


Figure 5.10: Left: default normalized kernel of the three-layer skin model. Right: a non-normalized kernel yields an unnatural dark reddish skin tone.

The subsurface scattering shader by Jimenez et al. [JJG12a] effectively applies a three-channel weighted blur filter that blends the color of surrounding pixels with a center pixel. The weights of the blur represent a diffusion profile of skin, modified by strength and falloff factors (see Section B.2.1.2). However, as Jimenez et al. note in their later work [JZJ+15], these two parameters are not very intuitive for changing

the appearance of the overall subsurface scattering effect. Additionally, because the diffusion profile and kernel weights are normalized (to white), we cannot alter the blur kernel significantly enough to change the color of the skin. See the kernel weight formula in Section B.2.1.2.

Note that the diffusion profile is normalized in accordance with the three-layer model for skin as described in d'Eon et al. [dL07] (also see Table B.1). This is to ensure that incoming light remains white on average after scattering. Additionally, the kernel weights are normalized in order to let a diffuse color map provide the final skin tone. Discarding normalization of the kernel weights will result in net energy loss or gain due to the subsurface scattering process, which is precisely what we want to achieve with skin discoloration. Unfortunately, it is not immediately obvious nor intuitive how this insight can be leveraged. Our tests have shown that the red and green color channels are too dark when using the aforementioned diffusion profile without normalizing the kernel weights. See Figure 5.10. We would thus have to resort to fine-tuning the diffusion profile to achieve the desired skin tone, effectively violating the physical correctness of the diffusion profile.

Instead, we make the observation that skin discoloration can be emulated by modifying the center weight *after* the diffusion kernel has been computed. This will allow us to precompute the kernel weights instead of having to perform expensive kernel computations in the subsurface scattering pixel shader. We make use of the fact that an RGB color value can be expressed as a multiplication of another RGB color value, where each channel is constricted to range $[0, 1]$. Empirical examination has shown a certain correlation between the color of erythematic skin and regular uninjured (Caucasian) skin. See Figure 5.11.

Sample	R	G	B
1A	0.66	0.59	0.51
1B	0.60	0.39	0.35
2A	0.63	0.56	0.46
2B	0.59	0.40	0.38
3A	0.60	0.48	0.44
3B	0.53	0.31	0.30
4A	0.86	0.64	0.55
4B	0.82	0.50	0.46
5A	0.90	0.72	0.62
5B	0.77	0.41	0.38
6A	0.80	0.49	0.28
6B	0.76	0.36	0.22

Ratio	R	G	B
1B/1A	0.91	0.66	0.70
2B/2A	0.94	0.72	0.83
3B/3A	0.89	0.64	0.69
4B/4A	0.95	0.77	0.84
5B/5A	0.86	0.57	0.62
6B/6A	0.95	0.73	0.78

Figure 5.11: Sampling the skin color around cutting wounds. For each sample we collected the average color of a 7×7 pixels square around the target location. Samples labeled A represent regular skin color, and samples labeled B represent discolored skin due to erythema. The table on the top-right lists the RGB colors for each sample in floating-point representation. The table on the bottom-right shows per-channel ratios of B-samples to corresponding A-samples.

```
1  float4 ssss(float4 position : SV_POSITION, float2 texcoord : TEXCOORD0) : SV_TARGET
2  {
3      // sample color and depth
4      float4 color = ColorMap.Sample(PointSampler, texcoord);
5      if (color.a == 0.0) discard;
6      float1 depth = DepthMap.Sample(PointSampler, texcoord).r;
7
8      // determine scale of the blur filter
9      float1 distproj = 1.0 / tan(0.5 * radians(FOVy));
10     float1 scale = distproj / depth;
11
12     // step to fetch surrounding pixels
13     float2 step = scale * KernelWidth * Direction * color.a * 0.5;
14
15     // sample discolor and convert from [0,1] to [0,2]
16     float4 discolor = DiscolorMap.Sample(PointSampler, texcoord);
17     discolor.rgb = discolor.rgb * float3(2,2,2);
18
19     // discoloration of center sample
20     Weights[3] = lerp(Weights[3], Weights[3] * discolor.rgb, discolor.a);
21
22     // add contribution of center sample
23     float4 final_color = float4(color.rgb * Weights[3], 1.0);
24
25     // add contribution of surrounding samples
26     for (int i = 0; i < 7; i++)
27     {
28         if (i == 3) continue; // skip center pixel
29
30         // sample color and depth for current sample
31         float2 offset = texcoord + Offsets[i] * step;
32         float3 sample_color = ColorMap.Sample(LinearSampler, offset).rgb;
33         float1 sample_depth = DepthMap.Sample(LinearSampler, offset).r;
34
35         // follow surface
36         sample_color = lerp(sample_color, color.rgb,
37             saturate(300.0 * distproj * KernelWidth * abs(depth - sample_depth)));
38
39         // add contribution of offset, modulated by a kernel color
40         final_color.rgb += sample_color.rgb * Weights[i];
41     }
42
43     return final_color;
44 }
```

Snippet 5.4: Separable subsurface scattering pixel shader with local discoloration.

Based on these findings we conclude that the discoloration color can be expressed as a percentage of the color of uninjured skin surrounding a cut. This is about 85% to 95% for the red channel, 60% to 75% for the green channel, and 60% to 85% for the blue channel. By modifying the center weight of the blur kernel in the subsurface scattering shader we can adjust the overall color of the skin. Since the center sample has the highest contribution to the final skin color we can keep the other samples unaltered to retain the blur effect.

The final subsurface scattering shader is shown in Snippet 5.4. Only lines 15 to 20 are different from the version presented in Section B.2.1.2. A texture map is sampled in order to achieve localized discoloration. The alpha value of this texture is interpreted as a blend factor between the original skin color and the discolored skin color, and the RGB value represents the per-channel modifier of the center sample. Note that color packing in the discoloration map allows for modifiers to reduce or increase the color of the center sample; its range is in $[0, 2]$. Although not used for our purposes, this allows for interesting ways to (locally) change the overall skin tone due to subsurface scattering.

For every cut we paint onto the discoloration texture map of the target mesh. By default, this texture has an alpha value of zero, and thus does not interfere with the original subsurface scattering process. The general painting procedure works similarly to the wound painting process: a pixel shader draws to the faces that lie in a certain radius around the cut. Since skin discoloration can occur quite far away

from the cutting wound we choose the radius to be equal to cutting height c_h, which is twice the distance as that used for wound patch painting. Recall that the corresponding faces were collected at an earlier stage (see Section 5.3.2.1).

Next we iterate over each of the collected faces and set them up for rendering using the SetupVertexBuffer function that was described in Section 5.3.2.2. The pixel shader that paints on the discoloration texture accepts four shader variables: the first point of the cutting line P_0, the last point of the cutting line P_1, the maximum painting distance ($0.5\,c_h$), and the discoloration factor. The latter is an RGB value where the channel values are randomly chosen to be between the percentages described above ($R \in [0.85, 0.95]$; $G \in [0.60, 0.75]$; $B \in [0.60, 0.85]$). We execute the shader shown in Snippet 5.5 for each face bordering the cutting line.

```
float4 discolor(float4 position : SV_POSITION, float2 texcoord : TEXCOORD) : SV_TARGET
{
    // compute distance from cutline
    float t = 0.0;
    float dist = distance(P0, P1, texcoord, t);
    t = saturate(t) * 2.0 - 1.0; // clamp to [0,1], then convert to [-1,1]

    // compute maximum distances for inner and outer layers
    float range_inner = MaxDistance * (-0.5*t*t+1); // MaxDistance = CutHeight/2
    float range_outer = 2.0 * range_inner;

    // pack color value ([0,2] to [0,1])
    float3 discolor = Discolor.rgb * float3(0.5,0.5,0.5);

    // compute alpha intensity for outer layer
    float range = 1.0 - (dist / range_outer);
    float noise = fbm(texcoord * 0.5, 4, 0.5, 64.0);
    float alpha = clamp(noise, 0.0, range);

    // increase alpha intensity for inner layer
    if (dist <= range_inner)
        alpha += (1.1 - (dist / range_inner));

    return float4(discolor, alpha);
}
```

Snippet 5.5: Pixel shader that generates localized discoloration.

First we compute the distance of the current pixel to cutting line segment $\overline{P_0P_1}$. As in the wound patch painting shader, each of these points is expressed in texture space. Variable t represents the parametric distance of the projection of `texcoord` onto $\overline{P_0P_1}$ and determines the maximum range of the discoloration. We clamp the value of t since it can exceed the desired range of $[0, 1]$ (see Figure 5.12), and then convert it to range $[-1, 1]$. This value is then used as input for the following parabolic function that determines the curve of the inner and outer ranges of the discoloration.

$$f(t) = 0.5\,c_h(-0.5t^2 + 1)$$

This ensures that the range is maximum at the center and less pronounced at the edges:

$$f(-1) = 0.25\,c_h, \qquad f(0) = 0.5\,c_h, \qquad f(1) = 0.25\,c_h.$$

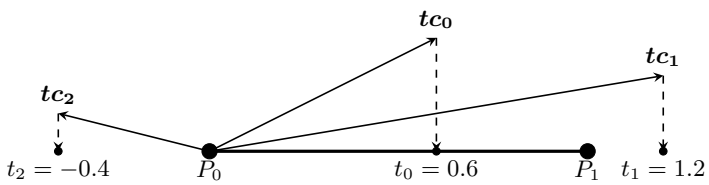


Figure 5.12: Variable t is only in range $[0, 1]$ when projected points lie on $\overline{P_0P_1}$.

As mentioned, we use the alpha value of the discoloration map as strength factor. Computing its intensity for the outer layer (or noise layer) is based on the relative distance of the pixel from the cutting line to the maximum distance allowed. Thus, the `range` variable will be 1.0 for pixels on the cutting line and 0.0 at maximum range.

We then use noise – or more specifically, fractal Brownian motion – to give the discoloration effect a splotchy appearance. A higher frequency is used to reduce the distance between splotches. However, note that finding the right appearance for noise requires a decent amount of fine-tuning and may be different from one application to the next. Clamping the noise between zero and `range` ensures that the noise fades off around the maximum distance.

Finally, for the inner layer we linearly increase the intensity of the alpha channel by a value inversely proportional to the distance from the cutting line $\overline{P_0P_1}$. Using a higher value than 1.0 as basis (in this case 1.1) increases the distance at which the fade-off starts.

Figure 5.13 illustrates the visual effect of the addition of localized skin discoloration to the subsurface scattering shader. Notice how the wound patch and discoloration complement each other to obtain the desired result.

Figure 5.13: Various cuts rendered with local skin discoloration, both with and without mesh carving.

There are two major drawbacks of working in texture space. Firstly, due to UV distortion – or stretching – locations in texture space do not always correspond exactly with model-space locations. This causes an undesirable shift of the discoloration in areas of significant stretching. See Figure 5.14. The same problem occurred when drawing the wound patch, but here the effect is exacerbated because we only use two points to paint the discoloration in between. As with the artifacts exhibited by the wound patch, a solution would be to compute and sample a stretch-correction texture to modulate per-pixel texture-space distances.

The second drawback of working in texture space is overwriting the color of a previous discoloration patch. Although blending can be configured to produce an average color of both instances, faces outside of the new region will abruptly change in color. See Figure 5.15.

Figure 5.14: Stretching of the discoloration patch. Notice that although the discoloration is centralized at the first and last points of the cut, it curves to the left in between those points.

Figure 5.15: Overwrite of the color of the discoloration patch. The cut at the top that was created last bleeds its color into the discoloration of the first cut, causing a sharp color change at its border.

Closing remarks

In this chapter we presented our approach for visualizing cutting wounds which consisted of a number of novel techniques. We rendered the cutting gutter with a simple pre-generated color gradient texture patch that is packed into the color map of the target mesh. This is sufficient for applications where the interior geometry does not have to be represented in detail. Visualizing the surface wound around the border of the cut was done by procedurally generating and painting a wound texture patch during runtime. Moreover, we altered the subsurface scattering shader by Jimenez et al. [JJG12a] to allow local color variations and used it to visualize erythema around the surface wound.

With a scarcity of relevant literature on the topic of wound visualization, we have attempted to provide one possible method of rendering wounds. Our main contribution is not the aesthetic of the visualization itself but rather describing a process to represent wounds without the use of pre-generated content. Most notably, the shader that generates the wound patch can be completely replaced to produce a more accurate or detailed wound patch, or visualizations for other types of material than Caucasian skin. Additionally, we expect that our simple subsurface scattering extensions are applicable to more situations than just the one presented here.

The primary drawback of our approach is that texture stretching distorts texture-space distances which our texture painting methods rely upon. We expect that by computing and using stretch maps this can be somewhat alleviated, but we have not pursued this endeavor and therefore unsure whether this will solve all issues. Another issue is that the performance of accumulating and painting onto faces around the cutting line depends on the level of mesh refinement. This is why this process is done before the mesh cutting simulation, but it would be preferable to execute both stages simultaneously.

6. Implementation and performance

This chapter describes the implementation of our test application, a three-dimensional mesh viewer that allows a user to define new cuts. We briefly describe the scene description, discuss our texture maps, describe the user interface, and list the libraries used to build our test application. We also include some performance tests and results.

Various images in previous chapters already showed specific parts of our test application. Figure 6.1 shows the default view of the test application with the default scene and settings. The window consists of the main scene view and a user interface that allows the rendering to be tweaked.

Figure 6.1: Default view of our test application.

A description of the scene and settings for the skin renderer are defined in JSON files that are loaded during program startup. The program executable allows custom paths to the configuration and scene files to be given as arguments, or uses the default scene and settings otherwise.

6.1 Scene description

The scene description is defined in an external JSON file that is read and parsed during program initialization. Exactly one camera, one model, and one or more lights must be defined to be able to construct a valid scene.

The camera is defined by a set of spherical coordinates, that is the zenith, azimuth, and distance in radians from the scene origin. Lights are also defined with spherical coordinates, and have an additional RGB value that specifies its color and intensity. A model is defined by its world coordinates, rotation in Euler angles, a path to a mesh file, and paths to a number of texture maps (see Section 6.2). The mesh file is specified in the Wavefront OBJ file format (ASCII variant) [Wav95], and texture maps are specified by DirectDraw Surface (DDS) file format (uncompressed and without mipmaps).

Section 3.3.1 already described the mesh loading process. Since parsing an ASCII OBJ file can take tens of seconds for reasonably sized meshes, a binary representation of the mesh is written to a file once a mesh has been successfully loaded. This file contains index and vertex arrays that define the mesh. Reading from this file simplifies mesh construction and greatly reduces application startup time while requiring less storage space than an ASCII-formatted OBJ file.

6.2 Texture maps

Various texture maps are used for rendering and wound visualization. Five textures are mapped to the target mesh, another is used for irradiance environment mapping, and the last contains precomputed values for specular shading. In Figure 6.2 a composition of the model-bound texture maps is shown. We will now give an overview of the purpose of each texture map.

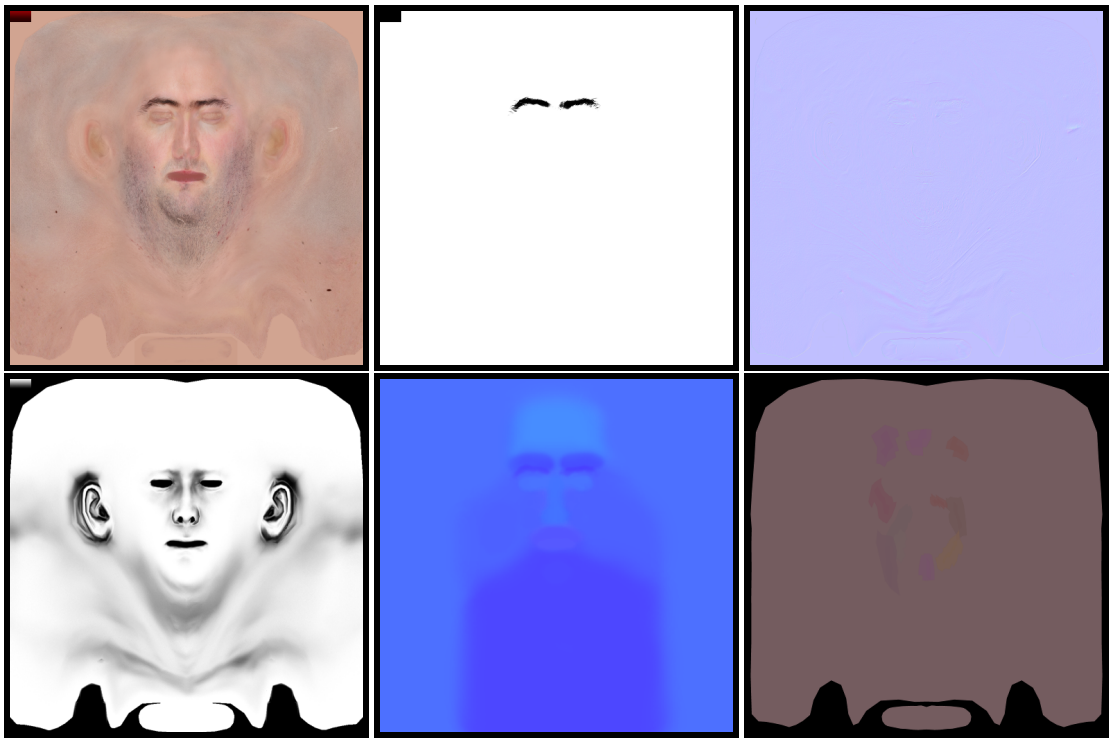


Figure 6.2: Model-specific texture maps required by the system. First row: diffuse color map (RGB channels), subsurface scattering strength (alpha channel of the color map), and normal map. Second row: ambient occlusion map, specular map, and a discolor map containing a number of splotches (RGB channels).

The diffuse color (or albedo) texture defines the primary color of the skin surface. Composed of four channels, its RGB value simply defines the color and its alpha value represents parameterized strength of the subsurface scattering, where lower values reduce the width of the scattering filter. In the top-center image of Figure 6.2 the eyebrows and the small area used for inner tissue color are completely black, thus disabling subsurface reflectance around those areas.

Adding meso-features such as bumps, pocks, and grooves to the skin surface is done via the normal texture map. It consists of two channels, one for the tangent direction and one for the bitangent direction in tangent space.

Precomputed ambient occlusion resulting from overhead lighting is stored in a single-channel texture map. Darker areas signify the severity of the occlusion. Note that the cutting gutter patch is a gradient from white at the surface to black at the bottom to simulate shadow effects inside the cutting gutter.

A two-channel specular texture map contains parameterized information about specular intensity and specular roughness, respectively. These values are used in the specular Kelemen/Szirmay-Kalos shader (see Appendix B).

The discolor map allows local changes to the color of subsurface reflectance as described in Chapter 5. The bottom-right image of Figure 6.2 shows an example of what this texture would look like after a number of cuts have been made. Note that only the RGB channels are shown since the alpha channel hides most of the texture map around each cut. These channels contain the discoloration multiplication factor, while the alpha channel contains the interpolation factor between the original subsurface reflectance color and its discolored variant.

6.3 User interface

Figure 6.1 shows the user interface with its default values. A user can rotate around a model as well as zoom in and out, and can perform a couple of actions. By holding shift and clicking the left mouse button the user can define two points on the model between which a new cut is to be created. There are three possible responses to this event:

1. Wound patch painting: a newly generated wound patch is painted onto the model between the given points. The geometric mesh is unaltered.
2. Cutting line merging: same as wound patch painting, but additionally a cutting line is formed between the two points and merged with the existing mesh.
3. Incision carving: the full method in which a wound patch is generated and painted, a cutting line is formed and merged with the mesh, the cut is opened, and gutter geometry is generated.

Another event is triggered by holding control and clicking the left mouse button. Depending on the selected method of subdivision – 3-split, 4-split, or 6-split – the mesh face under the cursor is subdivided in either three, four, or six child faces. This can be used to test specific scenarios for cutting line merging and cutting line opening.

Furthermore, a number of options control the skin rendering. With the panel on the left side of the screen we can toggle the various texture maps on and off, and the shading and light intensity can be fine-tuned to achieve specific lighting scenarios.

6.4 Libraries used

Window management, input handling, and the program loop in our test application are implemented using the Windows API (specifically the Win32 API). Direct3D 11.0 [Mic16] is used for graphics rendering. As such, the program will only run on Windows systems, and is only tested on Windows 7 and later.

Our application is programmed in C++11 and compiled with Visual Studio 2012. Windows SDK 8.0 or 8.1 is required to build the source code. The following additional libraries were used:

- DirectXTK: helper classes for Direct3D 11 [Wal16a].
- DirectXTex: texture loading and processing [Wal16b].
- SimpleJSON: JSON file parsing [Anc15].
- ImGui: graphical user interface [Cor15].

6.5 Performance

In this section we describe a number of tests that measure the performance of our approach. Specifically, we want to know the performance impact on both the CPU and the GPU due to adding one or multiple cuts. The following tests have been conducted:

- CPU running time per stage for predefined cuts of varying length.
- GPU rendering time of a single frame of the overall skin renderer and visualization.
- GPU rendering time of the modified localized subsurface scattering shader compared to the original shader.

Table 6.1 lists the specifications of our test system. We acquire CPU measurements using the `Performance QueryCounter` function in the Windows API, and profile rendering performance using the Frame Debugger and Frame Profiler tools in AMD GPU PerfStudio 3.

OS	Windows 8.1 Pro x64
CPU	Intel Core i7 920 clocked at 2.67GHz
RAM	6GB DDR3 (Dual Channel) clocked at 534MHz
GPU	ATI AMD Radeon HD 7850 with 2GB GDDR5

Table 6.1: Specifications of our test system.

6.5.1 Test 1: Per-stage running time

With this first test we want to find out what the total delay between the definition and the appearance of a new cut is, or in other words the total CPU running time of our approach. To identify possible bottlenecks we measure the running time of each stage of the cutting simulation and wound visualization. Additionally, we want to find out the impact of both the length of the cut and the level of refinement of the local mesh area around a cut.

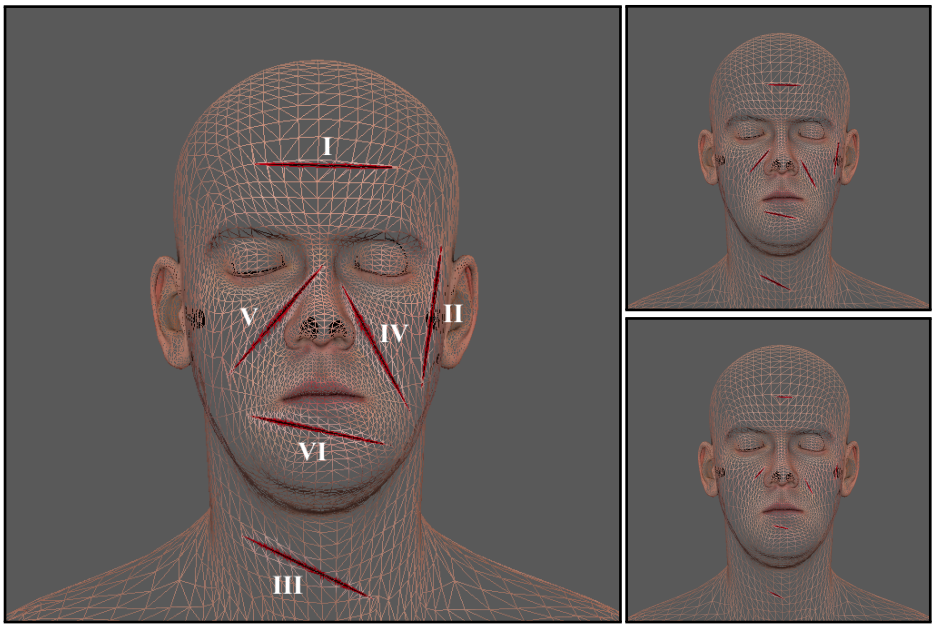


Figure 6.3: Six predefined cuts of various lengths used in our performance test. Cut samples I–III are in areas of lower refinement and samples IV–VI are in areas of higher refinement. Left: large cut lengths. Right: medium and small cut lengths.

Figure 6.3 shows the configuration of our test on a wireframe of our model. Six cut samples of relatively large length were defined manually in screen space. For two successive tests we again create six cut samples at the same locations, but reduced in length by 50% and 25% respectively. Cut samples labeled as I–III are created in mesh areas with lower level of refinement, and samples IV–VI in areas with higher level of refinement. Samples are quantified based on their length in texture space and the level of refinement around the cut, that is, the number of faces adjacent to the cutting line (before remeshing). Higher values for both quantities are expected to produce more complex cuts.

For each sample, we measure the running time of each individual stage in milliseconds as an average over 100 runs. Table 6.2 displays the results. Note that the sample numbers correspond with those depicted in Figure 6.3. The cutting simulation and wound visualization stages are ordered chronologically, which are abbreviated as follows: **FC** – form cutting line; **GP** – generate wound patch; **PP** – paint wound patch and discoloration; **MC** – merge cutting line into mesh; **CI** – carve incision (open cut and create gutter).

sample	**faces**	**length**	**FC**	**GP**	**PP**	**MC**	**CI**	**total**
small I	4	0.025	0.014	0.219	6.848	3.638	2.721	13.440
small III	6	0.042	0.017	0.225	12.717	2.154	1.106	16.220
small II	6	0.051	0.017	0.224	13.003	2.184	1.120	16.548
small VI	8	0.027	0.019	0.226	16.841	2.006	1.011	20.103
small V	8	0.038	0.020	0.226	20.919	1.839	0.924	23.928
small IV	9	0.048	0.018	0.227	25.360	2.049	0.966	28.621
medium I	9	0.052	0.021	0.237	12.532	2.540	1.199	16.530
medium III	11	0.084	0.025	0.238	18.767	2.180	1.044	22.254
medium II	11	0.096	0.022	0.255	19.865	2.307	1.044	23.492
medium IV	17	0.095	0.026	0.253	54.019	3.222	1.388	58.908
medium VI	18	0.061	0.027	0.221	32.418	3.471	1.419	37.556
medium V	19	0.076	0.030	0.225	46.359	3.446	1.450	51.511
large I	22	0.119	0.035	0.267	23.740	4.085	1.593	29.720
large II	24	0.191	0.036	0.264	34.373	4.081	1.690	40.444
large III	25	0.176	0.038	0.256	34.134	4.073	1.727	40.227
large IV	30	0.173	0.040	0.249	145.400	4.962	1.916	152.567
large V	39	0.155	0.051	0.248	101.956	6.079	2.347	110.681
large VI	46	0.151	0.053	0.249	95.329	7.562	2.672	105.865

Table 6.2: Performance measurements of the cutting simulation and wound visualization stages. Cut samples are ordered by the number of faces adjacent to the cut first, and by their world-space length second. Average running times over 100 runs are shown for all individual stages, as well as the average total running time per sample. All times are in milliseconds.

From these results we can derive that the average running time over all samples is about 45 ms. The highest recorded delay between defining and creating a cut is a little over a tenth of a second, which is drastically reduced for cuts that are both shorter and located in areas where mesh faces are relatively larger. We want to look at the influence of both criteria, and first focus on how the cut length influences total running time.

By plotting the length against the running time we can get a sense of how these two quantities relate to each other. See Figure 6.4. We can see an increasing discrepancy in running time for the six samples from each cut sample group (large, medium, and small) as the cut length increases. This happens to coincide with the level of refinement of the mesh area that the samples are located in, where the lower samples correspond to a lower level of refinement and the upper samples with a higher level of refinement. The level of refinement thus becomes more influential as the length of the cut increases. Also note that the total running time increases much faster as the cut length increases, as the trending curve is near-linear for small and medium cuts, and becomes exponential for larger cuts.

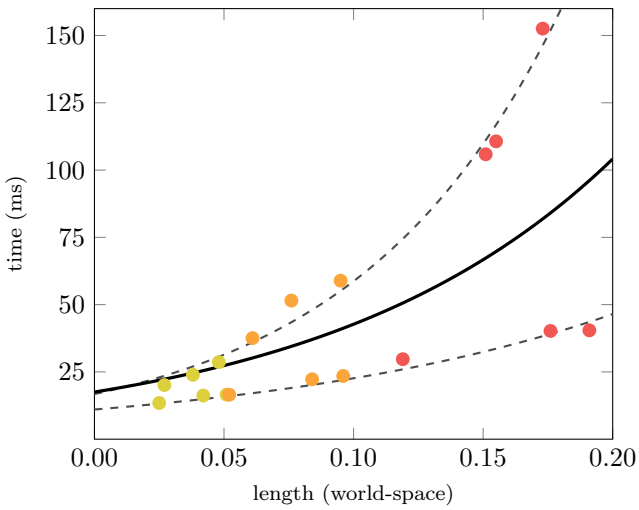


Figure 6.4: Plotting the cut length against the total running time. Yellow, orange, and red points indicate samples from small-sized, medium-sized, and large-sized cut lengths, respectively. The black curve shows the average (exponential) upward trend of the time increase, and the lower and upper dashed curves indicate the trends for samples in areas with lower and higher levels of refinement, respectively.

To find out how much the cut length influences the running time of each individual stage, we have computed the average running time per stage, per sample group. The resulting data is graphed in a stacked bar chart; see Figure 6.5. From this graph we can see that the running time of the *paint wound patch* stage (**PP**) differs the most between cut lengths. This is not surprising as most of the algorithms used in this stage depend on the number of adjacent faces, which is obviously higher for larger cuts. Further tests have shown that the bottleneck lies in the texture painting algorithms, possibly due to overwriting graphics memory for every face that is painted onto. For the remaining stages the difference in running time between cut lengths is negligible.

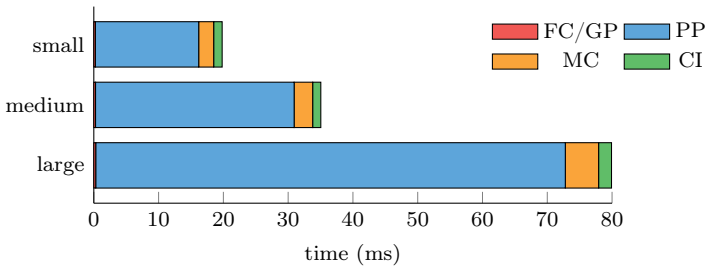


Figure 6.5: Stacked bar chart of the running time per simulation stage for small, medium, and large cut length sample groups. Note that because the *form cutting line* and *generate wound patch* stages have such a low contribution to the total running time they are grouped in this graph.

We will now investigate how the level of refinement around the cut influences the running time, while ignoring the cut length. By plotting the amount of adjacent faces against the total running time, and then interpolating and extrapolating the data using curve fitting, we can visualize this relationship. Figure 6.6 shows the result.

This graph shows that cut samples in areas with lower refinement (blue points and curve, corresponding to samples I–III described above) outperform those created in areas with higher refinement (red points and curve, samples IV–VI). Note that the relation between the amount of adjacent faces and running time is nearly linear for both sample groups.

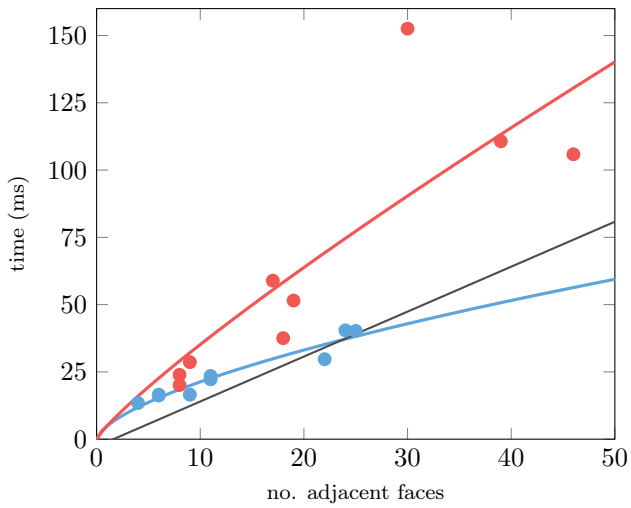


Figure 6.6: Plotting the level of refinement against the total running time. Blue and red points indicate samples in areas with lower and higher level of refinement, respectively. Matching curves show the interpolated running time for these two sample sets, and the gray line illustrates the difference in running time between these curves.

Furthermore, the difference in running time between both sample groups increases as the amounts of adjacent faces grows. For example, at around 10 faces the creation process is about $1\frac{2}{3}$ times faster in an area with lower refinement than in an area with high refinement. At 50 faces this has increased to a factor of about $2\frac{1}{4}$. This increase in time is near-linear and is indicated by the gray line in Figure 6.6.

6.5.2 Test 2: Frame rendering time

In this second test we measure the rendering time of a single frame on the GPU to find out how additional cuts influence the rendering performance. We start at no cuts and manually add up to 30 cuts while keeping them all visible in the viewport. The frame time is measured as an average over 10 frames using the GPUTime counter in GPU PerfStudio 3 at 0, 10, 20, and 30 cuts. See Table 6.3.

no. cuts	time
0	2.173
10	2.215
20	2.242
30	2.273

Table 6.3: Performance measurements of the GPU frame rendering time at various numbers of cuts. All times are in milliseconds.

As shown in the table, the total GPU processing time of a single frame is about 2.2 milliseconds. Note that this does not include the overhead of sending data from the CPU to the GPU on every frame, which accounts for an additional 0.2 to 0.4 milliseconds. Based on these measurements our test system renders the scene (with default viewport) at about 385–417 frames per second.

The total number of visible cuts seem to have only minimal impact upon the frame rendering time. Each additional 10 cuts add at most 30-40 µm (microseconds) of frame time. A likely explanation is that for each cut additional geometry has to be processed by the GPU. Note that the color and discolor maps are sampled unconditionally, whether or not a cut is located at a given texture coordinate. This is the reason why the rendering time differs so little between having no cuts or having any number of cuts.

6.5.3 Test 3: Subsurface scattering rendering time

In this final test we compare the modified subsurface scattering shader with localized discoloration to the original subsurface scattering by Jimenez et al. [JJG12a]. We do this by profiling the draw calls of the subsurface scattering (horizontal and vertical blur passes) using the Frame Debugger in GPU PerfStudio 3. We collect data at 10 cuts and at 50 cuts, where each draw call is measured as an average GPU time over 100 runs. Figure 6.7 shows the results of the test.

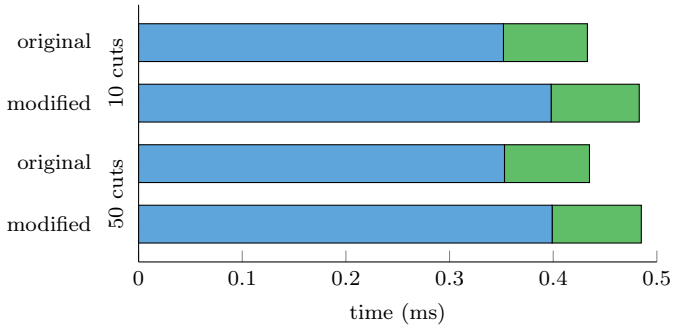


Figure 6.7: Performance measurements of the draw calls of the original and modified subsurface scattering shaders for both 10 and 50 cut samples. The horizontal pass is shown in blue and the vertical pass in green color.

As expected, the results show a slight performance hit between the original and modified subsurface scattering shaders. This increased rendering time of only 13% is due to sampling the discoloration map. Additionally, the results show that the number of cuts do not have any significant impact upon the rendering time. Overall we can say that since the combination of the horizontal and vertical draw calls is still well below 1 millisecond the performance impact of the modified shader is minimal.

7. Conclusion and discussion

In this thesis we explored the feasibility of constructing a real-time virtual damage model for cutting wounds. We looked at how existing mesh cutting simulation methods can be combined with state-of-the-art skin rendering techniques and novel wound visualization techniques.

To support a damage model that combines cutting simulation with wound visualization we proposed a representation for triangular surface meshes that maintains a loosely coupled geometrical component (positions and attributes) and a topological component (nodes, edges, faces). Although not particularly storage-efficient, separating these components allows the topology to be modified without affecting the mesh parameterization. Moreover, having a loose coupling between these components significantly improves the efficiency of extracting a renderable mesh. In order to facilitate mesh cutting algorithms and rendering requirements our data structure provides efficient face-edge and edge-face traversal, as well as quick access from faces to geometric data. This contrasts with most popular mesh representations that focus on reducing memory requirements but require expensive traversal and conversion algorithms for remeshing and rendering purposes, respectively.

Our mesh cutting simulation approach is a combination of previous works, where we use a sweeping surface as described by Bruyns et al. [BSM+02], generate a cutting gutter following Zhang et al. [ZPD02], and define the width of the cut as proposed by Lim et al. [LJD07]. However, our approach differs in that we define a cut directly from two mesh surface points instead of representing the cutting tool and using a progressive cutting scheme. We describe a complete localized remeshing scheme that is able to handle all possible configurations between the cutting line segment and the mesh topology using appropriate elementary subdivision algorithms. Special attention is devoted to correctly maintaining the topology and parameterization of our mesh representation after each remeshing operation. By having a minimally intrusive approach where only the topology directly bordering a cut is modified we keep memory and processing requirements to a minimum.

Wounds are visualized using texture patches for the cut interior and the surface surrounding the cut. For the latter we present a method that generates and paints a wound texture on the fly using pixel shaders and runtime information. Our main objective is to describe a solution that avoids pre-generated content in an effort to reduce the reliance on artistic input. Although we do not reproduce the aesthetics of cutting wounds exactly, a shader is presented that procedurally generates a blendable wound patch texture. Furthermore, we extend a state-of-the-art subsurface scattering method to allow local color variations using a texture map.

In conclusion, extending mesh cutting simulation with wound visualization is feasible if care is taken to keep both the mesh topology and mesh parameterization in a valid state after every remeshing operation. What is considered as valid depends on the application, but in our case the topology must remain triangular, closed, orientable, and manifold, and the mesh parameterization must remain continuous over the surface – with disjoint parameterization for cutting gutters. Choosing a mesh representation that allows dynamic changes and that provides quick traversal to often-used mesh elements as well as to the geometry and attributes helps in this effort.

For the visualization of wounds we believe that a generalized method can only be achieved by reducing artistic influence, which can be realized by procedurally generating a wound texture using runtime information. Care must be taken to blend the wound into the original skin texture in a visually convincing

manner. Finally, we have found that subsurface scattering is a subtle but important visual effect and that slight variations allow us to create interesting visual effects.

We think that a system that combines cutting simulation with wound visualization can lead to interesting applications for both medical simulators and video games. Having medical simulators include a more realistic visual response may better prepare a user for real-life situations and may also open up possibilities for new visually-oriented surgical and forensic simulators. For video games, being able to precisely simulate cuts during runtime may increase their immersive experience. Additionally, our procedural wound generation approach may be useful in situations where reducing dependence on artistic input is desirable.

As an additional advantage, integrating our method into existing real-time software that uses triangular surface meshes is relatively straightforward, especially in comparison to most current mesh cutting solutions that use volumetric meshes. Adopting our mesh representation is not strictly necessary, though care must be taken to separate the topology from the parameterization. Using another representation might also lead to less efficient traversal operations. We expect that our method can be easily extended to support different types of wounds such as lacerations, punctures, and abrasions. Additionally, with proper adjustments to the simulation and visualization response our model can be used to simulate cuts on different types of material such as fruit, clay, and wood.

The performance of our cut simulation is dependent on both the length of the cut and the level of refinement around the cut definition. For smaller cuts the ratio of length to time is near-linear which increases to an exponential relation for larger cuts. Due to expensive graphics memory swapping the wound patch painting stage is currently the least efficient stage and most susceptible to increase or decrease in running time. The processing time of cuts created in areas with a lower level of refinement is much less impacted by the amount of adjacent faces than that of cuts created in high refinement areas, causing the former to outperform the latter by a factor of up to $2\frac{1}{4}$ for very large cuts.

Our wound visualization and skin rendering approach is very lightweight, and defining additional cuts has very little impact on the total frame rendering time. Moreover, our modified subsurface scattering shader is only about 13% slower compared to the original separable screen-space subsurface scattering shader by Jimenez et al.

7.1 Future work

This thesis explores and presents a mere subset of possible cutting simulation and wound visualization techniques. A number of improvements and extensions can be introduced to increase the realism and graphical fidelity of our model.

Our solution currently lacks soft body simulation or mesh deformation around a newly created cut. In reality, when a sharp object initially touches the surface the tissue deforms due to the pressure before puncturing the surface and creating an incision. Additionally, after the initial instantaneous separation, a cut should open over time following the laws of physics. For performance reasons we think that mass-spring or meshfree methods are preferable over finite element methods. The methods presented by Bruyns et al. [BSM+02] and Zhang et al. [ZPD02] describe how the simulation can be extended with mass-spring systems, and Lim et al. [LJD07] describes a meshfree soft-body simulation method.

For this thesis we have omitted a progressive cutting scheme (see for example Mor et al. [MK00], Zhang et al. [ZPD02], and Lim et al. [LJD07]). In progressive cutting the mesh surface is continuously remeshed while a cutting tool slices through the material, providing a more realistic account of how real cuts emerge. This is particularly important for surgical simulators where the user wields the cutting tool. For games and other simulators it may not always be necessary to accurately simulate progressive cutting, as visual effects can be used to mask the creation of a cut.

A third possible extension is to explicitly represent the cutting tool. As long as mesh surface points are used as basis for the cut definition the rest of the simulation can remain unchanged. Depending on the

application it may be interesting to represent the cutting tool exactly in order to allow cutting cavities, cut opening displacements, and cut groove depths to follow the precise dimensions of the tool.

We can think of various improvements of our wound visualization as well. Firstly, our texture painting method suffers from issues with texture stretching and seams. We think that by using texture projection, or decals, many of these problems can be circumvented. Two recent approaches look promising: volume decals by Persson [Per11] and especially screen-space decals by Kim [Kim12]. Note that care must be taken to correctly rotate and scale a wound texture, and to prevent projection onto the interior geometry, perhaps by using stencil mapping techniques.

Secondly, it would be preferable to define a physically-correct wound visualization system by using tangible data about the appearance of wounds. Future work could focus on formulating a mathematical model for procedurally generating a natural-looking wound texture that follows physiological information about the formation and appearance of wounds. This could be done for representing both the interior wound and the surface wound. Such an approach would completely remove the need for artistic input, providing a generalized method that can be utilized for any application where realistic looking wounds are desired.

An alternative solution would be to extend localized subsurface scattering to render injured tissue without relying on additional wound textures at all. This would require extensive research about how injuries change the scattering of light in skin tissue, particularly with respect to erythema. Such information can then be used by a subsurface scattering shader to predict the appearance of injured skin. However, the feasibility of this is questionable, as it could be quite performance intensive and would likely not be able to visualize all types of wounds without additional rendering techniques.

Future work could also focus on researching the physiological properties of the various layers of skin tissue and how to translate this into a realistic physical simulation that closely emulates the behavior of real (injured) skin. This could also include blood fluid simulation for open wounds, for which several solutions already exist. Note however that blood simulation would not be preferable for applications that focus on the appearance of the interior tissue.

References

[AKR05] A. Al-Khalifah and D. Roberts. A survey of modeling approaches for medical simulators. *International Journal on Disability and Human Development*, 4(3):153–160, 2005.

[AMHH08] T. Akenine-Moller, E. Haines, and N. Hoffman. *Real-Time Rendering, Third Edition*. CRC Press, 2008.

[Anc15] M. Anchor. SimpleJSON. https://github.com/MJPA/SimpleJSON, 2010–2015.

[Bau72] B.G. Baumgart. Winged edge polyhedron representation. Technical report, DTIC Document, 1972.

[BG00] D. Bielser and M.H. Gross. Interactive simulation of surgical cuts. In *Computer Graphics and Applications, 2000. Proceedings. The Eighth Pacific Conference on*, pages 116–442. IEEE, 2000.

[BKP+10] M. Botsch, L. Kobbelt, M. Pauly, P. Alliez, and B. Levy. *Polygon Mesh Processing*. AK Peters, 2010.

[BL03] G. Borshukov and J.P. Lewis. Realistic human face rendering for the matrix reloaded. In *ACM Siggraph 2003 Sketches & Applications*. ACM Press, 2003.

[Bli77] J.F. Blinn. Models of light reflection for computer synthesized pictures. In *ACM SIGGRAPH Computer Graphics*, volume 11, pages 192–198. ACM, 1977.

[Bli78] J.F. Blinn. Simulation of wrinkled surfaces. *ACM SIGGRAPH Computer Graphics*, 12(3):286–292, 1978.

[BM10] K.M. Brown and M.E. Muscari. *Quick Reference to Adult and Older Adult Forensics: A Guide for Nurses and Other Health Care Professionals*. Springer Publishing Company, 2010.

[BMG99] D. Bielser, V.A. Maiwald, and M.H. Gross. Interactive cuts through 3-dimensional soft tissue. In *Computer Graphics Forum*, volume 18, pages 31–38. Wiley Online Library, 1999.

[BS87] P. Beckmann and A. Spizzichino. The scattering of electromagnetic waves from rough surfaces. *Norwood, MA, Artech House, Inc., 1987, 511 p.*, 1, 1987.

[BS01] C.D. Bruyns and S. Senger. Interactive cutting of 3d surface meshes. *Computers & Graphics*, 25(4):635–642, 2001.

[BSBK02] M. Botsch, S. Steinberg, S. Bischoff, and L. Kobbelt. OpenMesh – A generic and efficient polygon mesh data structure, 2002.

[BSM+02] C.D. Bruyns, S. Senger, A. Menon, K. Montgomery, S. Wildermuth, and R. Boyle. A survey of interactive mesh-cutting techniques and a new method for implementing generalized interactive mesh cutting using virtual tools. *The journal of visualization and computer animation*, 13(1):21–42, 2002.

[Cha60] S. Chandrasekhar. *Radiative Transfer*. Dover publications, 1960.

[CKS98] S. Campagna, L. Kobbelt, and H. Seidel. Directed Edges: A scalable representation for triangle meshes. *Journal of Graphics tools*, 3, 1998.

[COM98] J. Cohen, M. Olano, and D. Manocha. Appearance-preserving simplification. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 115–122. ACM, 1998.

[Cor09] T.H. Cormen. *Introduction to algorithms*. MIT press, 2009.

[Cor15] O. Cornut. ImGui. https://github.com/ocornut/imgui, 2014–2015.

[CT82] R.L. Cook and K.E. Torrance. A reflectance model for computer graphics. *ACM Transactions on Graphics (TOG)*, 1(1):7–24, 1982.

[dBCvKO08] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications (3rd Edition*. Springer, 2008.

[Del98] H. Delingette. Toward realistic soft-tissue modeling in medical simulation. *Proceedings of the IEEE*, 86(3):512–523, 1998.

[DJ05] C. Donner and H.W. Jensen. Light diffusion in multi-layered translucent materials. In *ACM Transactions on Graphics (TOG)*, volume 24, pages 1032–1039. ACM, 2005.

[dL07] E. d'Eon and D. Luebke. Advanced techniques for realistic real-time skin rendering. *GPU Gems*, 3(3):293–347, 2007.

[dLE07] E. d'Eon, D. Luebke, and E. Enderton. Efficient rendering of human skin. In *Proceedings of the 18th Eurographics conference on Rendering Techniques*, pages 147–157. Eurographics Association, 2007.

[DS03] C. Dachsbacher and M. Stamminger. Translucent shadow maps. In *Proceedings of the 14th Eurographics workshop on Rendering*, pages 197–201. Eurographics Association, 2003.

[Ebe03] D.S. Ebert. *Texturing & Modeling: A Procedural Approach, Third Edition*. Morgan Kaufmann, 2003.

[Ede01] H. Edelsbrunner. *Geometry and topology for mesh generation*. Cambridge University Press, 2001.

[Eri04] C. Ericson. *Real-time collision detection*. CRC Press, 2004.

[EVNT78] G. Eason, A.R. Veitch, R.M. Nisbet, and F.W. Turnbull. The theory of the back-scattering of light by blood. *Journal of Physics D: Applied Physics*, 11(10):1463, 1978.

[Fin03] A. Finkelstein. Subdivision surfaces. http://www.cs.princeton.edu/courses/archive/spr03/cs426/lectures/14-subdivision.pdf, 2003.

[FPW92] T.J. Farrell, M.S. Patterson, and B. Wilson. A diffusion theory model of spatially resolved, steady-state diffuse reflectance for the noninvasive determination of tissue optical properties invivo. *Medical physics*, 19(4):879–888, 1992.

[Fun99] T. Funkhouser. Modeling: Mesh Representations. www.cs.princeton.edu/courses/archive/fall99/cs426/lectures/mesh/mesh.pdf.gz, 1999.

[FvDFH95] J.D. Foley, A. van Dam, S.K. Feiner, and J.F. Hughes. *Computer Graphics: Principles and Practice in C (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[GM13] S. Gustavson and I. McEwan. Textureless Perlin noise and textureless simplex noise. https://github.com/ashima/webgl-noise, 2011–2013.

[Gos04] D. Gosselin. Real time skin rendering. In *Game Developer Conference, D3D Tutorial*, volume 9, 2004.

[Gre04] S. Green. Real-time approximations to subsurface scattering. *GPU Gems*, pages 263–278, 2004.

[Gri10] B. Grimes. Shading a Bigger, Better Sequel: Techniques in Left 4 Dead 2. http://www.valvesoftware.com/publications/2010/GDC10_ShaderTechniquesL4D2.pdf, 2010. Slides of a talk given at Game Developers Conference 2010.

[GSM04] D. Gosselin, P.V. Sander, and J.L. Mitchell. Real-time texture-space skin rendering. In W. Engel, editor, *ShaderX3: Advanced Rendering Techniques in DirectX and OpenGL*. Charles River Media, Cambridge, MA, 2004.

[Gus05] S. Gustavson. Simplex noise demystified. *Linköping University, Linköping, Sweden, Research Report*, 2005.

[HDD+93] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Mesh optimization. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 19–26. ACM, 1993.

[HK93] P. Hanrahan and W. Krueger. Reflection from layered surfaces due to subsurface scattering. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 165–174. ACM, 1993.

[INN05] T. Igarashi, K. Nishino, and S.K. Nayar. The appearance of human skin. Technical report, Department of Computer Science, Columbia University, 2005.

[Jam08] D. James. Combining hash values. http://www.boost.org/doc/libs/1_55_0/doc/html/hash/combine.html, 2005-2008.

[JB02] H.W. Jensen and J. Buhler. A rapid hierarchical rendering technique for translucent materials. In *ACM Transactions on Graphics (TOG)*, volume 21, pages 576–581. ACM, 2002.

[JG10] J. Jimenez and D. Gutierrez. *GPU Pro: Advanced Rendering Techniques*, chapter Screen-Space Subsurface Scattering, pages 335–351. AK Peters Ltd., 2010.

[JJG12a] J. Jimenez, A. Jarabo, and D. Gutierrez. Separable Subsurface Scattering. Technical Report Technical Report RR-02-12, Dpto. Informatica e Ingenieria de Sistemas, Universidad de Zaragoza, 2012.

[JJG+12b] J. Jimenez, A. Jarabo, D. Gutierrez, E. Danvoye, and J. von der Pahlen. Separable subsurface scattering and photorealistic eyes rendering. In *ACM SIGGRAPH*, 2012.

[JMLH01] H.W. Jensen, S.R. Marschner, M. Levoy, and P. Hanrahan. A practical model for subsurface light transport. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 511–518. ACM, 2001.

[JSB+10] J. Jimenez, T. Scully, N. Barbosa, C. Donner, X. Alvarez, T. Vieira, P. Matts, V. Orvalho, D. Gutierrez, and T. Weyrich. A practical appearance model for dynamic facial color. *ACM Transactions on Graphics (TOG)*, 29(6):141, 2010.

[JSG09] J. Jimenez, V. Sundstedt, and D. Gutierrez. Screen-space perceptual rendering of human skin. *ACM Transactions on Applied Perception (TAP)*, 6(4):23, 2009.

[JWSG10] J. Jimenez, D. Whelan, V. Sundstedt, and D. Gutierrez. Real-time realistic skin translucency. *IEEE computer graphics and applications*, 30(4):32–41, 2010.

[JZJ+15] J. Jimenez, K. Zsolnai, A. Jarabo, C. Freude, T. Auzinger, Wu X.C., J. von der Pahlen, M. Wimmer, and D. Gutierrez. Separable subsurface scattering. *Computer Graphics Forum*, 2015.

[Kaj86] J.T. Kajiya. The rendering equation. In *ACM Siggraph Computer Graphics*, volume 20, pages 143–150. ACM, 1986.

[KB04] A. Krishnaswamy and G.V.G. Baranoski. A biophysically-based spectral model of light interaction with human skin. In *Computer Graphics Forum*, volume 23, pages 331–340. Wiley Online Library, 2004.

[Ket99] L. Kettner. Using generic programming for designing a data structure for polyhedral surfaces. *Computational Geometry*, 13, 1999.

[Kim12] P. Kim. Screen space decals in Warhammer 40,000: Space Marine. In *ACM SIGGRAPH 2012 Talks*, page 6. ACM, 2012.

[Kin05] G. King. Real-time computation of dynamic irradiance environment maps. *GPU Gems*, 2:167–176, 2005.

[KSK01] C. Kelemen and L. Szirmay-Kalos. A microfacet based coupled specular-matte brdf model with importance sampling. In *Eurographics Short Presentations*, volume 25, page 34, 2001.

[LC10] C.Y. Lee and S. Chin. Interactive wound synthesis on 3d face using inverse projection mapping. http://bi.snu.ac.kr/~cylee/slides/2010_cacs_1204.pdf, 2010. Slides of a talk given at the 2010 International Congress on Computer Applications and Computational Science.

[LD05] A. Lagae and P. Dutré. An efficient ray-quadrilateral intersection test. *journal of graphics, gpu, and game tools*, 10:23–32, 2005.

[Len12] E. Lengyel. *Mathematics for 3D Game Programming and Computer Graphics, Third Edition*. Cengage Learning, 2012.

[LJD07] Y. Lim, W. Jin, and S. De. On some recent advances in multimodal surgery simulation: A hybrid approach to surgical cutting and the use of video images for enhanced realism. *Presence*, 16(6):563–583, 2007.

[LLC11] C.Y. Lee, S. Lee, and S. Chin. Multi-layer structural wound synthesis on 3D face. *Computer Animation and Virtual Worlds*, 22(2-3):177–185, 2011.

[LLM+10] T. Lewiner, H. Lopes, E. Medeiros, G. Tavares, and L. Velho. Topological mesh operators. *Computer Aided Geometric Design*, 27:1–22, 2010.

[Log09] A. Logg. Efficient representation of computational meshes. *International Journal of Computational Science and Engineering*, 4:283–295, 2009.

[Man11] D. Manocha. Triangle meshes. http://gamma.cs.unc.edu/COMP770/LECTURES/11trimesh.pdf, 2011.

[Mic14] Microsoft Corporation. Coordinate Systems. http://msdn.microsoft.com/en-us/library/windows/desktop/bb204853(v=vs.85).aspx, 2014.

[Mic16] Microsoft Corporation. Direct3D 11 Graphics. https://msdn.microsoft.com/en-us/library/windows/desktop/ff476080(v=vs.85).aspx, 2016.

[MK00] A.B. Mor and T. Kanade. Modifying soft tissue models: Progressive cutting with minimal new element creation. In *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2000*, pages 598–607. Springer, 2000.

[MT97] T. Möller and B. Trumbore. Fast, minimum storage ray-triangle intersection. *Journal of graphics tools*, 2:21–28, 1997.

[MV00] S.A. Mitchell and S.A. Vavasis. Quality mesh generation in higher dimensions. *SIAM Journal on Computing*, 29(4):1334–1370, 2000.

[MVN68] B.B. Mandelbrot and J.W. Van Ness. Fractional brownian motions, fractional noises and applications. *SIAM review*, 10(4):422–437, 1968.

[NRH+77] F.E. Nicodemus, J.C. Richmond, J.J. Hsia, I.W. Ginsberg, and T. Limperis. *Geometrical Considerations and Nomenclature for Reflectance*. National Bureau of Standards, 1977.

[NvdS01] H.W. Nienhuys and A.F. van der Stappen. A surgery simulation supporting cuts and finite element deformation. In *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2001*, pages 145–152. Springer, 2001.

[NvdS04] H.W. Nienhuys and A.F. van der Stappen. A delaunay approach to interactive cutting in triangulated surfaces. In *Algorithmic Foundations of Robotics V*, pages 113–130. Springer, 2004.

[Per01] K. Perlin. Noise hardware. *Real-Time Shading SIGGRAPH Course Notes*, 2001.

[Per02] K. Perlin. Improving noise. In *ACM Transactions on Graphics (TOG)*, volume 21, pages 681–682. ACM, 2002.

[Per04] K. Perlin. Implementing improved perlin noise. *GPU Gems*, pages 73–85, 2004.

[Per11] E. Persson. Volume Decals. *GPU Pro*, 2:115–120, 2011.

[PGK02] M. Pauly, M. Gross, and L. Kobbelt. Efficient simplification of point-sampled surfaces. In *Proceedings of the conference on Visualization 2002*, pages 163–170. IEEE Computer Society, 2002.

[PH10] M. Pharr and G. Humphreys. *Physically Based Rendering: From Theory To Implementation, Second Edition*. Morgan Kaufmann, 2010.

[Pho75] B.T. Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, 1975.

[Ros01] J. Rossignac. 3D Compression Made Simple: Edgebreaker on a Corner-Table. In *Shape Modeling and Applications, SMI 2001 International Conference on*. IEEE, 2001.

[RSC87] W.T. Reeves, D.H. Salesin, and R.L. Cook. Rendering antialiased shadows with depth maps. In *ACM SIGGRAPH Computer Graphics*, volume 21, pages 283–291. ACM, 1987.

[SAM09] P. Shirley, M. Ashikhmin, and S. Marschner. *Fundamentals of Computer Graphics, Third Edition*. CRC Press, 2009.

[SBJ07] C. Sussman and B.M. Bates-Jensen. *Wound Care: A Collaborative Practice Manual for Health Professionals*. Lippincott Williams & Wilkins, 2007.

[Sch94] C. Schlick. An Inexpensive BRDF Model for Physically-based Rendering. In *Computer graphics forum*, volume 13, pages 233–246. Wiley Online Library, 1994.

[She14] A. Sheffer. Mesh Basics: Definitions, Topology & Data Structures. http://www.cs.ubc.ca/~sheffa/dgp/ppts/date_structures.pdf, 2014.

[SHS01] D. Serby, M. Harders, and G. Székely. A new approach to cutting into finite element models. In *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2001*, pages 425–433. Springer, 2001.

[SSF06] W.U. Spitz, D.J. Spitz, and R.S. Fisher. *Spitz and Fisher's Medicolegal Investigation of Death: Guidelines for the Application of Pathology to Crime Investigation*. Charles C Thomas Publisher, 2006.

[SSF11] S.P Serna, A. Stork, and D.W. Fellner. Considerations toward a dynamic mesh data structure. In *SIGRAD Conference*, pages 83–90, 2011.

[SSKLK13] D. Shreiner, G. Sellers, J.M. Kessenich, and B.M. Licea-Kane. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, version 4.3*. Addison-Wesley Professional, 2013.

[SSL+07] G. Sela, J. Subag, A. Lindblad, D. Albocher, S. Schein, and G. Elber. Real-time haptic incision simulation using fem-based discontinuous free-form deformation. *Computer-Aided Design*, 39(8):685–693, 2007.

[Sta95] J. Stam. Multiple scattering as a diffusion process. In *Rendering Techniques 1995*, pages 41–50. Springer, 1995.

[Sup10] B. Supnik. To Strip or Not To Strip. http://hacksoflife.blogspot.nl/2010/01/to-strip-or-not-to-strip.html, 2010.

[TM06] R.F. Tobler and S. Maierhofer. A mesh data structure for rendering and subdivision. *WSCG2006 Short Papers Proceedings*, 2006.

[Vla10] A. Vlachos. Rendering Wounds in Left 4 Dead 2. http://www.valvesoftware.com/publications/2010/gdc2010_vlachos_l4d2wounds.pdf, 2010. Slides of a talk given at Game Developers Conference 2010.

[Wal16a] C. Walbourn. DirectX Tool Kit. http://directxtk.codeplex.com, 2013–2016.

[Wal16b] C. Walbourn. DirectXTex texture processing library. http://directxtex.codeplex.com, 2013–2016.

[Wav95] Wavefront Technologies. Wavefront Object File Format. http://www.martinreddy.net/gfx/3d/OBJ.spec, 1995.

[WDW13] J. Wu, C. Dick, and R. Westermann. Efficient collision detection for composite finite element simulation of cuts in deformable bodies. *The Visual Computer*, 29(6-8):739–749, 2013.

[Wil78] L. Williams. Casting curved shadows on curved surfaces. In *ACM Siggraph Computer Graphics*, volume 12, pages 270–274. ACM, 1978.

[WZW+05] D. Wang, Y. Zhang, Y. Wang, Y.S. Lee, P. Lu, and Y. Wang. Cutting on triangle mesh: local model-based haptic display for dental preparation surgery simulation. *Visualization and Computer Graphics, IEEE Transactions on*, 11(6):671–683, 2005.

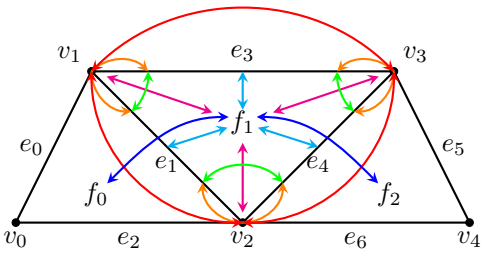
[ZPD02] H. Zhang, S. Payandeh, and J. Dill. Simulation of progressive cutting on surface mesh model. *DRAFT6-08 Sept02*, 2002.

Appendix A. Common mesh representations

In this appendix we review a number of commonly used mesh representations. We first introduce the most naive method of representing adjacency information and then describe representations that reduce memory requirements in various ways. These representations can be categorized depending on which mesh element takes precedence, and thus can be primarily face-based, edge-based, or vertex-based.

A.1 Adjacency lists

An adjacency list [Fun99, Fin03] is a naive and exhaustive approach to managing mesh connectivity. It maintains adjacency tables for each of the three mesh elements, and each element stores references to all its incident vertices, edges, and faces. An adjacency list provides efficient traversal: all local connectivity information can be found in $\mathcal{O}(1)$. However, it simultaneously consumes a large amount of memory, especially for general – not necessarily triangular – polygon meshes. An example of a full adjacency representation for a simple three-face mesh is shown in Figure A.1.



Vertex list
$v_0 = \langle v_1, v_2, e_0, e_2, f_0 \rangle$
$v_1 = \langle v_0, v_2, v_3, e_0, e_1, e_3, f_0, f_1 \rangle$
$v_2 = \langle v_0, v_1, v_3, v_4, e_1, e_2, e_4, e_6, f_0, f_1, f_2 \rangle$
$v_3 = \langle v_1, v_2, v_4, e_3, e_4, e_5, f_1, f_2 \rangle$
$v_4 = \langle v_2, v_3, e_5, e_6, f_2 \rangle$

Edge list
$e_0 = \langle v_0, v_1, e_1, e_2, e_3, f_0 \rangle$
$e_1 = \langle v_1, v_2, e_0, e_2, e_3, e_4, e_6, f_0, f_1 \rangle$
$e_2 = \langle v_0, v_2, e_0, e_1, e_4, e_6, f_0 \rangle$
$e_3 = \langle v_1, v_3, e_0, e_1, e_4, e_5, f_1 \rangle$
$e_4 = \langle v_2, v_3, e_1, e_2, e_3, e_5, e_6, f_1, f_2 \rangle$
$e_5 = \langle v_3, v_4, e_3, e_4, e_6, f_2 \rangle$
$e_6 = \langle v_2, v_4, e_1, e_2, e_4, e_5, f_2 \rangle$

Face list
$f_0 = \langle v_0, v_1, v_2, e_0, e_1, e_2, f_1 \rangle$
$f_1 = \langle v_0, v_1, v_3, e_1, e_3, e_4, f_0, f_2 \rangle$
$f_2 = \langle v_2, v_3, v_4, e_4, e_5, e_6, f_1 \rangle$

Figure A.1: A full adjacency data structure for a simple three-face surface. Colored arrows indicate connectivity for elements incident to face f_1. Vertex relationships: vertex-vertex (red), vertex-edge (orange), vertex-face (magenta). Edge relationships: edge-vertex (orange), edge-edge (green), edge-face (cyan). Face relationships: face-vertex (magenta), face-edge (cyan), face-face (blue).

Because storing comprehensive adjacency relationships is usually not desirable or necessary, a partial adjacency list [Fun99, Fin03] offers an alternative. For this structure only the most important adjacencies are stored explicitly and others are derived. The general strategy is to represent adjacency queries that are time-critical as explicitly as possible while minimizing the memory consumption.

A.2 Face-based representations

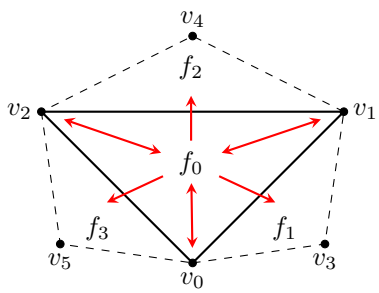
A.2.1 Face set

The simplest type of face-based mesh representation is the *face set* [Fun99, BKP+10, Man11, She14], which stores a set of individual faces represented by their vertex positions. For triangular meshes this is an array of triples of vertex coordinates. We know that adjacent faces share either one or two vertices due to the connectedness assumption. However, because this connectivity information is not explicitly described, vertices are replicated as many times as their degree (on average each vertex is stored six times). The only way to determine whether two faces are adjacent is to compare the vertex coordinates of all pairs of polygons, which is at best an $\mathcal{O}(n \log n)$ process for n faces [FvDFH95]. Additionally, roundoff errors in the coordinate quantities may cause faces to be disconnected, further complicating adjacency queries. Due to the inherent lack of explicitly representing the mesh connectivity, the face set is commonly referred to as a polygon soup.

Using 32 bit floating-point values for the vertex coordinates in $\mathbb{R}^3$, one triangle face uses up $3 \times 3 \times 4 = 36$ bytes of memory. Since $F \approx 2V$, the structure will consume 72 byes per vertex on average. Since this is not particularly memory efficient and the lack of connectivity information makes topological traversal extremely difficult, this is not considered to be an efficient data structure to support mesh processing algorithms.

A.2.2 Face-vertex table

Face sets can be modified to eliminate the vertex redundancy problem. The resulting representation is called an *indexed face set* or *face-vertex table* [Fun99, BKP+10, She14]. See Figure A.2. Vertices can be shared by storing them in a separate vertex table and by encoding faces as tuples of references to vertex entries. Note that the geometry in the vertex table is stored separately from the topological information in the face table. Although indexed face sets share vertices between faces, explicit adjacency information is still missing, and thus only the `FV` query can be answered in $\mathcal{O}(1)$ time.



Face list
$f_0 = \langle v_0, v_1, v_2 \rangle$
$f_1 = \langle v_0, v_3, v_1 \rangle$
$f_2 = \langle v_2, v_1, v_4 \rangle$
$f_3 = \langle v_0, v_2, v_3 \rangle$

Vertex list
$v_0 = \langle 2.0, 0.0, 0.0 \rangle$
$v_1 = \langle 4.0, 2.0, 0.0 \rangle$
$v_2 = \langle 0.0, 2.0, 0.0 \rangle$
$v_3 = \langle 3.75.0.25.0, 0.0 \rangle$
$v_4 = \langle 2.0, 3.0, 0.0 \rangle$
$v_5 = \langle 0.25, 0.25, 0.0 \rangle$

Figure A.2: An indexed face set for a three-face mesh. Each face record stores references to three vertex records in the vertex list, thus making vertex sharing possible. Optional adjacency information for face f_0 is highlighted. Image based on [BKP+10].

A vertex table uses 12 bytes per vertex (3 coordinates $\times$ 4 bytes), and the face table requires about 24 bytes (3 indices $\times$ 4 bytes for a pointer/integer) per triangle. Since on average two triangles reference a particular vertex, a total of 36 bytes per vertex is stored. Because this representation is compact and

contains the minimum amount of information for rendering, it is used in popular file formats for mesh storage (such as the Wavefront OBJ file), and is usually the only accepted input for rendering algorithms. However, without additional connectivity information, this data structure requires exhaustive searches to retrieve local adjacency information for a mesh element and is thus not efficient enough for most processing algorithms.

A.2.3 Triangle lists/strips/fans

Hardware accelerators use face-vertex tables as input to the rendering pipeline [Mic16, SSKLK13]. Here, faces are defined by a list of indices that point to vertices in the vertex table. Such a list of indices can be organized using triangle lists or triangle strips [BKP+10]. Triangle lists simply store successive triples of indices, where each triple describes the vertices incident to a triangle face. Triangle strips take advantage of the fact that a connected and closed mesh can be described by end-to-end adjacent triangle faces. Instead of specifying triangles individually, triangle strips only define the first triangle normally, and each successively added index will define a new triangle that uses the last two indices of the previous triangle. Using triangle strips will reduce the size of the index list by three. Although this property seems appealing, meshes cannot in general be represented by strips without lengthy construction times. Maintaining triangle strips is also much more complex for non-static meshes than maintaining triangle lists.

A.2.4 Face-based connectivity

Additional adjacency can be stored in an indexed face set by letting faces store additional references to their neighboring faces, and letting each vertex store a reference to one (or all) of the faces it is incident to. See the left side of Figure A.2. This representation is called a face-based connectivity structure [BKP+10], and is primarily applicable for triangle meshes. The connectivity information described by this structure allows efficient adjacency traversal between faces and vertices, as well as obtaining a one-ring neighborhood of vertices or faces around a given vertex.

In addition to the 36 bytes per vertex used by the regular face-vertex table, 24 bytes per vertex are necessary for the triples of references to neighbor faces ($2 \times 3 \times 4$ bytes), and storing one face reference for each vertex requires 4 bytes. Thus a total of 64 bytes per vertex is consumed, which is still less than the naive face set. A major drawback of this data structures is that it does not explicitly store edges, and thus any mesh traversal query that involves edges cannot be performed atomically. Another issue is that it is not as efficient for general polygon meshes due to increased memory consumption for faces.

A.3 Edge-based representations

A.3.1 Winged-edge

Because edge-based data structures encode almost all connectivity in the mesh edges they are well-suited to represent general planar meshes in which the faces are not necessarily triangles. The traditional winged-edge data structure [Bau72] stores the three mesh elements in separate tables. Vertices and faces store references to one of their incident edges from which all other connectivity may be acquired. Edges store references to their two endpoint vertices, to two incident faces (left and right of the edge), and to the next and previous edge within its left and right face (see Figure A.3. Whether a face is to the left or right of an edge depends on the order in which the edge endpoints are accessed, and thus edges represent both directions (edges are unoriented). Provided that all faces are simply connected, this information suffices to fully describe the mesh and to recover all adjacency information in $\mathcal{O}(1)$ time.

Winged-edge structures are particularly suited to provide high flexibility for meshes that need to support dynamically changing geometry or topology. However, rendering winged-edge meshes requires generating

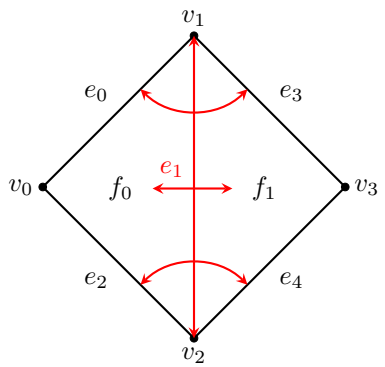


Figure A.3: Connectivity in an edge-based structure such as the winged-edge representation. Each edge references two incident vertices, two incident faces, and four neighboring edges – two on each side of the edge. Additionally, each vertex and face references one of their incident edges (not shown). Image based on [BKP+10].

a vertex-face table each time the geometry changes. Another drawback are the high memory requirements: each vertex uses 16 bytes (three floats and an edge reference), each face uses 4 bytes for an edge reference, and each edge stores a total of eight references or 32 bytes amounting to a total of $16 + 2 \times 4 + 3 \times 32 = 120$ bytes per vertex (since $F \approx 2V$ and $E \approx 3V$). Additionally, it suffers from increased complexity due to complex reference maintenance. Because of their large storage requirements winged-edges are nowadays commonly discarded in lieu of half-edge data structures.

A.3.2 Half-edge

Halfedge data structures [Ket99, BSBK02] are an evolution of edge-based data structures where each unoriented edge is decomposed into two halfedges with opposite orientations. Therefore, they can be used to represent arbitrary polygonal meshes that are orientable manifolds. A halfedge stores a reference to an incident vertex, an incident face, the opposite halfedge, and to the next and previous halfedge on the boundary of that face (see Figure A.4. It is a matter of convention whether the source or target vertex is the one chosen to be stored in a halfedge [Ket99]. Whether the left or right face is stored and which edge is considered to be the next in the chain depends on the mesh orientation described earlier: edges and vertices around a face have either clockwise or counterclockwise winding order. Moreover, each face stores a reference to one of its halfedges, and each vertex stores an outgoing halfedge. Like winged-edge representations, halfedge data structures are particularly well suited to represent meshes whose faces have an arbitrary number of edges. Additionally, adjacent elements for each vertex, edge, or face can be efficiently enumerated and especially one-ring neighborhood traversals are well supported; these queries can be performed in $\mathcal{O}(1)$ time. Each vertex requires $3 \times 4 + 4 = 16$ bytes, each face 4 bytes, and each edge $5 \times 4 = 20$ bytes. Since the number of halfedges is on average six times the number of vertices (and $F \approx 2V$), the total memory consumption is $16 + 2 \times 4 + 20 \times 6 = 144$ bytes per vertex. Not explicitly storing the previous halfedge reduces the memory cost to 120 bytes per vertex. Twin edges can also be stored implicitly by grouping them in pairs in the edge table, further reducing storage costs to 96 bytes per vertex, but this is not always practical. Some restrictions of the halfedge data structure are not being able to represent non-orientable surfaces and its creation is a cumbersome and time-consuming affair.

A.3.3 Directed-edge

The directed edge data structure [CKS98] is a memory-efficient variation of the halfedge structure specifically designed for triangular meshes. It is based on the idea that some of the connectivity information can be encoded implicitly with indices. The three halfedges belonging to a common triangle are grouped together and their indices are given as $H = \{3f + i \mid i \in \{0, 1, 2\}\}$ for given face index f. If h denotes

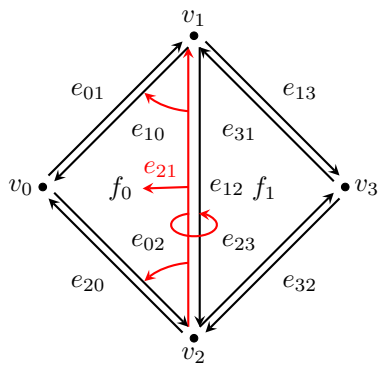


Figure A.4: Connectivity in a halfedge-based structure. As opposed to the edge-based structure, edges are essentially duplicated and given an explicit direction. Each halfedge references one of its two adjacent vertices, the face it is incident to, the next edge in the cycle around that face, its opposite or twin edge pointing in the other direction, and optionally the previous edge in the cycle. Additionally, each vertex and face references one of its incident edges. Image based on [BKP+10].

the index of a halfedge, then the index of its adjacent face is given by $(h/3)$, and its index within that face is $(h \mod 3)$. The index of the next halfedge can be found with $((h+1) \mod 3)$.

Other connectivity information is stored explicitly: each vertex stores an index to an outgoing halfedge, and each halfedge stores the indices of its opposite halfedge and source vertex. This way the memory consumption is reduced to 16 bytes per vertex and 8 bytes per halfedge, which is $16 + 6 \times 8 = 64$ bytes per vertex. Its drawbacks are the restriction to pure triangle or quadrilateral meshes, the complexity of maintaining indices, and the lack of explicit representation of edges.

A.4 Render-optimized representations

Many mesh representations neglect to account for rendering issues and merely describe the topological component of the mesh. In this section we describe two representations that attempt to combine both components.

A.4.1 Tobler and Maierhofer mesh representation

Tobler and Maierhofer [TM06] proposed a data structure for rendering and subdivision algorithms that maintains rendering and topological data separately. Geometric information is represented by an indexed face set, where each index points to an entry in a position array and optional attribute arrays. Topological information is described by indexed lists of faces, edges and vertices. Instead of storing the face list explicitly, faces reuse the references to their vertices defined in the face set. Additionally, each vertex stores references to any adjacent edge, and each edge stores references to the two incident faces.

Figure A.5 illustrates the local connectivity for a single face. Since an index on its own does not contain any information about the orientation of a referenced element, parallel lists must be maintained to store on which side the source element is compared to the referenced element, resulting in a memory increase of about 25%. Maintaining mesh elements using indexed lists complicates updating the connectivity whenever the topology is modified. For this reason the authors have chosen to disallow dynamic modification of topological information. Such algorithms must instead be implemented in such as way that they output a new mesh with the modified topology. Tobler and Maierhofer claim that the memory consumption of this representation is 26 bytes for geometric data and 37 bytes for topological data per triangle face, which would amount to about 126 bytes per vertex.

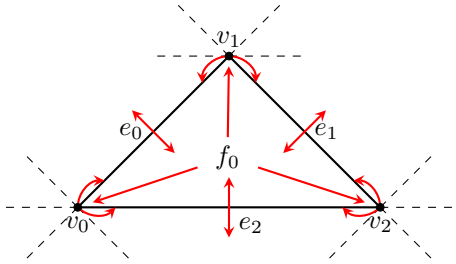


Figure A.5: Connectivity information for a single face in the mesh data structure by Tobler and Maierhofer, wherein each face references its incident vertices, each vertex references all of its incident edges (only two are shown here), and each edge references its two adjacent faces.

A.4.2 Corner table

Another representation that is convenient for rendering is the vertex-based *corner table* data structure [Ros01]. In it, a corner represents the interior of a vertex with respect to one of its incident faces, and and stores all connectivity information for that corner: the associated vertex and face, the next and previous corners in the face, the opposite corner, and the left and right corners. This wealth of connectivity information allows queries to be answered in $\mathcal{O}(1)$ time. However, it suffers from having high redundancy, memory consumption, high construction time, and high maintenance complexity.

Appendix B. Skin rendering

One of the goals of this thesis is to produce convincing renderings of human skin. This chapter describes the theory and implementation of the shading techniques and algorithms for the skin renderer used in this thesis.

Modern real-time skin rendering can be separated into a surface reflection part and a subsurface scattering part. In our implementation, surface reflection consists of following components: ambient lighting, diffuse lighting, specular lighting, bump mapping, and shadow mapping. The subsurface scattering consists of subsurface reflection and transmittance. Both parts are combined in order to produce high quality real-time renderings of skin. We base the majority of the skin rendering implementation on the work by d'Eon et al. [dL07, dLE07] and Jimenez et al. [JG10, JWSG10, JJG12a].

B.1 Surface reflection

A small fraction of incoming light onto skin is reflected at the surface. This surface reflection is determined by computing the outgoing radiance towards the view direction for every point on the surface. In real-time graphics, radiance (or light intensity) is expressed as an RGB value that contributes to the final color of a rendered pixel. Computing outgoing radiance involves solving the reflectance equation [AMHH08], a simplification of the rendering equation [Kaj86] for non-emitting surfaces:

$$L_o(p, \boldsymbol{v}) = \int_{\Omega} f(\boldsymbol{l}, \boldsymbol{v})\ L_i(p, \boldsymbol{l}) \cos\theta_i\ d\omega_i,$$

where $L_o(p, \boldsymbol{v})$ is the outgoing radiance at surface point p in view direction $\boldsymbol{v}$, Ω is the hemisphere of directions at p, $L_i(p, \vec{l})$ is the incoming radiance at p from light direction $\boldsymbol{l}$, and θ_i is the angle between $\boldsymbol{l}$ and surface normal $\boldsymbol{n}$ at p. The remaining factor, $f(\boldsymbol{l}, \boldsymbol{v})$, is a bidirectional reflectance distribution function (BRDF) that describes how much of the light from incident direction $\boldsymbol{l}$ reflects towards the viewing direction $\boldsymbol{v}$ at a particular point p on the surface. It is defined as the ratio between differential outgoing radiance and differential irradiance [AMHH08, PH10]:

$$f(\boldsymbol{l}, \boldsymbol{v}) = \frac{dL_o(p, \boldsymbol{v})}{L_i(p, \boldsymbol{l}) \cos\theta_i\ d\omega_i}.$$

Empirical BRDFs such as Lambertian reflection and the Blinn-Phong reflection model [Bli77] are efficient for real-time rendering but are not physically realistic. Physically-based BRDFs have the additional constraint of energy conservation and often account for micro-geometry, producing more realistic results.

B.1.1 Physically-based reflectance

Following d'Eon and Luebke [dL07] we compute specular reflection using the Kelemen/Szirmay-Kalos specular BRDF [KSK01], which is an approximation of the Cook-Torrance reflection model [CT82]. The Cook-Torrance model is a physically-based BRDF that uses microfacet theory to determine the roughness of the surface, and accounts for Fresnel reflection, microfacet orientation, occlusion between microfacets,

and absorption. Kelemen and Szirmay-Kalos simplify the geometric term (occlusion), resulting in a BRDF that is more attractive for real-time rendering:

$$f_{KSK}(\boldsymbol{n}, \boldsymbol{l}, \boldsymbol{v}) = D(\boldsymbol{n}, \boldsymbol{h}) \frac{F(\boldsymbol{v}, \boldsymbol{h})}{\boldsymbol{h'} \cdot \boldsymbol{h'}},$$

where $\boldsymbol{l}$ is the light direction, $\boldsymbol{v}$ is the view direction, $\boldsymbol{h'} = \boldsymbol{l} + \boldsymbol{v}$ is the non-normalized half-way vector, and $\boldsymbol{h} = \boldsymbol{h'}/|\boldsymbol{l} + \boldsymbol{v}|$ is the normalized half-way vector. Furthermore, $D(\boldsymbol{n}, \boldsymbol{h})$ is the Beckmann normal distribution function [BS87] that describes the fraction of the microfacets oriented towards $\boldsymbol{h}$ and thus contribute to the reflection of light from $\boldsymbol{l}$ to $\boldsymbol{v}$:

$$D_{Beckmann}(\boldsymbol{n}, \boldsymbol{h}, m) = \frac{1}{m^2 \cos^4 \alpha} \exp(-\left(\frac{tan^2 \alpha}{m^2}\right)),$$

where α is the angle between $\boldsymbol{n}$ and $\boldsymbol{h}$, and m is a measure of the roughness. Finally, $F(\boldsymbol{v}, \boldsymbol{h})$ is the Fresnel reflectance term that describes the amount of reflected light from each microfacet.

In order to improve the performance of the skin renderer, we follow d'Eon and Luebke [dL07] recommendation to precompute and render the Beckmann distribution to an 8-bit texture map, and then read out during runtime. Moreover, we use Schlick's approximation [Sch94] to compute the Fresnel reflectance term:

$$F(\boldsymbol{v}, \boldsymbol{h}) = R_0 + (1 + R_0)(1 - \cos(\boldsymbol{v} \cdot \boldsymbol{h}))^5$$
$$R_0 = \left(\frac{\eta_1 - \eta_2}{\eta_1 + \eta_2}\right)^2,$$

where η_1 and η_2 are the indices of refraction of the two media at the interface. For a typical scene we have $\eta_1 = 1$ for air and $\eta_2 = 1.4$ for skin, resulting in a constant value of $R_0 \approx 0.028$.

Aside from specular reflection, the Kelemen/Szirmay-Kalos BRDF also includes a diffuse (or matte) component. In the interest of energy conservation, the authors propose a coupled interdependent diffuse component that is proportional to albedo of the specular component. D'Eon and Leubke [dL07] describe a method of adding energy conservation using a precomputed texture, but note that the visual impact should be evaluated on a per-application basis. Note that the diffuse component is actually traditionally used as a very crude approximation of subsurface scattering [JMLH01]. For simplicity we have chosen to use Lambertian diffuse reflectance, since this layer is modified by a subsurface scattering shader later.

Using the Kelemen/Szirmay-Kalos specular BRDF, the specular radiance at a point p towards view direction $\boldsymbol{v}$ from a single light source with direction $\boldsymbol{l}$ becomes

$$L_{spec}(p, \boldsymbol{v}) = k_{spec}\ D_{Beckmann}(\boldsymbol{n}, \boldsymbol{h}, m)\ \frac{F(\boldsymbol{v}, \boldsymbol{h})}{\boldsymbol{h'} \cdot \boldsymbol{h'}}\ I_L(p, \boldsymbol{l})\ \boldsymbol{n} \cdot \boldsymbol{l},$$

where I_{spec} is a global specular intensity factor and $I_L(p, \boldsymbol{l})$ is the incident radiance from light source L. In our application we use spotlights because they represent real-life lights quite well while having a fairly simple equation:

$$I_L = \boldsymbol{c}_L \cdot f_{attenuation} \cdot f_{falloff},$$

where $\boldsymbol{c}_L$ is the color of the light expressed as an RGB value, and functions $f_{attenuation}$ and $f_{falloff}$ determine the light intensity based on the distance from the source. See any graphics textbook for more information about spotlights. The equation for diffuse radiance is similar:

$$L_{diff}(p, \boldsymbol{v}) = \boldsymbol{c}_{albedo}\ I_L(p, \boldsymbol{l})\ \boldsymbol{n} \cdot \boldsymbol{l},$$

where $\boldsymbol{c}_{albedo}$ is a sampled RGB value from a color texture map that provides the diffuse skin color. Final specular and diffuse radiance for a surface point are computed straightforwardly as summations over each light source.

Note that the specular radiance is modified by two factors: specular intensity k_{spec} and specular roughness m. By sampling these quantities from a two-channel specular texture we allow them to vary over the

surface of a model, improving the fidelity of the specular component. Additionally, we sample a normal map to perturb the surface normal n located at point p that is used in the equation above. Like specular mapping, normal mapping [Bli78, COM98] allows us to vary the roughness over the surface so that meso-features such as bumps, wrinkles and pores can be represented. Note that surface features can be classified into three scales: macro-features, meso-features, and micro-features [AMHH08]. Macro-features and micro-features were already modeled by the mesh geometry and the BRDF, respectively.

B.1.2 Global illumination

The diffuse and specular reflectance BRDFs in the previous section are local illumination models, and fail to account for indirect lighting. They only use the information at the surface point to compute the lighting. In global illumination techniques however, information from other points (and sometimes objects) is used in order to improve realism and to provide visual cues for spatial relationships [AMHH08].

B.1.2.1 Ambient light

A simple yet important model of indirect illumination is ambient light, which has a constant intensity L_A (either scalar quantity or RGB value) and is direction invariant. In light transport theory ambient light is explained by rays emitting from light sources that scatter from other surfaces before arriving at their final destination. Although ambient light could be simulated in its entirety with ray tracing methods, this is much too expensive for real-time computer graphics. By assuming that this type of illumination is constant and view-independent, the ambient radiance term L_{amb} for a Lambertian surface can be approximated as the product of the albedo color $\boldsymbol{c}_{albedo}$ and ambient intensity L_A [AMHH08].

So far the ambient lighting equation results in a completely flat color due to the lack of any directional variation. Ambient occlusion attempts to remedy this by adding soft shadows based on how much each surface point is exposed to ambient lighting. Because ambient occlusion is not dependent on individual light sources like other types of shadowing, we can be precompute and store the ambient occlusion scalar $k_A \in [0, 1]$ that ranges from full occlusion to no occlusion in a single-channel texture map. This new value is sampled during runtime for a given surface point and multiplied with the albedo color and ambient intensity.

As a final addition we account for incoming radiance from environmental lights in all directions that are too far away to consider directly, yet have a significant contribution to ambient lighting. We use precomputed scene-specific irradiance environment maps [Kin05, AMHH08] as an approximation for this type of global illumination. We thus compute the total contribution from ambient lighting as

$$L_{amb} = \boldsymbol{c}_{albedo} \otimes \boldsymbol{c}_{irradiance} \cdot L_A \cdot k_A,$$

where $\boldsymbol{c}_{irradiance}$ is a sampled RGB value from an irradiance map.

B.1.2.2 Shadows

Another important global illumination feature which adds depth and realism are shadows. Shadows cast by static environmental lights were already taken into consideration by ambient occlusion. Now we will add shadows cast by dynamic lights, for which we implement a technique called shadow mapping [Wil78]. In shadow mapping the depth-buffer is used to store the distance (depth) to the nearest geometric obstruction for each light source. The contents of these depth-buffers are stored in so-called shadow maps. When rendering the geometry with respect to the camera, the following depth test is executed: if the rendered geometric point is further away than the stored depth for a given light source, it is occluded by some other geometry, and thus in shadow.

In its naive form, shadow mapping produces hard and blocky shadows. Pseudo-soft shadowing can be achieved by applying percentage-closer filtering [RSC87] during rendering. This technique attempts to simulate soft shadows produced by area lights by considering that the amount of light reaching a surface

point is proportional to the visible surface area of a light source from that point. Since spotlights were modeled to emit light from only a single point, soft shadows obviously cannot be computed from the point of view of the light. Percentage-closer filtering reverses the process by approximating the visibility of the light source using a set of surface locations close to the original point. Technically, it works similarly to texture sampling. During rendering, instead of taking a single sample from the shadow map, the four nearest samples are also retrieved. The results of all depth tests are then bilinearly interpolated to determine the percentage of samples that are visible to the light source. The process results in anti-aliased shadows for a minimal loss in performance.

For each surface point, the diffuse and specular components described above are multiplied by the scalar shadow factor k_S resulting from percentage-closer filtering.

B.2 Subsurface scattering

The reflectance equation alone does not properly account for light penetrating the surface of the skin where it is then scattered and absorbed by the interior particles. Subsurface scattering can cause light to exit the material at a different point than where it entered, where subsurface reflection occurs due to back-scattering and transmittance due to forward-scattering [PH10]. Transmittance usually only occurs when light is able to travel long distances in the medium, whereas subsurface reflectance is almost always present in translucent materials since it occurs over much smaller distances.

Accurately rendering translucent materials such as skin requires solving the subsurface scattering equation, a simplification of the full equation of transfer [Cha60] for a finite medium:

$$L_o(p, \boldsymbol{v}) = \int_A \int_\Omega S(p, \boldsymbol{v}, q, \boldsymbol{l})\ L_i(p, \boldsymbol{l})\ cos\theta_i\ d\omega_i\ dA,$$

where the integration is over both area A (points on the surface being rendered) and incident direction ω. Here, $S(p, \boldsymbol{v}, q, \boldsymbol{l})$ is a bidirectional scattering-surface reflectance distribution function (BSSRDF). This function generalizes the BRDF to account for light that exits the surface at a point other than where it entered, and gives the proportion of light incident at position q from direction $\boldsymbol{l}$ that radiates out from position p into viewing direction $\boldsymbol{v}$:

$$S(p, \boldsymbol{v}, q, \boldsymbol{l}) = \frac{dL_o(p, \boldsymbol{v})}{dE(q, \boldsymbol{l})}.$$

By using a BSSRDF instead of a BRDF we are able to produce a soft skin appearance. Note that accounting for subsurface scattering is yet another implementation of global illumination.

B.2.1 Subsurface reflection

A common analytic method of solving the subsurface scattering equation is with diffusion theory [Sta95]. Jensen et al. [JMLH01, JB02] use diffusion theory to arrive at a dipole diffusion approximation for the BSSRDF that depends only on the scattering properties of the material, the Fresnel transmittance F_t at p and q, and the distance between these points:

$$S(p, \boldsymbol{v}, q, \boldsymbol{l}) = \frac{1}{\pi} F_t(q, \boldsymbol{l})\ R(\|q - p\|)\ F_t(p, \boldsymbol{v}),$$

where $R(r)$ is the diffusion profile of the material. The Fresnel transmittance terms describe the amount of light transferred from one medium to another. The $\frac{1}{\pi}$ term describes the amount of incoming light that is reflected towards a single direction ($\boldsymbol{v}$) on the hemisphere around p.

Donner and Jensen [DJ05] later extended the dipole approximation to a multipole model (using a sum of dipoles) that better simulates light transfer in multi-layered materials and therefore proves a better fit for the appearance of skin. Using a method for approximating (reflectance and transmittance) diffusion profiles of multi-layered materials, they computed a complete three-layer model for human skin that uses measured scattering parameters.

A diffusion profile is an approximation of the scattering behavior of light inside a translucent material. It describes how much of the light incident to a single surface point emerges as a function of angle and distance. However, because light diffuses so quickly, diffusion profiles can be considered to be radially symmetric with minimal loss of accuracy. Note that the wavelength of light influences the scattering behavior – which is the reason for the characteristic reddish hue of skin – and thus diffusion profiles are strongly color dependent. Using diffusion profiles, simulating subsurface scattering amounts to scattering incoming diffuse light into neighboring locations based on the exact shapes of the profiles. This will result in a soft, translucent appearance. Unfortunately, the dipole and multipole models are too expensive for use in real-time applications.

B.2.1.1 Texture-space diffusion

D'Eon et al. [dL07, dLE07] make the key observation that the diffusion profiles predicted by the dipole and multipole models can be approximated by a weighted sum of Gaussians, allowing subsurface scattering to be computed in real-time. Mapping from a diffusion profile $R(r)$ to a Gaussian sum involves finding appropriate weights w_i and variances v_i for k Gaussians:

$$R(r) \approx \sum_{i=1}^{k} w_i \; G(v_i, r),$$

where the Gaussian of variance v is defined as

$$G(v_i, r) = \frac{1}{2\pi v} \; \exp(-r^2/2v).$$

Constant $\frac{1}{2\pi v}$ ensures that each Gaussian has unit total diffuse response, or in other words, the total area under the Gaussian function is equal to one:

$$\int_0^\infty 2\pi r \; G(v, r) \; dr = 1.$$

Each diffusion profile can thus ne approximated as a linear combination of Gaussian convolutions. D'Eon et al. found that four Gaussians are sufficient to model most single-layered materials, and presented a six-Gaussian fit for Donner and Jensen's three-layer model for Caucasian skin; see Table B.1. The Gaussian weights for each profile sum to one because the color of the skin is defined by a color map instead of embedding it into the diffusion profiles themselves. D'Eon et al. describe that the six Gaussians for the red color channel can also accurately be used for the green and blue color channels so as to reduce cost.

Variance (mm^2)	Red weights	Green weights	Blue weights
0.0064	0.233	0.455	0.649
0.0484	0.100	0.336	0.344
0.1870	0.118	0.198	0.000
0.5670	0.113	0.007	0.007
1.9900	0.358	0.004	0.000
7.4100	0.078	0.000	0.000

Table B.1: Six-Gaussian fit for a three-layer skin model [dL07].

Note that the BSSRDF in the subsurface scattering equation actually expresses a two-dimensional surface convolution with the radially symmetric non-separable diffusion profile $R(r)$ [dLE07]. In texture-space diffusion [BL03, Gre04], this process is simulated by rasterizing irradiance (incoming diffuse light) into a

texture, applying a convolution operation (blur filter) onto that texture, and texture mapping the result back on the mesh. The desired shape of the blur filter exactly matches with the diffusion profiles of the material.

Although the exact diffusion profile could directly be evaluated in an expensive two-dimensional convolution pass on the irradiance texture, d'Eon et al. note that the separability property of Gaussians allows us to reduce this to a set of cheaper one-dimensional convolutions. By expressing the non-separable diffusion profile as a Gaussian sum, irradiance convolution can be evaluated separably and hierarchically, producing a series of convolved irradiance textures with increasingly wider blur kernels. The weighted sum of these convolution textures then approximates the convolution of irradiance by the original diffusion profile. In combination with the sampled albedo this will produce the final skin tone due to subsurface scattering.

B.2.1.2 Screen-space diffusion

Although the texture-space diffusion approximation delivers real-time frame rates, it suffers from a number of drawbacks that make it sluggish and cumbersome to use. Computing a series of irradiance convolution textures on every frame for every translucent object is still relatively costly, causing performance to quickly deteriorate in the case of multiple subjects on screen. To solve these and other issues, Jimenez et al. propose to translate the evaluation of the diffusion approximation from texture space to screen space. In recent work they presented a separable screen-space diffusion approximation method. The skin renderer for this thesis implements the subsurface scattering technique described in [JJG12a, JJG+12b]. This method was chosen because it provides a high-performance non-proprietary state-of-the-art solution for real-time skin rendering that can easily be integrated as a post-process shader.

We have seen that a diffusion profile performs a two-dimensional convolution on the irradiance, which was approximated by a Gaussian sum in order to achieve real-time frame rates. Jimenez et al. [JJG12a, JJG+12b] propose a new approach where the non-separable two-dimensional diffusion kernel is decomposed into two one-dimensional kernels, avoiding the costly computation of convolution textures.

Two-dimensional diffusion kernel $S[x, y]$ is decomposed into cheaper horizontal and vertical one-dimensional kernels by defining a separable one-dimensional filter $s[x]$:

$$S[x, y] \approx s[x] \times s^T[y],$$

where $[x, y]$ is the sample location and $s^T[y]$ is the transpose of $s[x]$. Using a transpose filter is possible due to the radial symmetry of the diffusion profile. Filter $s[x]$ is defined as a function of an initial diffusion profile $p[x]$ and is parameterized to allow it to be better fitted to the exact target diffusion profile:

$$s(w, \boldsymbol{t}, \boldsymbol{f})[x] = p\left[\frac{wx}{\boldsymbol{f}}\right]\boldsymbol{t} + \delta(x)(1 - \boldsymbol{t}); \qquad \delta(x) = \begin{cases} 1 & \text{for the center sample} \\ 0 & \text{otherwise} \end{cases}.$$

Width w defines the global width of the subsurface scattering kernel in world-space units. Strength $\boldsymbol{t}$ specifies how much of the diffuse light penetrates the skin and thus will be modified by the subsurface scattering. It can be seen as a mix factor between the original unfiltered image and the blurred image. Falloff $\boldsymbol{f}$ defines the shape of the gradient with distance. Large falloffs spread the shape making it wider, while small falloffs make it narrower. Whereas the width is a scalar, uniform over the RGB channels, strength and falloff are spectral values, defined on a per-channel basis.

These three parameters are optimized to find the kernel that best fits the exact target two-dimensional diffusion kernel S[x]. Optimization yields the following quantities for the width, strength and falloff: $w = 0.014$, $\boldsymbol{t} = \{0.78, 0.70, 0.75\}$, $\boldsymbol{f} = \{0.57, 0.13, 0.08\}$. Although in our implementation we use these values directly, they can also be adjusted to tune the appearance of subsurface scattering.

Initial one-dimensional profile $p[x]$ is approximated by the red channel of the six-Gaussian skin profile described in Table B.1. Jimenez et al. note that applying this profile to the green and blue channels and factoring it by the spectral falloff parameter produces visually indistinguishable results to using the actual green and blue profiles.

Rendering with a separable diffusion kernel is simply reduced to applying a two-pass convolution filter over the rendered diffuse image. In order to perform a convolution with one-dimensional kernel $s[x]$ a discretization of the kernel is necessary. This is done by computing the sample offsets $\mathcal{X}$ around the center location, and the spectral weights $\mathcal{W}$ of each sample based on the initial profile and the strength and falloff parameters. A tradeoff between the quality and performance must be made when defining the number of samples in the kernel. Jimenez et al. note that using between 9 and 19 samples gives reasonable results. Since the visual impact is minimal, we have elected to use only 9 samples for performance reasons.

Sample offsets $\mathcal{X}$ can be computed by spacing them out uniformly over the kernel range and then modifying them with an exponential function to give higher importance to samples closer to the center. The kernel range defines the maximum extent of x when sampling diffusion profile $p[x]$, and thus influences the accuracy of the discretization process. A kernel ranging from -2 to 2 is sufficient when using 19 samples or less. For n samples the initial set of uniformly spaced offsets is computed as

$$\mathcal{X}_u = -r + (2r/n - 1)i \qquad \text{for } i = 0 \dots n-1,$$

where r is half the absolute kernel range. As an example, for $n = 9$ and $r = 2$ the following values are obtained: $\{-2.0, -1.5, -1.0, -0.5, 0.0, 0.5, 1.0, 1.5, 2.0\}$. Since it is desirable to have a higher sampling density around the center, the set is instead computed by modulating the original offsets with an exponential function:

$$\mathcal{X} = r \ \text{sign}(x_i) \ \text{abs}(x_i^m)/r^m \quad \text{for } i = 0 \dots n-1 \quad \text{and} \quad x_i \in \mathcal{X}_u,$$

where exponent m controls the shape of the curve ($m = 2$ gives a reasonable sample redistribution). Assuming n is odd, samples are always guaranteed to exist at 0.0 and at $\pm r$. For the example above, the new offsets will become $\{-2.0, -1.125, -0.5, -0.125, 0.0, 0.125, 0.5, 1.125, 2.0\}$, illustrating that the sample density is increased around at the center.

Computing the set of kernel weights $\mathcal{W}$ requires us to first compute the modified profile $p[x] = \boldsymbol{R}(r)$, where $r = x \in \mathcal{X}$ are the sample offsets. The profile returns the per-channel diffuse reflectance at each of the sample locations. As previously mentioned, it is discretized using the red channel of the six-Gaussian sum for the three-layered skin model, and parameterized by the strength and falloff parameters. This is a direct implementation of the function

$$p[x] = \boldsymbol{R}(r) = \sum_{i=0}^{k} w_i G(v_i, r),$$

where $r = x$, the number of Gaussians is $k = 6$, and the individual Gaussian weights w_i and variances v_i are defined by the following sets (again see variances and weights of the red channel in Table B.1):

$$w_i \in \{0.233, 0.100, 0.118, 0.113, 0.358, 0.078\}$$
$$v_i \in \{0.0484, 0.187, 0.567, 1.99, 7.41\}.$$

The first Gaussian can be considered to represent directly scattered light, and can thus be omitted due to its narrow variance, reducing the number of Gaussians to $k - 1$. The actual Gaussians are computed according to the following equation

$$\boldsymbol{G}(v, r) = \frac{1}{2\pi v} \exp\left(-\frac{(r/(\boldsymbol{f} + \epsilon))^2}{2v}\right).$$

This is similar to the Gaussian function described by d'Eon et al., but now includes a division of radius r by falloff $\boldsymbol{f}$, allowing the shape of the Gaussian to be changed. Higher falloff values widen the Gaussian and smaller values narrow it (on a per-channel basis). Falloff values can range from 0 to 1 for each color channel and are summed with $\epsilon = 0.001$ to prevent division by zero. Note that because falloff is an RGB vector, function $\boldsymbol{G}$ also returns an RGB value, and so does $\boldsymbol{R}(r)$.

The set of kernel weights $\mathcal{W}$ is computed by multiplying each sample weight by the area a_i the sample spans, thus correctly spreading the profile over the total sample range (the total area equals $2r$). Each individual sample area a_i is computed as

$$a_i = \begin{cases} 0.5 \cdot \text{abs}(x_i - x_{i+1}) & \text{if } i = 0 \\ 0.5 \cdot \text{abs}(x_i - x_{i-1}) & \text{if } i = n-1 \\ 0.5 \cdot (\text{abs}(x_i - x_{i-1}) + \text{abs}(x_i - x_{i+1})) & \text{otherwise} \end{cases},$$

where x_i are the sample offsets. The set of kernel weights can now be computed as follows

$$\mathcal{W} = \frac{p[x_i]\ a_i}{\sum_{i=0}^{n}\ p[x_i]\ a_i} \qquad \text{for } i = 0 \ldots n-1.$$

Note that each weight is normalized so that the resulting color from the kernel averages to the color of incident light (which is considered to be white). This is an implementation of the post-scatter texturing approach mentioned in [dL07].

The final step to computing the kernel weights is to modulate them with strength $\boldsymbol{t}$ using the following formula:

$$\boldsymbol{w}_i = \begin{cases} \boldsymbol{w}_i \otimes \boldsymbol{t} + (\boldsymbol{u} - \boldsymbol{t}) & \text{if } i = n/2 \text{ (the center sample)} \\ \boldsymbol{w}_i \otimes \boldsymbol{t} & \text{otherwise} \end{cases},$$

where $\boldsymbol{w}_i$ is a kernel weight expressed as an RGB value, $\boldsymbol{u} = (1, 1, 1)$ is the RGB representation of the color white, and operation $\otimes$ denotes component-wise multiplication between color vectors. Note that this function allows us to increase or decrease the contribution from the center sample at $\boldsymbol{w}_{n/2}$ without exceeding the average response from the diffusion profile; the sum of the kernel weights remains 1.0. It can be seen that for $\boldsymbol{t} = 0$ only the center sample is used to determine the color from the diffusion profile, and that the original normalized kernel weights are used when $\boldsymbol{t} = 1$.

With the kernel offsets and kernel weights computed, the actual filter can now be applied to the screen-space image using the separable subsurface scattering pixel shader; see Listing B.1. Note that the final size of the kernel (in world-space units) is relative to the projected surface area of a pixel, which depends on its depth and surface orientation. On lines 7 and 8 the scale of the kernel due to the depth of the current pixel is determined. This value can be seen as the size of the pixel and is found by dividing the distance to the image plane by the depth value d stored in the depth buffer:

$$scale = \frac{\cot(\theta_{fovy}/2)}{d},$$

where θ_{fovy} is the vertical field of view of the camera used to view the scene. Computing the cotangent of half of this angle yields the distance from the camera's origin to the image plane (see Figure B.1). Note that for a pixel located on the image plane the *scale* value will be 1.0, which decreases with distance. Next, the `step` taken during sampling is computed on line 11. This is a product of the *scale*, global scattering width w seen earlier, and a two-dimensional direction vector that points in either the horizontal $(1, 0)$ or vertical $(0, 1)$ direction depending on whether this is the first or second pass of the filter. It is divided by the half-range r of the sample offsets to bring it back to a range of $[0, 1]$.

Computing the actual diffusion convolution involves accumulating the contribution from each sample. For the center sample we simply need to calculate the product of the skin color fetched from the color map and the central kernel weight. Color contributions from the other samples are determined by sampling nearby texels in the color map, effectively bleeding the color from neighboring locations into the target pixel handled by the shader. Note that linear sampling is used to improve the quality, and that increasing the number of samples will lead to more accurate results.

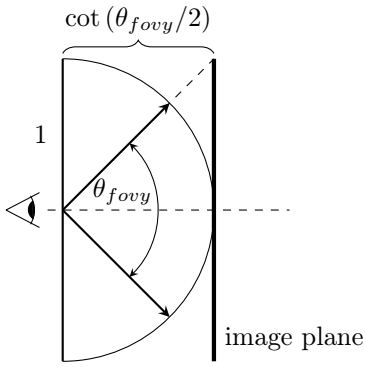


Figure B.1: Computing distance to the image plane using the vertical field of view.

```
1  float4 ssssps(float2 texcoord, float4[] kernel, float width, int nsamples, int range,
       float fovy, float2 direction, Texture2D colormap, Texture2D depthmap) {
2      // fetch color and depth
3      float4 color = colormap.Sample(point, texcoord);
4      float depth = depthmap(point, texcoord).r;
5
6      // compute kernel scale
7      float projdist = 1.0 / tan(0.5 * radians(fovy));
8      float scale = projdist / depth;
9
10     // compute final fetch step
11     float2 step = scale * width * direction / range;
12
13     // accumulate contribution from each sample
14     float4 result = float4(0, 0, 0, 1);
15     for (int i = 0; i < nsamples; i++) {
16         if (i == nsamples/2) {
17             result.rgb += color.rgb * kernel[nsamples/2].rgb;
18         }
19         // fetch color and depth for sample
20         float2 offset = texcoord + kernel[i].a * step;
21         float4 tap = colormap.Sample(linear, offset);
22
23         // lerp back to original color if depth difference is too large
24         float d = depthmap.Sample(linear, offset).r;
25         float s = saturate(300.0 * projdist * width * abs(depth - d));
26         tap = lerp(tap, color.rgb, s);
27
28         result.rgb += kernel[i].rgb * tap.rgb; // accumulate
29     }
30     return result;
31 }
```

Snippet B.1: Separable subsurface scattering pixel shader.

Large depth variations between neighboring samples and the center of the convolution are smoothed out by linearly interpolating the contribution of the sample with the original pixel color if the disparity is larger than the diffusion profile. This will reduce unwanted color bleed between locations that are close in screen-space but far in object space, forcing the convolution to follow the surface of the 3D model more precisely. Note that the correction value of 300.0 on line 25 is an empirically determined quantity.

As previously mentioned, the skin renderer applies the separable subsurface scattering pixel shader as a post-process filter over the diffuse image. A first pass computes the convolution in the horizontal direction, and a second pass performs the vertical convolution over this result. Since subsurface scattering is only applied to diffuse component, the specular component computed earlier was separated and added back in the final rendering pass in order to avoid blurring it.

B.2.2 Transmittance

As discussed in the introduction, subsurface scattering can consists of both reflectance and transmittance. Transmittance occurs when incoming light is able to travel far enough into the material that a fraction of it will exit at the other side. The exact properties of a material will determine the distance and diffusion of light traveling through a material. Because skin has a relatively high absorption factor and an average scattering factor it is neither transparent nor opaque, but translucent. Some light is allowed to pass through thin regions of skin like the ears and nose, but only in a highly diffused form.

Various methods have been developed to incorporate translucency effects into skin rendering methods. In the realm of offline rendering, Donner and Jensen derived equations to compute transmittance profiles along with the reflectance profiles seen earlier [DJ05]. For real-time rendering, one way to account for transmittance is by using modified shadow maps. Our skin renderer implements the screen-space translucency technique by Jimenez et al. [JWSG10], which is based on the texture-space variant presented by d'Eon et al. [dL07, dLE07].

B.2.2.1 Texture-space translucent shadow maps

Using shadow maps to compute translucency effects was first introduced by Dachsbacher and Stamminger [DS03]. So-called *translucent shadow maps* (TSM) not only store the distance from a light to the closest surface point, but also include the incident irradiance color and the surface normal at that point. With this information the translucency integral from Jensen et al. [JMLH01] can be computed as a filter on the irradiance, where the filter weights depend on the corresponding depth and normal values at each pixel.

In their texture-space diffusion method, d'Eon et al. use modified translucent shadow maps [dL07, dLE07] to simulate transmittance. Instead of storing depth, irradiance, and surface normal, they store the (u,v) coordinates and depth to the light-facing surface. Figure B.2 shows the geometrical situation for computing the translucency. While rendering a shadowed location x_{out}, the shadow maps are accessed to obtain the distance m and the (u,v) coordinates of point x_{in} on the light-facing surface.

Radiant exitance $M(x, y)$ at surface point x_{out} is the convolution of irradiance $E(x_{in})$ around point x_{in} by the reflectance profile $R(r)$ through the thickness of the object. However, evaluating $M(x, y)$ at nearby point x'_{out} instead allows for radiant exitance to be computed much more efficiently without introducing any significant errors:

$$M(x'_{out}) = E(x_{in}) * R(\sqrt{r^2 + d^2}).$$

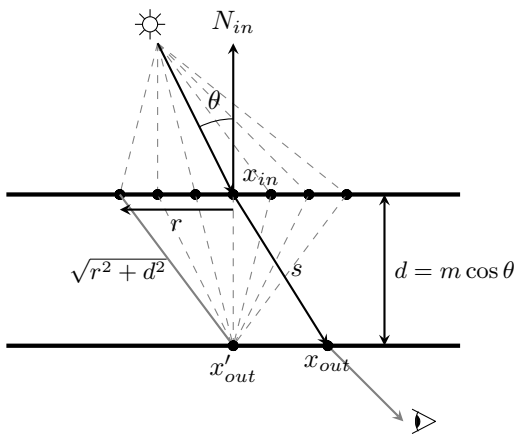


Figure B.2: Computing translucency through a thin region.

Convolution kernel $R(r)$ is computed using the Gaussian-sum approximation like before

$$R(\sqrt{r^2+d^2}) = \sum_{i=1}^{k} w_i \; G(v_i, \sqrt{r^2+d^2}) = \sum_{i=1}^{k} w_i \; \exp(d^2/v_i) \; G(v_i, r),$$

where $d = m\cos\theta$ is the local thickness of the object (again see Figure B.2).

This last form is convenient, because the light-facing points have already been convolved by $G(v_i, r)$ during the reflectance step. This allows us to skip the expensive calculation of the diffusion profile by directly accessing the convolved irradiance textures using the (u,v) coordinates stored in the TSM. Thus, the diffuse light exiting at point x'_{out} is a sum of k texture lookups, each of which is weighted by w_i and an exponential term based on v_i and d. Note that by expressing $R(r)$ as a Gaussian sum the convolution textures can reused for both local scattering (subsurface reflectance) at x_{in} and global scattering (transmittance) at x_{out}. Their weighted sum will accurately approximate the total convolution by the original diffusion profile.

B.2.2.2 Screen-space translucent shadow maps

Jimenez et al. [JWSG10] describe a solution to compute translucency for screen-space diffusion that works similarly to the texture-space approach. The main consequence of working in screen space is having less information about the surface points that are on the rear side of the object (from the perspective of the camera). As before, the translucency problem amounts to solving the following equation

$$M(x,y) = \iint E(x,y) \; R(\sqrt{r^2+d^2}) \; dx \; dy.$$

Distance d traveled through the object can still be obtained from shadow maps. However, the incident irradiance $E(x_{in})$ at point x_{in} on the light-facing surface is unavailable in screen space. Therefore, a number of assumptions are introduced to approximate the irradiance.

First, the normal at x_{in} is replaced by the reversed normal at the point being shaded: $N_{in} = -N_{out}$. Second, the albedo α_{out} at the front (from the camera view) is used to approximate the albedo at the back. This is possible because the surface reflectivity generally does not drastically vary for skin. Furthermore, the irradiance on the light-facing surface is assumed to be approximately locally constant so that all sample points around x_{in} will have the same value as point x_{in} itself ($E(x_{in}) = E$), and:

$$E = \alpha_{out} \; \max(0, N_{out} \cdot L),$$

where L is the light vector. By this assumption, we have:

$$M(x,y) = E \int_0^{\infty} 2\pi r \; R(\sqrt{r^2+d^2}) \; dr.$$

By using d'Eon's Gaussian-sum approximation and considering that the Gaussians have a unit total diffuse response, $M(x,y)$ is reduced to (see [JWSG10] for the full derivation):

$$M(x,y) = E \sum_{i=1}^{k} w_i \exp(-d^2/v_i),$$

which depends only on E and d. To reduce computational complexity d is approximated by s (again see Figure B.2), which can be directly obtained by transforming point x_{out} to light-space and subtracting it from the stored depth value at that point. This gives us a physically based function $T(s)$ that relates the attenuation of the light intensity with the distance traveled inside the object

$$T(s) = \sum_{i=1}^{k} w_i \exp(-s^2/v_i).$$

This function can be precomputed and stored to be used as an attenuation texture, or directly evaluated in the shader. In the skin renderer the latter option is chosen because accessing texture memory is usually slower than computing the sum directly, and will also save a little bit of memory.

Using the reversed normal $-N_{out}$ can cause non-smooth transitions between areas illuminated by reflectance to areas exclusively illuminated by transmittance. Wrap lighting (see [Gre04]) is used to start the transmittance gradients approximately 17 degrees before a regular dot product, resolving any abrupt illumination artifacts:

$$E = \alpha_{out}\, max(0.0, 0.3 + (-N_{out} \cdot L)).$$

The pixel shader for translucency is shown in Listing B.2. Note that the transmittance is computed as part of the standard shading process, and evaluated for each light after the diffuse and specular lighting computations.

```
1  float3 transmittance(float3 worldpos, float3 normal, float3 lightdir, Texture2D
       shadowmap, matrix viewproj, float1 farplane, float3 spotlight, float3 albedo) {
2      float4 pos = float4(worldpos - 0.005 * normal, 1.0);
3      float4 lightpos = mul(pos, viewproj);
4      float d1 = shadowmap.Sample(linear, lightpos.xy / lightpos.w).r * farplane;
5      float d2 = lightpos.z;
6      float s = (8.25 * (1.0 - Translucency) / ScatterWidth) * abs(d1 - d2);
7
8      float ss = -s*s;
9      float3 Ts = float3(0, 0, 0);
10     Ts += float3(0.233, 0.455, 0.649) * exp(ss / 0.0064);
11     Ts += float3(0.100, 0.336, 0.344) * exp(ss / 0.0484);
12     Ts += float3(0.118, 0.198, 0.000) * exp(ss / 0.1870);
13     Ts += float3(0.113, 0.007, 0.007) * exp(ss / 0.5670);
14     Ts += float3(0.358, 0.004, 0.000) * exp(ss / 1.9900);
15     Ts += float3(0.078, 0.000, 0.000) * exp(ss / 7.4100);
16
17     float3 E = saturate(0.3 + dot(-normal, lightdir));
18     return E * Ts * spotlight * albedo;
19 }
```

Snippet B.2: Computing translucency as part of the main pixel shader.

The first seven lines obtain the depth value at x_{in} (`d1`) and compute it for x_{out} (`d2`), whose difference equals distance s. Using shadow maps for depth approximations leads to inaccuracies at the edges of the object when background pixels are projected onto the object. To ensure that all sample points fall onto the object while querying the depth from the shadow maps (line 3), the sample position is shrunk in the direction of the surface normal by a small amount (line 1). Finding the correct sample position in the shadow map involves performing the perspective transform manually (line 3).

Distance s is scaled by two parameters that configure the intensity of the transmittance. The first is the width of the scattering filter in world-space units. This quantity was previously used for the subsurface reflectance filter. The translucency factor is a value from 0 to 1 that directly controls the intensity of the transmittance effect. An empirically chosen scaling value of 8.25 controls the distance at which transmittance becomes noticeable.

Lines 8–15 compute the color corresponding to distance s using the Gaussian-sum approximation of Table B.1. Next, irradiance E is approximated as described above, and the total transmittance contribution is found by multiplying $M(x, y) = ET(s)$ with the light source intensity at the shaded point, and this is done for each light source for a given pixel.

The skin renderer computes the final pixel color as a combination of the diffuse lighting – modified by the subsurface scattering reflectance filter and the transmittance value – and the separate specular lighting. Both components are modulated by normal mapping and shadows.

Appendix C. Class diagram

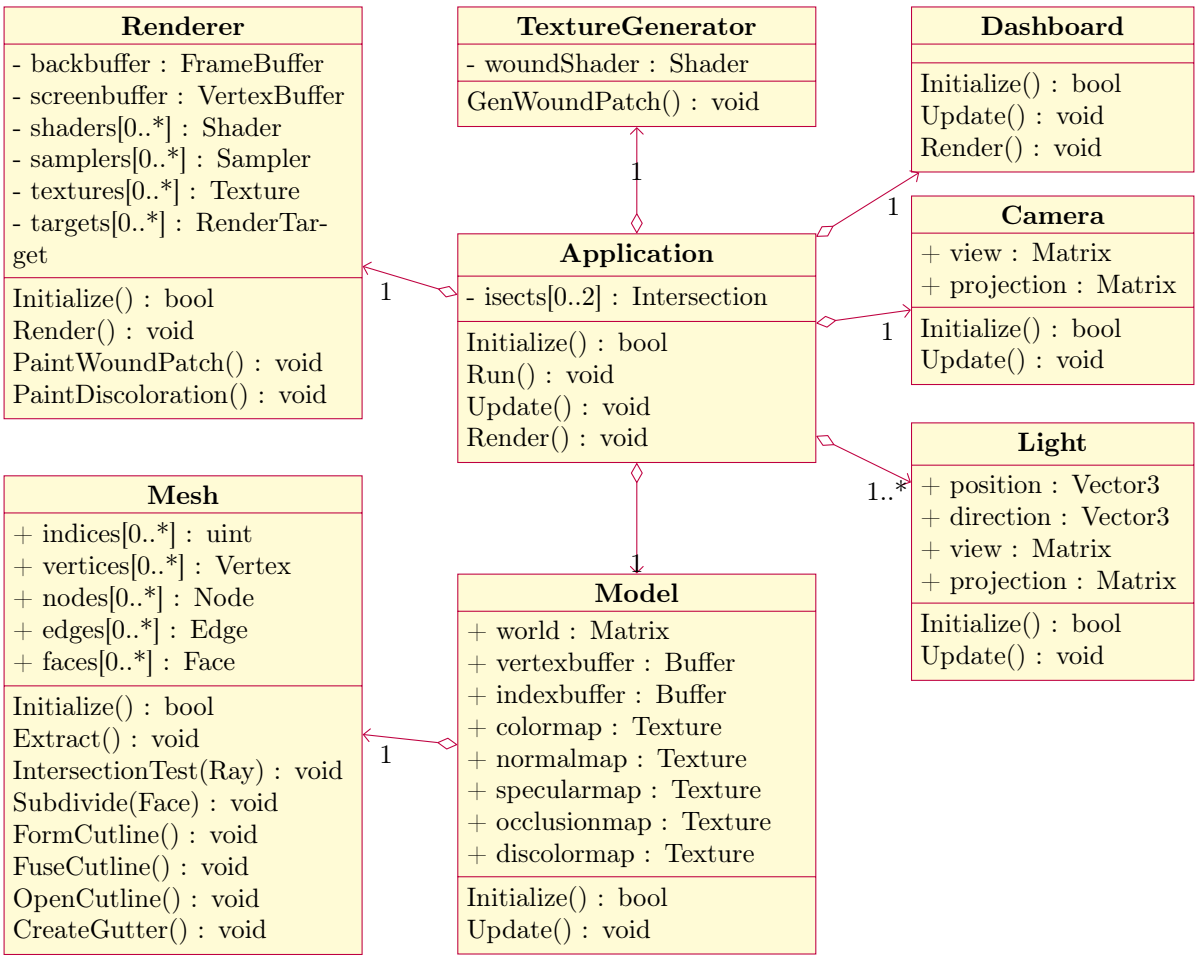


Figure C.1: Class diagram of our system. Only the most important classes and member variables and functions are shown.