Universiteit Utrecht

Real-time simulation of abrasion wound healing

Master Thesis

ICA-3998878

KRIEN LINNENBANK

Supervisors:
dr. ir. J. Egges
dr. ir. A.F. van der Stappen

Januari 2015
Version 3.3

Abstract

This thesis proposes a method for simulating the healing process of abrasion wounds, a subject which has received little attention until now. Systems that simulate wounds mostly only simulate the wound at one fixed point in its healing process. Each type of wound has its own unique look, which changes throughout the healing process. In computer graphics, textures are used to store all sorts of visual attributes, such as color- or normal information. We propose a framework for procedurally generating textures, aimed at wound healing simulation. The framework contains a flexible timeline that can automatically generate textures, each representing a phase in the healing process of a wound. We provide a set of techniques which can be used for the texture creation. We present a prototype simulation system based on this framework that can automatically simulate the healing process of an abrasion wound.

Contents

Chapter 1

Introduction

This thesis focuses on the simulation of the healing process of abrasion wounds, a subject which has received little attention until now. There have been studies of simulations that include wounds, for example a dynamic wound rendering system as seen in the game Left 4 Dead 2 (see figure 1.1) or a system that can be used to teach surgeon students how to remove debris from a pre-generated laceration wound [1]. In this thesis we introduce several techniques that each can be used to create a visual simulation of different parts of the healing process of a wound. These techniques are applied to create a dynamic wound simulation system. Wound healing is a complex process. The visual look of the wound depends on many (environmental) variables. Because every type of wound can have a completely different look and healing process, the choice has been made to only focus on shallow abrasions that do not bleed (i.e. no blood flowing out of the wound). Possible applications for wound simulation include real-time simulations, for example medical simulations for educational purposes or videogames, and non-real-time simulations, for example pre-rendered films for either entertainment or educational purposes.

The objective of this thesis consists of two parts: the first part is to propose a framework that can be used to simulate the healing process of a wound. The second part is to use this framework to create a real-time simulation of the healing process of an abrasion wound.

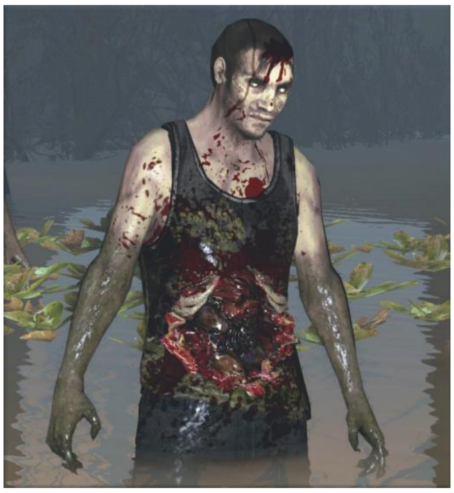

Figure 1.1 – Dynamic wound rendering in Left 4 Dead 2

1.1 Thesis overview

Chapter 2 discusses the relevant medical background that is necessary to understand the process of wound healing and gives an overview of relevant related work in the field of games and visual simulation. Chapter 3 proposes a framework for procedurally generating textures, aimed at wound simulation. Chapter 4 describes a system that uses the framework proposed in chapter 3 to create a visual simulation of the healing process of an abrasion. Chapter 6 contains the conclusion and future work section.

Chapter 2

Related work

This chapter will provide an overview of existing methods for simulating wounds. First, section 2.1 describes the physiological wound healing process. Section 2.2 describes existing methods, which are designed to simulate wounds. Section 2.3 discusses these methods and further motivates the method described in this thesis.

2.1 Physiological wound healing

Physiology is the study of the functions of living organisms and their parts. So, if one would fall and scrape its knee to cause an open wound, the physiological wound healing process kicks in. A wound is a type of injury where living tissue is damaged by a cut, blow, or other impact, typically one in which the skin is cut or broken. Wounds can be placed in two categories: open wound and closed wounds. Incisions, lacerations, abrasions, avulsions, puncture wounds and penetration wounds are classified as open wounds. Hematomas and crush injury are considered closed wounds. Incisions and lacerations are often confused as both wounds can have a somewhat similar appearance. Incisions are caused by a sharp-edged object cutting through the skin and lacerations are tear-like wounds caused by blunt trauma. Abrasions are wounds where the topmost layer of the skin is scraped off. Avulsions are a type of amputation where a part of the body forcibly pulled off. Puncture wounds and penetration wounds are also closely related. Puncture wounds are caused by an object puncturing the skin. Penetration wounds are cause by an object entering and coming out from the skin. Hematomas are caused by a damaged blood vessel. That means blood starts to collect near the damaged part of the blood vessel. Crush injuries are caused by a great amount of force applied to a part of the body over a long period of time. Most wounds involve the skin, which include all open wounds and most closed wounds. This section describes the healing process for wounds for open wounds, which also include abrasions.

The healing process of human tissue consists of multiple phases. These phases all lie on a non-linear timeline and have a variable beginning and end [2]. This timeline with the phases is depicted in figure 2.1. The gradient fades in the image indicate the variable lengths of the phases.

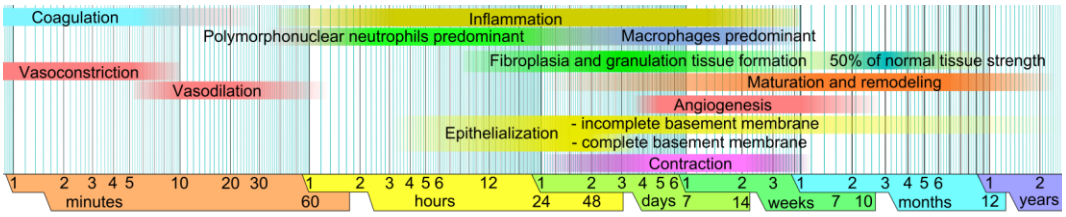

Figure 2.1 – Wound healing timeline containing all phases with variable beginning and end

Coagulation (clotting) of platelets in the wound site causes bleeding to stop and makes sure no more blood can flow out of a damaged blood vessel. Coagulation happens parallel to vasoconstriction, which is the narrowing of blood vessels (capillaries) around the wound site. This makes the skin slightly paler, because less blood reaches the surface of the skin. Vasoconstriction lasts between 5 to 10 minutes. After this the blood vessels dilate during the vasodilation. This causes the skin around the wound site to have a slightly redder color, which lasts about 20 minutes. The combination of coagulation, vasoconstriction and vasodilation, which all combined cause the wound to stop bleeding is called hemostasis. Within an hour after injury, white blood cells (polymorphonulcear neutrophils) arrive at the wound site, which clean up bacteria and damaged skin tissue. They are mostly present in the first and second day after injury. Two days after injury, macrophages replace the white blood cells as the predominant cells in the wound. These Macrophages cleanup white blood cells in the wound and help clean up the remaining bacteria and damaged tissue.

After the bleeding has stopped, the proliferative stage begins, which is a set of four phases: angiogenesis, fibroplasia and granulation tissue forming, epithelialization and contraction. The inflammation phase runs parallel to most of the proliferative stage. Angiogenesis is the process of creating new blood vessels from existing blood vessels. Stem cells regenerate the damaged blood vessels underneath and the extracellular matrix: a collection of extracellular molecules secreted by cells that provide structural and biochemical support to the surrounding cells. Fibroblasts enter the wound two to five days after injury near the end of the inflammation phase. Fibroblasts lay down a collagen matrix in the wound site. Collagen is one of the main structural proteins present in connective tissues. Granulation tissue also appears two to five days after injury, near the end of the inflammation phase. It is a rudimentary tissue that mainly consists of new small blood vessels, fibroblasts, inflammatory cells and endothelial cells. Granulation tissue forming partly overlaps with contraction, because granulation tissue keeps growing until the wound is covered. During the epithelialization, epithelial cells move across the new granulation tissue to form a barrier between the wound and the environment. This will eventually form the scab on the wound. Contraction begins about a week after injury and is a key phase in wound healing. After fibroblasts have formed into myofibroblasts, contraction begins. The myofibroblasts move from the outer edges of the wound towards the center of the wound site and by doing so, close the wound. Contraction can last for several weeks. A large wound can become 40 to 80% smaller after contraction.

Maturation and remodeling is the last phase of wound healing. Maturation and remodeling starts after 3 days to 3 weeks after injury. During maturation, disorganized collagen fibers are rearranged and aligned along tension lines, after which the wound is healed.

2.2 Virtual wounds

Some of the wound healing phases described in the previous section have a direct visual appearance, e.g. hemostasis or contraction, while other phases are not visible at all, e.g. the white blood cells or the macrophages. Several methods have been developed which include a visual simulation of a wound, although none of these methods include the simulation of multiple wound healing phases. This section describes methods created by other researchers which can be used to create a visual simulation of a wound, as well as examples of wound simulation methods which are used in practice.

2.2.1 Left 4 Dead 2 wound rendering

Left 4 Dead 2 is a game developed and published by Valve Corporation. The game features zombies which the player must shoot to survive the game. The game features a dynamic wound rendering system, which is explained by Alex Vlachos from Valve Corporation in a presentation at the Game Developers Conference in San Francisco, CA [3].

One of the goals of the wound system was to be able to dynamically and accurately determine the location and size of the wound on a zombie character after the player has shot it. They have achieved this by using pose-space ellipsoids to cull pixels of the zombie characters. The created hole in the character is filled with a pre-made wound model, as seen in figure 2.2.

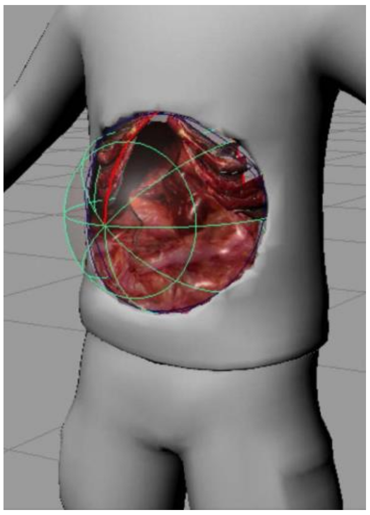
Figure 2.2 – Pose-space ellipsoid used to create a wound hole

This system works very well in the fast-paced shooter that is Left 4 Dead 2. It can simulate up to 54 unique wounds per zombie character, has a very low memory cost (13% of that of the system in Left 4 Dead 1) and very few shader instructions (15 for the vertex shader and 7 for the pixel shader). However, this system requires an artist to make several wound models and textures to fit in the ellipsoid holes. Also, the wounds do not heal. They are static wounds that represent the wound just after injury.

2.2.2 A Simulation-Based Training System for Surgical Wound Debridement

Seevinck et al. have developed a simulation-based training system for surgical wound debridement [1]. It is built to help train medical personnel in cleaning a wound that is filled with debris. The cleaning of the wound is done on a mesh of a human leg with a shallow laceration wound in it. The trainees use haptic input devices to remove debris from the wound in the simulation and clean it with a saline solution, as seen in figure 2.3. The haptic devices give haptic feedback to the user so they get a realistic feel of how the wound feels when using the debridement equipment.

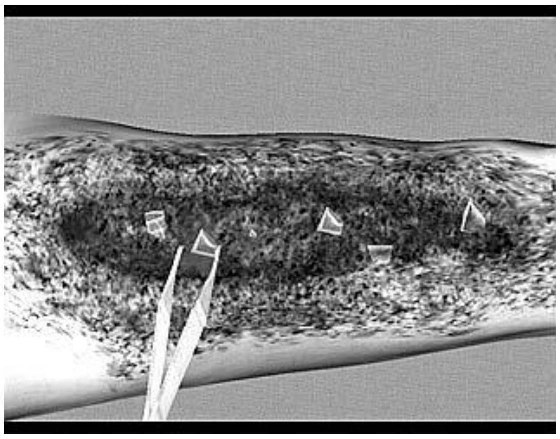
Figure 2.3 – Wound debridement

The leg mesh is a low-poly mesh with approximately 1000 polygons. The wound shape and color is pre-made and thus static. The debris is modeled using triangulated polygonal shapes and textures that are laid on top of wound on the leg mesh. The majority of the debris, such as dust and dirt, are rendered as a texture laid on top of the wound, while larger objects are rendered as polygonal meshes. A bleeding system is also implemented using particles.

This system is a useful tool to train medical personnel in how to clean wounds. However, in terms of visual wound simulation, the system is somewhat lacking. It can only simulate one static wound that always has the same shape and color. Also, like the wound system of Left 4 Dead 2 (as discussed in section 2.2.1), this wound system also does not simulate the healing process of the wound. Only the initial injury is shown.

2.2.3 Multi-layer Structural Wound Synthesis on 3D Face

Lee et al. presented a method that could simulate wounds on a 3D face while taking into account the varying depth of the skin on the face [4]. The flesh on the face of a human is does not have the same thickness overall. For example, the thickness of the flesh near the cheeks (the length from the outside of the skin to the nearest bone) is larger than the thickness of the flesh on top of the head. Using this information along with the fact that the skin consists of three primary layers, the authors developed a method that can approximate the depth of the wound at an arbitrary point on a virtual face mesh. Using this depth information, they offset the mesh geometry of the face at the location of the wound to give the wound depth. For the rendering of the wounds they used a collection of input wound images classified into types of wounds (abrasions, lacerations, burn wounds, etcetera). These images are laid on top of the location of the wound. The combination of the depth in geometry and the color of the input image gives the wound color and texture, as shown in figure 2.4.

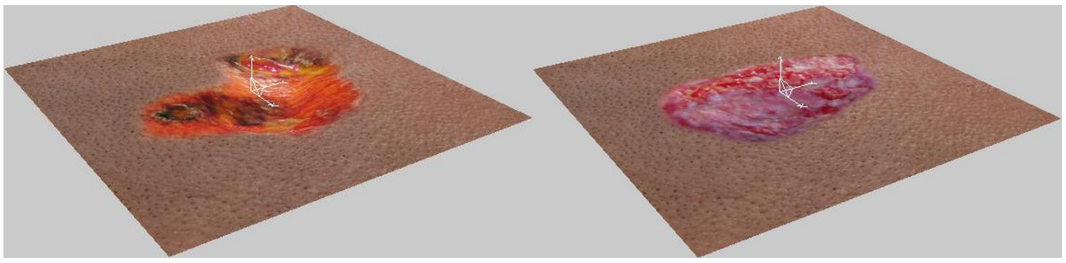

Figure 2.4 – Wounds laid on skin with varying depth

This system is able to render realistic looking wounds both the depth of the wound and the visual colors of the wound. The color of the wound is only achieved by sampling from an input image, however, so the wounds always look the same (apart from the varying depth).

2.2.4 Skin tissue visualization

Almost all wounds involve skin one way or another (see section 2.1). So when simulating a wound, one must also consider modeling the skin that may lie in or around the wound. Subsurface Scattering (SSS), sometimes called Subsurface Light Transport (SSLT) are techniques that can accurately simulate reflection of light in multiple layered materials, such as skin. It is a subject in which a lot of research has been done. It was pioneered by Hanrahan and Krueger in 1993 [5]. They described a Monte Carlo method which could be added to a ray tracer for rendering layered materials appearing in nature, such as skin, leaves, snow or sand. It produces good results, as seen in figure 2.5, but is quite expensive to calculate (over 100 shader instructions per light ray).

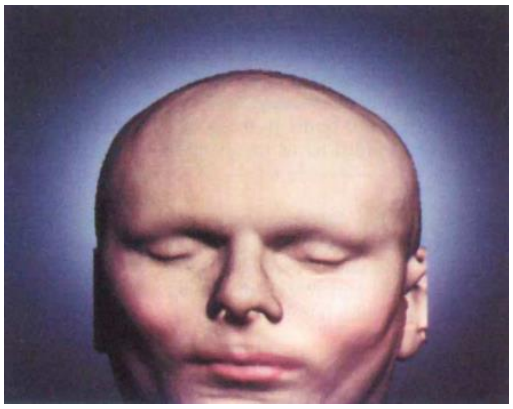

Figure 2.5 – Hanrahan and Krueger Subsurface Scattering

A more recent Subsurface Scattering method developed by d'Eon and Luebke [6] is able to create realistic renderings of human skin, as seen in figure 2.6. They formulate a real-time Subsurface Scattering technique specialized in rendering skin. They combine physically based multilayer skin models with algorithms based on texture-space diffusion and translucent shadow maps to create a level of realism in real-time renderings which were previously only available in offline rendering.

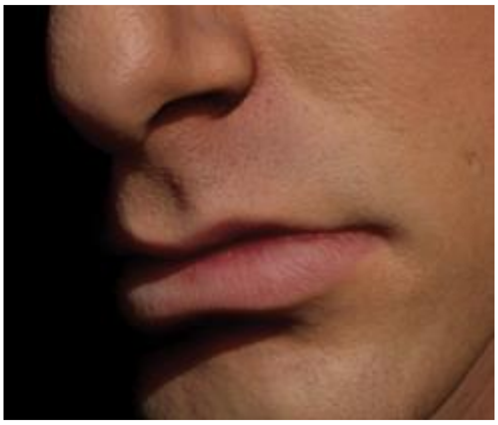

Figure 2.6 – d'Eon and Luebke Subsurface Scattering skin rendering

2.3 Motivation

The previous section has shown several methods for simulating wounds. Each method has its own unique pros and cons. However, one disadvantage that they all share is that they are all static methods in the sense that each method is only capable of simulating a static point in the wound healing process, which is most of the time the moment of injury. Subsurface Scattering is the only method that does not share this limitation, because it is a generic lighting technique. However, Subsurface Scattering techniques are difficult to implement and therefore cost a lot of development time. See future work in section 5.1 for more on this.

What is needed is a method that can generate and simulate the entire healing process of a wound, unlike the methods described in this chapter which can only simulate a fixed position in the wound healing process. This method can be used to create a system that will allow the user to step through the healing process to an arbitrary point on the healing timeline. Chapter 3 proposes a framework that can be used to simulate the healing process of a wound. Chapter 4 describes an implementation of this framework that simulates the healing process of an abrasion.

Chapter 3

Procedural wound texture generation

Each type of wound has its own unique look, which changes throughout the healing process. In computer graphics, textures are used to store all sorts of visual attributes, such as color or normal information. In this chapter, we propose a framework for procedurally generating textures, aimed at wound healing simulation. Section 3.1 describes the used definitions for the framework. Section 3.2 gives an overview of how the framework works, section 3.3 describes commonly used techniques in the framework. A summary and discussion is provided in section 3.4.

3.1 Definitions

It's important to have an unambiguous base on which all used techniques are based. This way it is always clear how the techniques are formed and how they connect to each other. This section gives definitions to textures and meshes which are used in the framework.

3.1.1 Textures

A texture is a set τ of color values ρ with coordinates (x, y) on $\mathbb{Z}^2$ where $0 \leq x \leq n$ and $0 \leq y \leq m$.

$$\tau_\rho(x, y) = \{\rho : \rho \in \tau\}$$

$$\tau_{uv}(u, v) = \tau_\rho\left(\frac{x}{n}, \frac{y}{m}\right) = \rho_{uv} = \begin{pmatrix} r \\ g \\ b \\ a \end{pmatrix}$$

where n and m are the dimensions of a $n \times m$ lattice on $\mathbb{Z}^2$, u and v are the normalized coordinates of the x- and y-axis, respectively, ρ_{uv} is a color value at location (u, v) and r, g, b and a represent the colors red, green, blue and alpha, respectively, ranging $[0,1]$.

3.1.2 Meshes

Meshes are a set of vertices v and edges e. The vertices v each have coordinates (x, y, z) on $\mathbb{R}^3$, The edges e are a tuple (v_1, v_2) connecting two vertices.

The edges can share vertices, which allows the formation of a 3D triangle grid. The edges of the triangles in this triangle grid are shared between adjacent triangles. Each vertex v is defined as an n-tuple containing various amounts of data. The four most used data types are position data, UV coordinate data, normal vector data and tangent vector data.

3.2 Framework overview

This section gives an overview of the developed framework. It will show how all the major parts of the framework work together.

The framework uses a one-dimensional timeline on which several positions are defined. Each position defines visual representation of the simulated object at that point in time. These positions are called key positions. These key positions output one or more textures (for example, one color texture and a normal map texture). Linear interpolation between the outputted textures of the immediate left and right key positions is used to calculate the visuals of an arbitrary point on the timeline. These interpolated textures can then be placed on a loaded 3D mesh after which lighting techniques can be applied to give the final desired look.

3.2.1 Structure

All the key positions on the timeline generate their output on the basis of a binary input shape and a hull. The binary input shape is a texture where a white pixel indicates a position on the shape and a black pixel indicates a position that is not on the shape. The hull runs around the edge of the input shape to further emphasize the borders of the shape. More on this input shape and its hull can be found in the next chapter, section 4.2.

As mentioned before, the textures generated by the key positions are used by the timeline to calculate the correct color values for the textures on an arbitrary point on the timeline. The minimum amount of key positions on the timeline is two key positions: one on the far left and one on the far right. This means that no matter where you are on the timeline, there are always two key positions to interpolate between, as seen in figure 3.1. Note that these two key positions don't necessarily need to output different textures, although it would be pointless if they would.

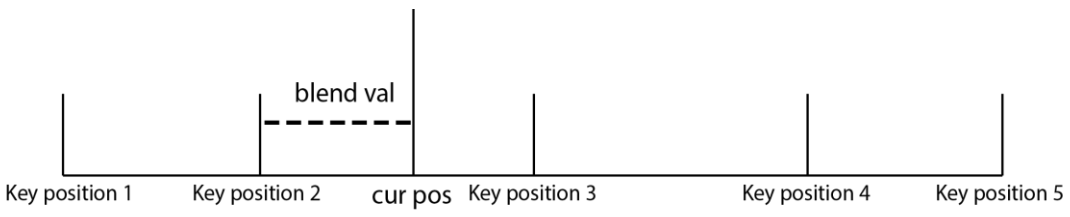

Figure 3.1 – Key position blending in the timeline

The interpolated textures that the timeline creates can be seen as the final output of the whole simulation. What happens next with these textures depends on the implementation. The implementation used for this thesis uses it in a static scene where the user can scroll through the timeline, but one can also imagine other types of scenes with, for example, animation and/or no user interaction.

A full schematic overview of the whole simulation structure is seen in figure 3.2.

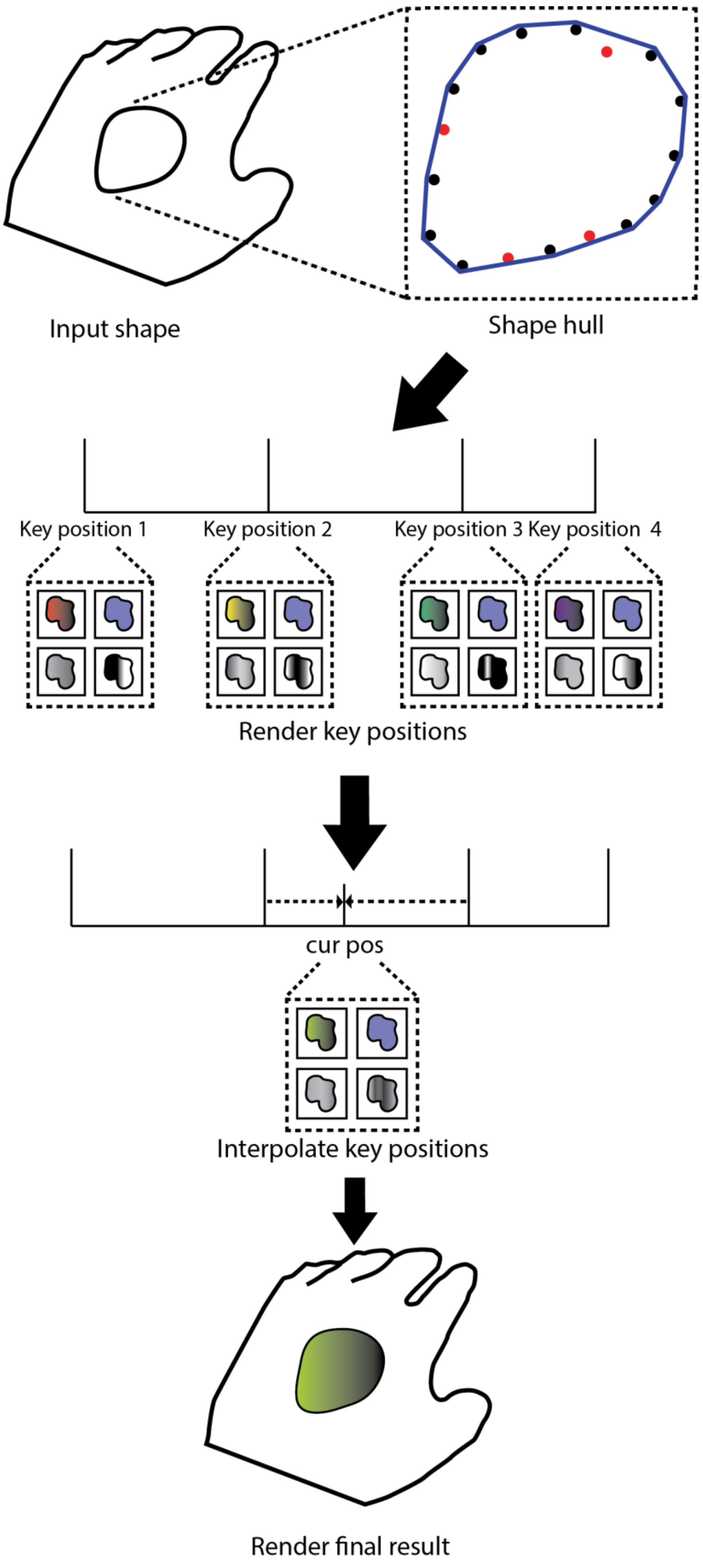

Figure 3.2 – Schematic overview of the simulation

3.3 Key position techniques

Each key position has its own unique output. This section describes common techniques which can be used to give key positions their unique output. A specialized form of convolution erosion is described that can create rough edges on binary textures. The noise generation section describes the calculation of Value noise. Lastly, gradient edge generation uses the hull of the input shape to create a gradient edge around this input shape.

3.3.1 Input shape erosion

The shape erosion technique is used to add small details to the edge of the shape texture. It helps increase the feel of something that scraped over the skin, causing the abrasion. What this technique does is it uses an $N \times 1$ convolution kernel to erode the edge of the shape texture, where N is a varying kernel size. This creates rough edges, as seen in figure 3.3. For more information on convolution kernels, see [7]. This convolution kernel is applied to every pixel in the shape texture.

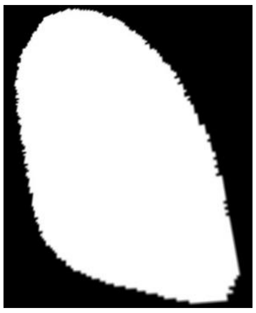

Figure 3.3 – Eroded edges

The convolution kernel in this technique has a fixed height and a variable width. This means that the height of the erosion kernel is the same for every processed pixel, but the width of the kernel is variable. The width of the kernel is pre-calculated for each pixel before the erosion is applied after which the erosion itself is applied. A two-dimensional array with the same dimensions as the input shape texture is filled with random positive integers, excluding 0 (as a kernel with a width of 0 would make no sense). These numbers indicate the size of the erosion kernel at the current pixel. Only kernels with an odd width are used, so there is always a center element in the kernel. To make sure each generated kernel width is an odd number, equation 3.1 is applied to every generated number:

$$kernelWidth(m) = m + (m \bmod 2) + 1 \tag{3.1}$$

Where m is the generated random number.

The complete erosion method is listed in Algorithm 3.1. Do note that color sampling outside the border of the texture is not shown in this pseudo-code. Pixels that lie outside the texture borders are processed as they have the highest possible color value, which means they are ignored.

Algorithm 3.1 Shape texture erosion (*T*, *ranNums*)
Input. Shape texture *T* to draw on and a 2D array of odd integers *ranNums*
Output. A texture containing eroded shape.

```
1.  For all pixels $\rho$ in T do
2.      outputColor ← Initialize to highest color value possible
3.      kernelWidth ← get width from ranNums using absolute location of $\rho$ in T
4.      edgeWidth ← kernelWidth / 2
5.      For –edgeWidth upto edgeWidth do
6.          outputColor ← min($\rho$, outputcolor)

7.  return outputColor
```

3.3.2 Noise generation

Procedural noise generation can be used to give some variation to the output of a technique to give it a more natural look. Coherent noise is a type of smooth pseudorandom noise, which is generated by a coherent-noise function. Coherent noise will always return the same output value for the same input value. This section describes Value noise, which is a type of coherent noise. The Value noise described in this section outputs 2D textures containing the final result.

Value noise starts with the creation of an array of random values. The noise function then calculates the interpolated noise value based on the values of the surrounding points. This noise function is run several times, depending on the input parameters. All the generated noise values are summed up and a weighted average of all the generated noise values is then used to create the final result. Each successive value noise function that is added to the final result is called an octave. The weight of each octave is called the amplitude of the octave and each octave had a different amplitude. See section 3.3.2.2 for more on this.

3.3.2.1 Generating octaves

Each octave uses one base noise 2D array to sample from. This texture has the same size as the output texture and is filled with random values ranging from 0 to 1. Bicubic interpolation is used to sample from this base noise array. To calculate the bicubic interpolated value on pixel p, 16 points around pixel *p* are sampled and interpolated (see figure 3.4). Points that fall outside the array, because pixel *p* lies to close to the edge of the array, are wrapped around to the other side of the array to make sure the sample points always sample a noise value. See equation 3.2 and 3.3. The sample points are sampled according to the period of the current octave. The period of an octave is the size of the sampling step and is defined as 2^i, where i is the current octave.

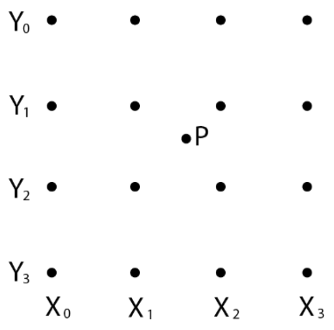

Figure 3.4 – Sampling points around pixel *p*

To be able to sample the correct pixels, the four positions of the rows and the four positions of the columns seen in figure 3.4 need to be calculated. The four row positions, labeled y_0, y_1, y_2 and y_3 are calculated as follows:

$$\begin{gathered} y_1 = \frac{p_y}{period} * period \\ y_0 = \begin{cases} height - period + y_1, & y_1 < period \\ y_1 - period, & y_1 \geq period \end{cases} \\ y_2 = (y_1 + period)\ \%\ height \\ y_3 = (y_2 + period)\ \%\ height \end{gathered} \tag{3.2}$$

Where p_y is the row of the current pixel and $height$ is the height of the entire texture.

The four column positions, labeled x_0, x_1, x_2 and x_3 are calculated in a similar way:

$$\begin{gathered} x_1 = \frac{p_x}{period} * period \\ x_0 = \begin{cases} width - period + x_1, & x_1 < period \\ x_1 - period, & x_1 \geq period \end{cases} \\ x_2 = (x_1 + period)\ \%\ width \\ x_3 = (x_2 + period)\ \%\ width \end{gathered} \tag{3.3}$$

Where p_x is the column of the current pixel and $width$ is the width of the entire texture.

Again, notice that for both row and column positions the sampling points wrap around to other side of the texture (bottom and right of the texture, respectively) if positions x_1 or y_1 lie too close to the border of the texture.

The last ingredient for bicubic interpolation is to calculate two interpolation blend values: one for the rows and one for the columns. These blend values represent the relative position between two points in the middle on the rows and columns. See figure 3.5.

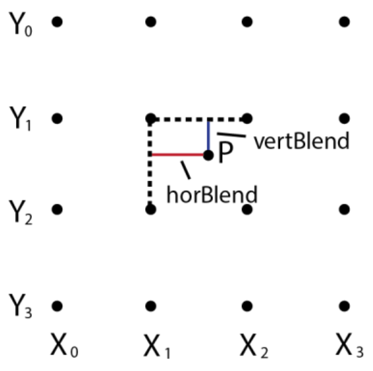

Figure 3.5 – Interpolation blend values

These blend values are calculated by multiplying the difference between two middle points by the frequency of the current octave. The frequency of the octave is defined as $^1/_{period}$.

The bicubic interpolated value for the current pixel can now be calculated, as shown in figure 3.6. First four values are calculated using cubic interpolation with the four points on each row and the horizontal blend value as input. This results in four interpolated values. These four values and the vertical blend value are used to calculate the final interpolated value, again using cubic interpolation. Cubic interpolation itself is shown in equation 3.4.

$$
\begin{gathered}
p = (v_3 - v_2) - (v_0 - v_1) \\
q = (v_0 - v_1) - p \\
r = v_2 - v_0 \\
interpolatedValue = p * blend^3 + q * blend^2 + r * blend + v_1
\end{gathered}
\tag{3.4}
$$

Where v_0, v_1, v_2 and v_3 are the row or column positions and $blend$ is the blend value for the row or column.

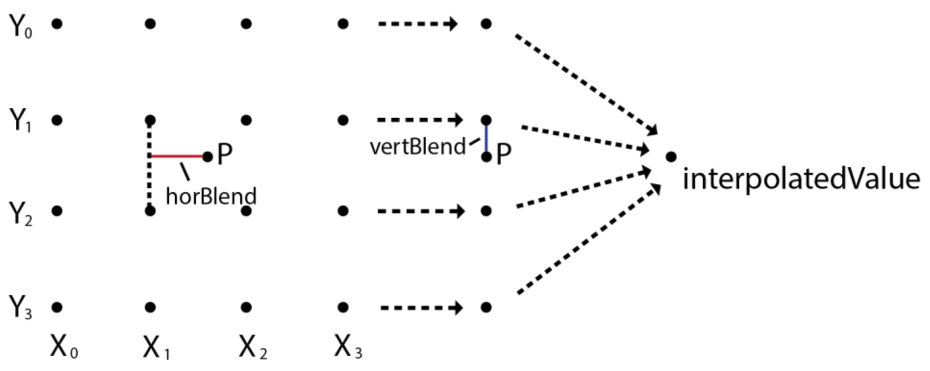

Figure 3.6 – Bicubic interpolation

The final result for the octave is calculated by multiplying the final interpolated value by the octave's amplitude. The amplitude determines the maximum value a noise pixel can have, see equation 3.5.

$$finalResult = interpolatedValue * amplitude \tag{3.5}$$

3.3.2.2 Combining the octaves

All the octaves are created iteratively (see algorithm 3.2). After the octaves have been created, it is time to combine them to create the final noise result. This last step is pretty straightforward. All the octaves are summed up together and then divided by the sum of all the amplitudes.

As mentioned before, each octave has a different amplitude. The starting amplitude, before iteratively creating the octaves, has a value of 1. Each iteration, before the octave is calculated, the amplitude is multiplied by a value called persistence. The persistence is the value that controls the weight (amplitude) of each successive octave.

Besides the number of octaves and the persistence, two other values can be adjusted to alter the result of the noise function: bias and gain. Bias can be seen as the "brightness" of the noise function and gain as the "contrast" of the noise function. Both bias and gain range from zero to one, inclusive, where a value of 0.5 does not modify the noise function. See figure 3.7 and figure 3.8 for examples of bias and gain.

Bias is used to bend a function either upwards or downwards over the $[0,1]$ interval, as seen in figure 3.7.

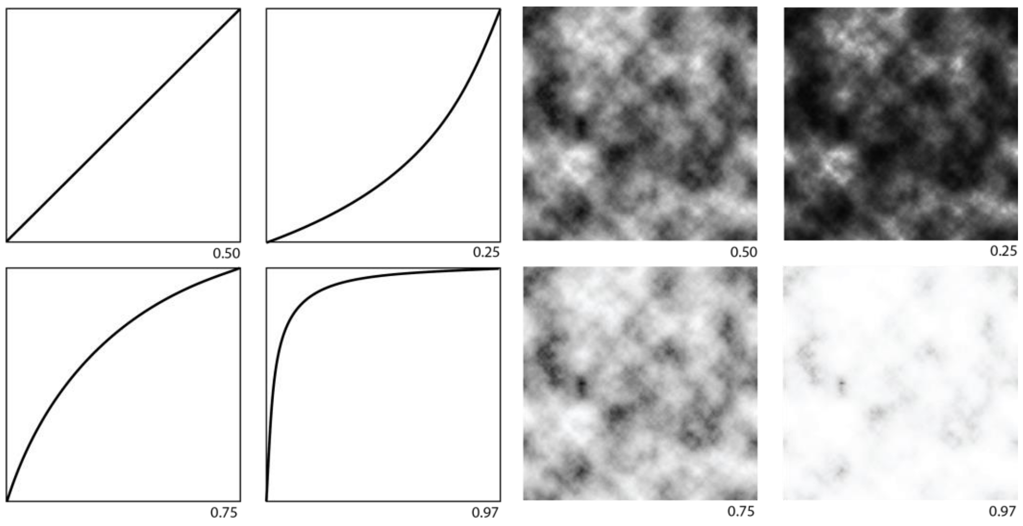

Figure 3.7 – Bias examples

The bias function is defined as shown in equation 3.6.

$$bias(v, b) = \frac{v}{\left(\frac{1}{b}\right) * (1 - v) + 1} \tag{3.6}$$

Where v is the input value and b is the bias value.

Gain is used to help shape how fast the midrange of a function goes from 0 to 1. A higher gain value means a higher rate in change, as seen in figure 3.8.

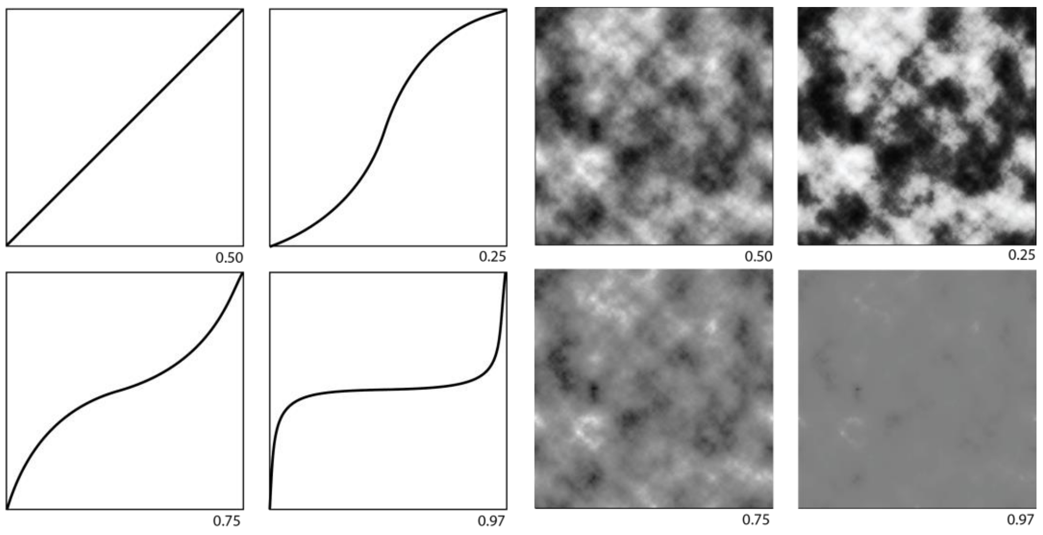

Figure 3.8 – Gain examples

The gain function is defined as shown in equation 3.7.

$$gain(v,g) = \begin{cases} \dfrac{bias(v*2,g)}{2}, & v < 0.5 \\ \dfrac{bias(v*2-1,1-g)}{2} + 0.5, & v \geq 0.5 \end{cases} \tag{3.7}$$

Where $bias()$ is equation 3.6, v is the input value and g is the gain value.

The complete noise creation method from begin to end is listed in Algorithm 3.2. Three functions are used in this algorithm:

cubicInterp	This function samples the correct values from the *baseNoise* array and applies cubic interpolation to these sampled values, as described in section 3.3.2.1.
getBias	Calculates the bias according to equation 3.6.
getGain	Calculates the gain according to equation 3.7.

Algorithm 3.2 Generate Value noise (*octaves*, *persistence*, *bias*, *gain*)
Input. The number of *octaves*, the *persistence*, *bias* and *gain*.
Output. A texture containing the Value noise.

```
1.  Let $T_{res}$ be a texture that will be used to write the final result to.
2.  Let baseNoise be a 2D array with the size of $T_{res}$.
3.  Fill baseNoise with random values ranging from 0 to 1.
4.  amplitude ← 1
5.  totalAmplitude ← 0

6.  For octaves downto 0 do
7.      amplitude ← amplitude * persistence
8.      totalAmplitude ← totalAmplitude + amplitude
9.      Let oct be the current octave
10.     period ← $2^{oct}$
11.     frequency ← 1 / period
12.     For all elements in baseNoise do
13.         Calculate vertical sampling positions and vertical blend value, let them be
            called $y_0, y_1, y_2, y_3$ and vertBlend, respectively.
14.         Calculate horizontal sampling positions and horizontal blend value, let them
            be called $x_0, x_1, x_2, x_3$ and horBlend, respectively.
15.         row1 ← cubicInterp($x_0, x_1, x_2, x_3, y_0$, horBlend)
16.         row2 ← cubicInterp($x_0, x_1, x_2, x_3, y_1$, horBlend)
17.         row3 ← cubicInterp($x_0, x_1, x_2, x_3, y_2$, horBlend)
18.         row4 ← cubicInterp($x_0, x_1, x_2, x_3, y_3$, horBlend)
19.         $T_{res}$ ← cubicInterp(row1, row2, row3, row4, vertBlend) * amplitude

20. For all pixels $\rho$ in $T_{res}$ do
21.     noiseVal ← $\rho$ / totalAmplitude
22.     $T_{res}$ ← getGain( getBias(noiseVal, bias), gain )

23. return $T_{res}$
```

3.3.3 Gradient edge generation

To be able to simulate the red glow that is present around wounds (see chapter 4), we have developed a technique that is able to draw a gradient around an input shape (see section 3.2). This technique can draw a gradient inwards or outwards from the hull. The shape texture is used to determine if a pixel is on the inside or outside of input shape. The length of the vector from the pixel to the closest hull edge is used to calculate a blending value, which is used to draw the gradient. The closest distance from a pixel to a hull edge is calculated using algorithm 3.3.

Algorithm 3.3 Calculate minimum distance from point to line segment (*l1*, *l2*, *p*)
Input. Two end point of the line segment *l1* and *l2* and point *p*.
Output. The minimum distance from point *p* to line segment from *l1* to *l2*

```
1.  Let v be a vector from l1 to l2
2.  Let w be a vector from p to l1
3.  If dot product of w and v > 0 then
4.      Let z be a vector from l1 to p
5.      return square-root of the dot product of z and z
```

```
6.  Let v be a vector from l2 to l1.
7.  Let w be a vector from p to l2.
8.  If dot product of w and v > 0 then
9.      Let z be a vector from l2 to p
10.     return square-root of the dot product of z and z

11. v ← (v.y, -v.x)
12. Normalize v
13. Let w be a line from l1 to p
14. return the absolute value of the dot product of v with w
```

The method to draw the gradient itself is listed in Algorithm 3.4. To calculate the closest distance from a pixel to the hull, iterate over all edges in the hull and for every edge calculate the minimum distance from the pixel to the edge, as shown in algorithm 3.3. The smallest calculated distance from a pixel to a hull edge is the closest distance from the pixel to the hull.

Algorithm 3.4 Draw gradient around hull (*T*, *hull*, *drawShape, thickness*, *renderSide*, *colorFrom*, *colorTo*)
Input. The input texture *T* to draw on, the *hull*, the drawn shape texture *drawShape,* the *thickness* of the gradient, the side to render on *renderSide* and the two colors to fade between *colorFrom* and *colorTo*.
Output. A texture containing the drawn gradient.

```
1.  For all pixels ρ in T do
2.      If renderSide = inside and drawShape color at location ρ is black
3.        or renderSide = outside and drawShape color at location ρ is white do
4.          Set color of ρ to black
5.          continue

6.      minDist ← Find closest distance from ρ to hull
7.      If minDist <= thickness do
8.          lerpValue ← minDist / thickness
9.          newColor ← Lerp(colorFrom, colorTo, lerpValue)
10.         Set color of ρ to newColor
11.     Else do
12.         Set color of ρ to black
```

3.4 Conclusion

This chapter introduced a framework for procedurally generating textures, aimed at wound healing simulation. The framework consists of a timeline containing key positions. Each key position outputs one or more textures. The output of an arbitrary point on the timeline is calculated by interpolating between the outputs of the key positions.

Several techniques are also described which can be used to create the output for a key position. Shape erosion is used to add small details to the edge of a shape texture. Noise generation is be used add some variation and gives the output of a key position a more natural look. Gradient edges can be used to simulate the red glow around a wound.

The introduced framework forms the basis for abrasion healing simulation, which is described in the next chapter.

Chapter 4

Modeling the healing process

In the previous chapter a framework was introduced for procedurally generating textures, aimed at wound simulation. This chapter describes a simulation which uses this framework to simulate the healing process of an abrasion. Section 4.1 describes the resources that are used in the simulation. Section 4.2 describes the user interface for the simulation. The main part of the simulation, the modeled wound phases, is described in section 4.3. Performance results are given in section 4.4. A summary is provided in section 4.5. Most of the parameters given in this chapter are obtained through trail-and-error methods, especially for the noise functions used. When choosing these parameters, a reference image for each phase was used for comparison to see which parameters could be used best. These reference images can be found in this chapter, in section 4.3.

4.1 Resources

The simulation described in this chapter uses a mesh of a left hand of an adult male. This hand is modeled by a 3D artist called Aphrodite3d. This mesh can be bought online at TurboSquid.com [8]. The mesh comes with four textures, modeling the skin of the hand itself. The mesh and the accompanying textures are shown in figure 4.1 in clockwise order: upper skin color, lower skin color, skin specularity and a skin normal texture. The upper skin color texture is used to model healthy skin, which can be seen on the mesh on the left side of figure 4.1. The lower skin color is used for modeling the wound phases (see section 4.3). The skin specularity texture is used for defining skin reflection and the skin normal texture is used to give the illusion of extra detail on the mesh.

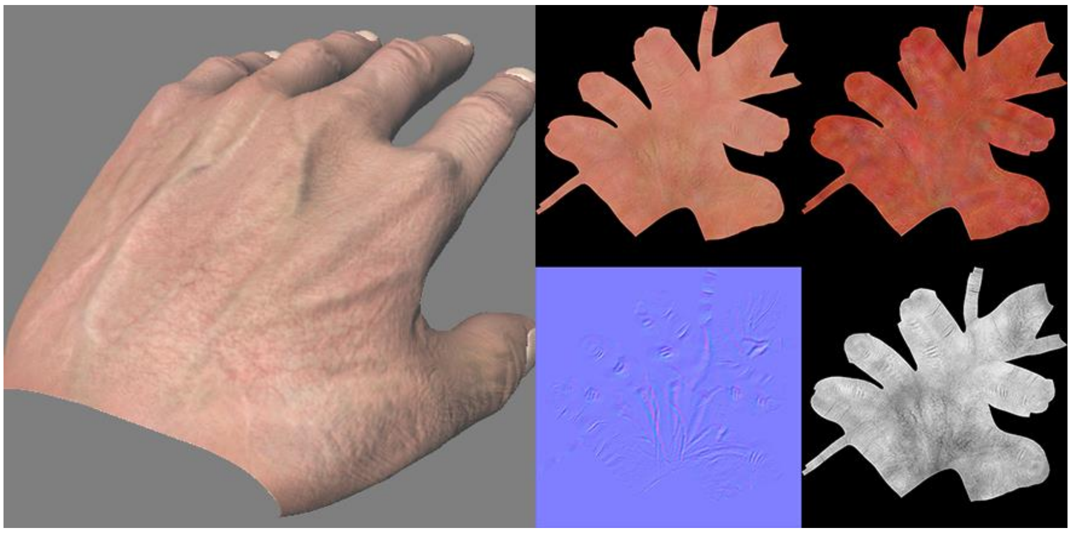
Figure 4.1 – Hand mesh and its skin textures

4.2 User interface

The simulation contains a user interface where the user can do two things: draw on the hand mesh to define the wound shape and scroll through the healing process in real-time.

The scrolling is done by connecting a slider in the user interface with the timeline in the framework described in chapter 3. This connection is a 1-on-1 connection, meaning the far left side of the slider is the beginning of the timeline and the far right side of the slider is the end of the timeline. The drawing of the wound shape is done via texture painting. The user draws the shape of an input shape texture on the mesh. A convex hull of the drawn shape is then calculated to be used as the hull of the input shape. An input shape texture is then created using the calculated hull.

4.2.1 UV recording

The first step in creating a convex hull is to get a recording of the path the user has drawn on the mesh. This recording is a list of UV coordinates on the mesh the user has drawn on. A ray is cast from the camera through the mouse' viewport position into the scene. If this ray hits the mesh, the user is currently drawing on the mesh, as seen in figure 4.2.

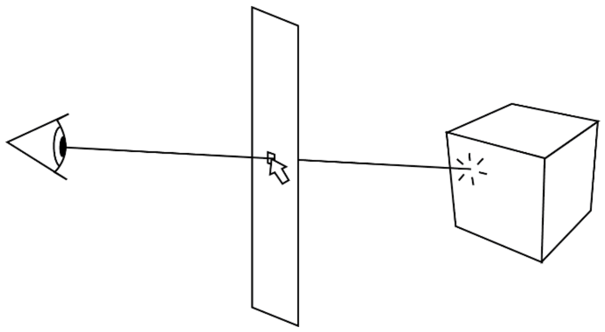

Figure 4.2 – Raycasting from camera through mouse position

If the ray hits the mesh, barycentric interpolation is applied to the hit triangle on that mesh. The encoded UV coordinates of the three vertices of the triangle, along with the position information, can be used to calculate the UV coordinate of the hit point.

First, the barycentric coordinates of the hit point position inside the triangle is calculated. Let A, B and C be the three vertices of the triangle, P be the hit point of the ray on that triangle, u be a vector from vertex A to B and v be a vector from vertex A to C, as seen in figure 4.3.

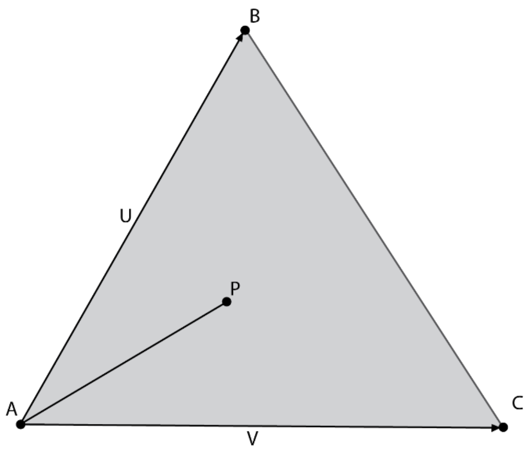

Figure 4.3 – Calculating position p inside triangle

The position of point P can be calculated using barycentric coordinates:

$$P_{pos} = sA_{pos} + rB_{pos} + tC_{pos} \tag{4.1}$$

The barycentric coordinates themselves, s, r and t, can be calculated as shown in equation 4.2.

$$r = \frac{\|v \times w\|}{\|u \times v\|} \tag{4.2}$$

$$t = \frac{\|u \times w\|}{\|u \times v\|}$$

$$s = 1 - r - t$$

These barycentric coordinates can then be used to calculate the UV coordinates of point P:

$$P_{uv} = sA_{uv} + rB_{uv} + tC_{uv} \tag{4.3}$$

When the user is drawing, each UV coordinate of hit points on the mesh are stored in an array. When the user is done drawing, the stored UV coordinate points are filtered. This filtering is done by calculating the convex hull of the drawn points. The method for calculating the convex hull of a set of points used for this thesis is described by De Berg & Kreveld [9]. The pseudo-code for this method is listed in Algorithm 4.1.

Algorithm 4.1 Calculate convex hull of a set of points(P)
Input. A set P of points in the plane.
Output. A list containing the vertices of the convex hull in clockwise order.

```
1.  Sort the points by x-coordinate, resulting in a sequence $p_1,…,p_n$
2.  Put the points $p_1$ and $p_2$ in a list $L_{upper}$, with $p_1$ as the first point.
3.  For $i \leftarrow 3$ to $n$
4.      Do Append $p_i$ to $L_{upper}$.
5.          While $L_{upper}$ contains more than two points and the last three points in $L_{upper}$
                    do  not make a right turn
6.              do Delete the middle of the last three points from $L_{upper}$
7.  Put the points $p_n$ and $p_{n-1}$ in a list $L_{lower}$, with $p_n$ as the first point.
8.  For $i \leftarrow 2 - 2$ downto $1$
9.      Do Append $p_i$ to $L_{lower}$
10.         While $L_{lower}$ contains more than two points and the last three points in $L_{lower}$
                    do not make a right turn
11.             do Delete the middle of the last three points from $L_{lower}$.
12. Remove the first and the last point from $L_{lower}$ to avoid duplication of the points
    where the upper and lower hull meet.
13. Append $L_{lower}$ to $L_{upper}$ and call the resulting list $L$.
14. Return $L$
```

In both for-loops in Algorithm 4.1, De Berg & Kreveld talk about three points that do not make a right turn. To see if those three points make a right turn or not, one simply has to check if the last point of those three lies on the left or the right of a line segment going from the first point to the second point. Let p_1, p_2 and p_3 be three points in 2D space. If you take the determinant of a 2x2 matrix created using those points, one can determine if the three points do or do not make a right turn:

$$\begin{gathered} det = \begin{vmatrix} p_2.x - p_1.x & p_3.x - p_1.x \\ p_2.y - p_1.y & p_3.y - p_1.y \end{vmatrix} \\ isRightTurn(det) = \begin{cases} true, & det < 0 \\ false, & det \geq 0 \end{cases} \end{gathered} \tag{4.4}$$

If the determinant is smaller than 1, the last of the three points lies on the right of a line segment going from the first point to the second point, which means the three points make a right turn.

4.2.2 Filling the shape texture

The shape texture is a binary texture of a convex hull. This means that each pixel that lies within the convex hull, as calculated in section 4.2.1, gets a white value. Each pixel that does not lie within the convex hull gets a black value.

The method for filling the shape texture used in this thesis is, to keep things simple, quite trivial. Smarter and more efficient methods can be used for this (see Future work, section 5.1). The method is listed in Algorithm 4.2.

Two functions are used in the algorithm:

GetHullBoundingBox	This is a small function that returns all the pixels that lay within the bounding box of the convex hull.
IsOnRightSide	This function determines if an input point lies on the right side of an input line segment or not. This is the implementation of the method described in section 4.2.1, equation 4.4.

Algorithm 4.2 Fill shape texture (*H*, *T*)
Input. A convex hull *H* consisting of a list of points in the plane and a texture *T*.
Output. A texture *T* containing a filled binary shape the shape of the convex hull *H*.

```
1.  BBox ← GetHullBoundingBox(H, T)
2.  isInHull ← true
3.  For each pixel $\rho$ in BBox do
4.      For each edge e in H do
5.          If IsOnRightSide(e, $\rho$) = true then
6.              isInHull ← false
7.              Break from For-loop
8.      If isInHull = true then
9.          Set color value of pixel $\rho$ in texture T to white
10.     Else Set color value of pixel $\rho$ in texture T to black
11. Return T
```

4.3 Wound phases

This section describes how the wound healing process of an abrasion wound is modeled in the simulation.

Five different key positions from the wound healing process have been modeled and placed on the timeline. These key positions are named after their relative position in the wound healing process: vasoconstriction, vasodilation, scab-forming, scab and maturation. Three wound healing phases are modeled by two key positions and two wound healing phases are modeled by one key position, as seen in figure 4.4. The double key positions output the same textures however. This is to keep the phase visible for a set period of time and not to immediately fade away to the next key position. When a position on the timeline is chosen that happens to be in between one of these double key positions, interpolation is done between key positions that output the same textures. Anywhere in between those double key positions will look the same. Also note that there is a sudden jump between the scab and maturation phases. See section 4.3.5 for more information about this.

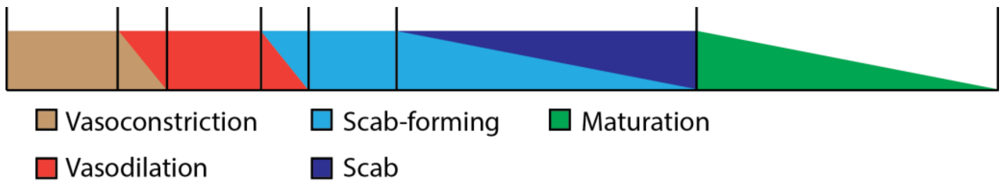

Figure 4.4 – Blending of key positions in the timeline

As mentioned in section 3.2, key positions can output one or more textures. The key positions in this abrasion simulation generate three types of textures as output: color textures, reflection textures and height textures. Each texture is filled using pixel shaders. The color textures are used to define the color of the wound, normal textures are used to add the illusion of extra detail on the hand mesh, reflection textures are used to define the reflectiveness of the wound and height textures are used to displace the vertices of the hand mesh to add thickness to the wound (see scab phase in section 4.3.4).

4.3.1 Vasoconstriction

Vasoconstriction is the first thing that happens in the healing process of a wound and is thus the first frame in the simulation. Vasoconstriction needs to model the initial state of the wound, which is, with an abrasion, the absence of the upper layer of the skin. The vasoconstriction key position makes smart use of this fact by using the lower skin color texture. Remember that the input shape texture is a binary texture: white pixels indicate a position on the shape, while black pixels indicate a position not on the shape. The input shape texture is sampled and on all pixels which are not black, the color of the lower skin color texture is applied. The color of upper skin color is applied to all pixels which are black.

As mentioned before in the introduction (chapter 1), flowing blood is not simulated in our model, so this is the only thing that the vasoconstriction frame does.

The final result of the vasoconstriction key position is shown in figure 4.5. A reference image [10] is shown on the left and the final render is shown on the right. Notice that in the final render on the right the wound consists of one big circle, while the reference on the left consists of a combination of multiple shapes. This is because the input shape is a convex shape, as described in 4.2. See future work (section 5.1) for more on wound shapes.

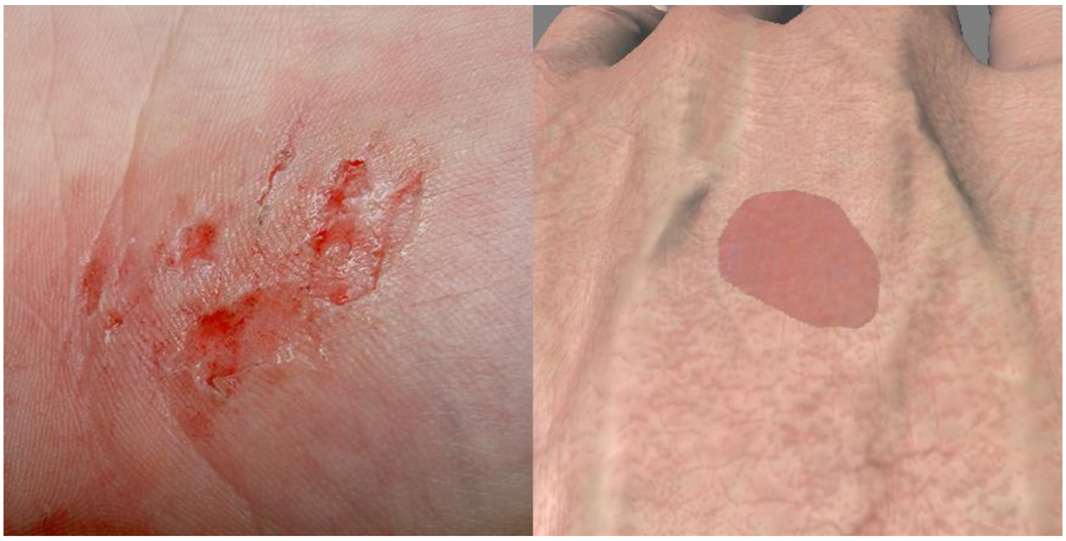

Figure 4.5 – Vasoconstriction comparison

4.3.2 Vasodilation

After vasoconstriction the blood vessels dilate during the vasodilation. This causes the skin around the wound site to have a slightly redder color. Vasodilation happens quite quickly after vasoconstriction. To simulate the increased blood flow through the veins underneath and around the wound, a red color is used (0.79 red, 0.0 green, 0.0 blue). Inside the wound, the lower skin texture of the hand is multiplied with this red color to make the wound appear redder.

A red glow on the skin around the wound is also present during vasodilation. This effect is simulated by generating a gradient outwards from the edge of the wound. The gradient is calculated using the convex hull around the wound site and the wound shape texture. More on calculating this gradient is found in section 3.3.3. The gradient is drawn outwards with thickness set to 100, fading from a red color (0.79 red, 0.0 green, 0.0 blue) to the upper skin color texture.

A small edge of scraped skin is visible around the border of the wound, see left side of figure 4.7. This is an edge of half scraped off skin, which is why it is only present on the edges of the wound and mostly in the direction of the scraping that caused the abrasion. This edge is created using the erosion technique described in section 3.3.1 applied to the wound shape texture. The more erosion is applied, the thicker this edge will become. The erosion technique is applied to the wound shape twice: first with a kernel width varying between 3 and 5 and second with a kernel width varying between 3 and 10. This erode wound shape is laid on top of the original wound shape, causing an edge to become visible. The color of the edge is set to the upper skin color multiplied by a light grey color (0.95 red, 0.95 green, 0.95 blue).

Inside the wound itself, scrape marks are visible mostly seen by the reflection of the light. These scrape marks are defined by the dented texture of the wound. Light falls on these dents and reflects back to the viewer. This creates the effect of fine lines over the surface of the wound. This effect is modeled by creating a high frequency noise texture (see section 3.3.2) and scaling the resulting noise texture in the y-axis 6400%. This creates long, thin white lines, as seen in figure 4.6. This texture is then used as the reflection texture of the wound. The noise is created with 5 octaves, a persistence of 0.09, a bias of 0.5 and a gain of 0.43.

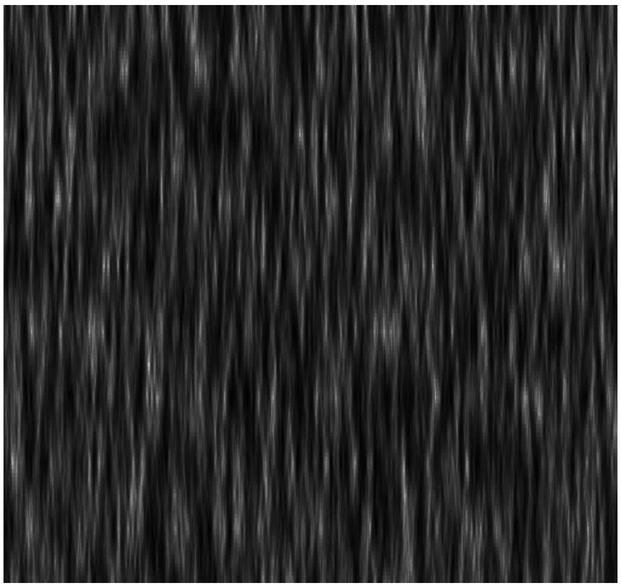

Figure 4.6 – Stretched noise along the y-axis

All these effects put together gives the result as seen in figure 4.7. A reference image [11] is shown on the left, the final render is shown on the right.

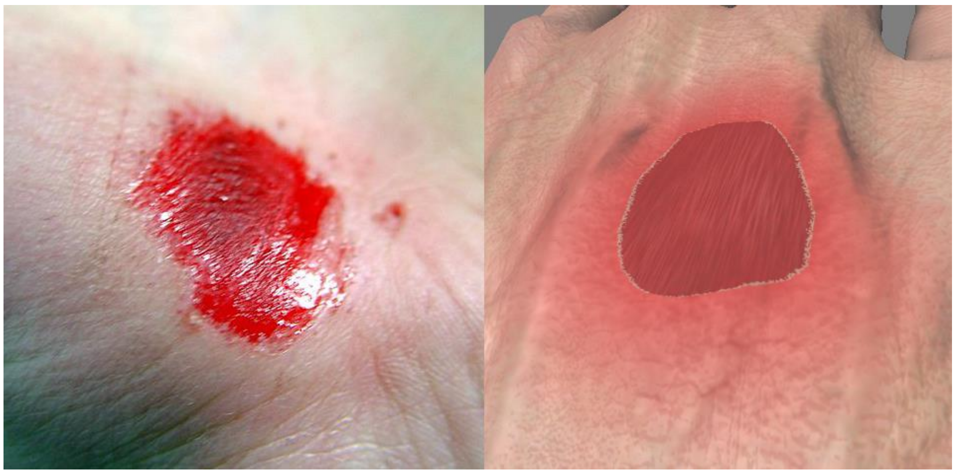

Figure 4.7 – Vasodilation comparison

4.3.3 Scab-forming phase

The scab-forming phase is a collective name for all the wound phases after vasodilation, but before the scab is formed. In this timespan, multiple wound phases occur, but collectively result in the same visual effect. This is why they are grouped together. Do note that scab-forming is not a medical term, it is just a name used for this simulation.

Just like with vasodilation, a red glow is visible around the wound. It is simulated in the same way it is done with vasodilation.

As one can see in the reference image [12] in figure 4.9, there are redder parts and lighter parts on the inside of the wound. The redder parts are carved deeper into the skin than the lighter parts. The whole inside has a noise-like pattern, fading between the redder and lighter parts. This is why the inside of the scab-forming phase is modeled using layered noise textures. One low frequency noise texture with a high persistence and low gain is used for the bottom layer (5 octaves, 0.85 persistence, 0.54 bias, 0.21 gain) and one noise texture with low bias and gain is used for the reddest parts of the wound (5 octaves, 0.48 persistence, 0.16 bias, 0.15 gain): the “speckles”, as seen in figure 4.8.

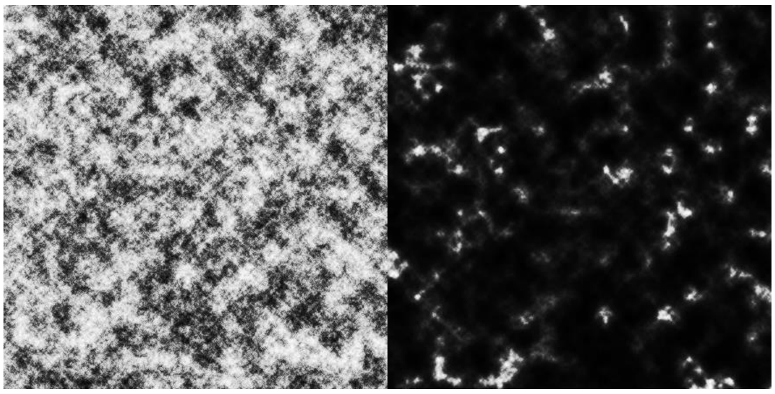
Figure 4.8 – Scab-forming noise

The low frequency noise texture is blended with the upper skin color texture and the lower skin color texture. The whiter the noise, the more of the upper skin color is used and the darker the noise, the more of the lower skin color is used. Equation 4.5 shows how this is calculated.

$$\begin{gathered} lowerWoundColor = lowerSkinColor.rgb * lowerNoiseColor.r \\ + upperSkinColor.rgb * (1 - lowerNoiseColor.r) \end{gathered} \tag{4.5}$$

Where $lowerSkinColor$ is a vector containing the color of the lower skin texture at the current pixel coordinate and $upperSkinColor$ is a vector containing the color of the upper skin color texture at the current pixel coordinate.

The right texture seen in figure 4.8 is multiplied with a reddish color (0.88 red, 0.20 green, 0.22 blue) and then blended with $lowerWoundColor$ in a similar way to the method seen in equation 4.5. This creates red “speckles” on the wound, indicating the deepest parts of the wound. The blending is seen in equation 4.6.

$$finalColor = speckleWoundColor.rgb * speckleWoundcolor.r \tag{4.6}$$

$$+ lowerWoundColor.rgb * (1 - speckleWoundColor.r)$$

Where $speckleWoundColor$ is a vector containing the color of the red "speckles" at the current pixel coordinate and $finalColor$ is the final output color of the key position.

The final result of the key position is shown in figure 4.9 along with a reference image.

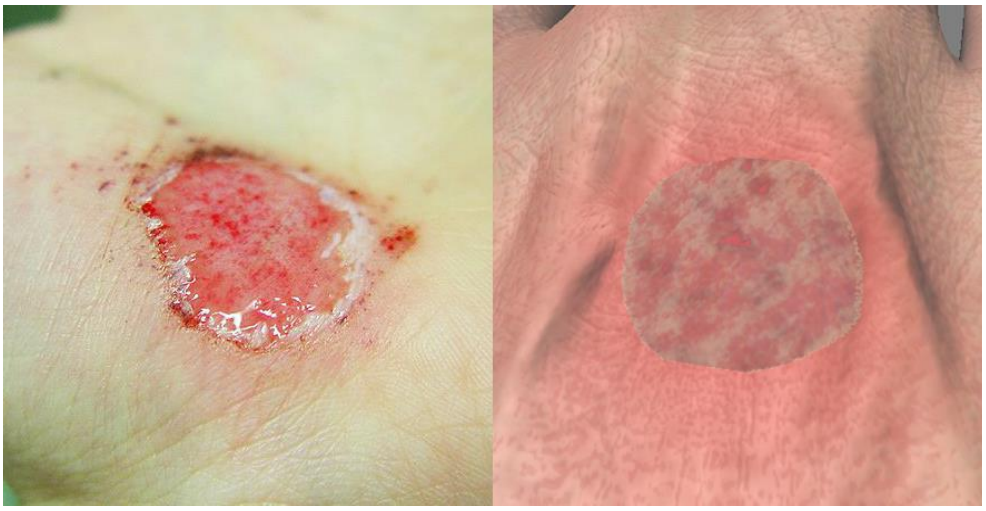

Figure 4.9 – Scab-forming comparison

4.3.4 Scab

The scab key position is placed fairly long after the scab-forming phase. The scab has a rough surface and has thickness, which means it sticks out of the skin. The thickness of the scab is simulated by generating a height texture (displacement map). Height textures are textures where, instead of colors, height information is stored (often in the red channel of the color). The height information indicates the amount the vertices of a mesh need to displace outwards. The whiter the color value, the further the vertex needs to displace outwards. This technique is dependent on the vertex density of the mesh on the place of the wound. This means that a low vertex density gives low detail in the displacement and a high vertex density gives high detail in the displacement. There are several ways to make sure the mesh has a high enough vertex density to display enough displacement detail. For example, one could pre-subdivide the mesh in separate 3D authoring software (e.g. Autodesk 3DS Max or Autodesk Maya). One could also make use of hardware tessellation. A graphics processor that supports DirectX 11 shader model 5.0 or OpenGL 4.0 is required. Our wound simulation uses hardware tessellation to dynamically and automatically subdivide the mesh. For more information on hardware tessellation the reader is referred to [13].

The height texture is created by multiplying two noise textures with each other, see figure 4.10. One noise texture (the far left) works as a mask for the other noise texture (the one in the middle). This creates the rough, irregular texture seen on the right. If this texture is applied as a height texture for the wound, it will give the rough, irregular surface that a scab has (see left side of figure 4.12). The left noise texture in figure 4.10 is created with 7 octaves, 1.0 persistence, 0.44 bias and 0.02 gain. The middle noise is created with 5 octaves, 0.54 persistence, 0.3 bias and 0.69 gain.

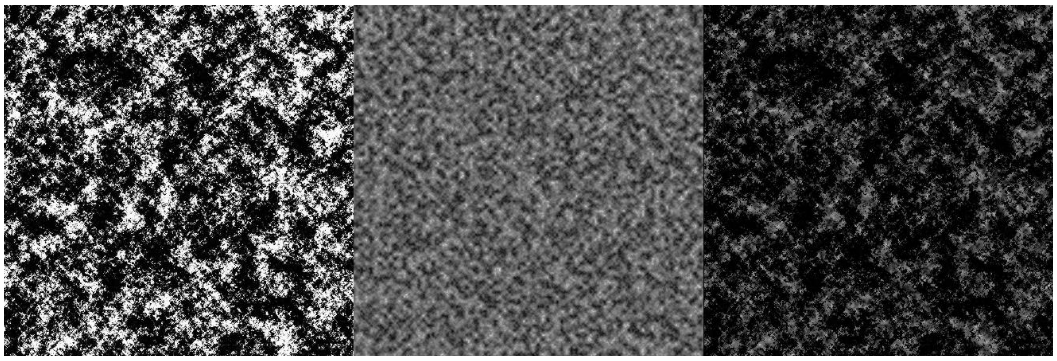
Figure 4.10 – Scab height noise textures

The color of the scab is made by a compilation of several textures. As can be seen in the reference image [14] in figure 4.12, the color of the scab is as irregular as the surface of the scab. We compose this color by modeling the darker red colors and the lighter yellowish colors and combining these colors to a final result. The darker red colors are more visible near de edge of the scab, thus we model this by creating a gradient edge inwards and masking this gradient with a noise texture. This creates the effect seen on the right side of figure 4.11. The gradient is drawn inwards with thickness set to 100, fading from a red color (0.5 red, 0.04 green, 0.03 blue) to black. The noise texture, seen on the left of figure 4.11, is created with 5 octaves, a persistence of 0.33, 0.60 bias and 0.59 gain.

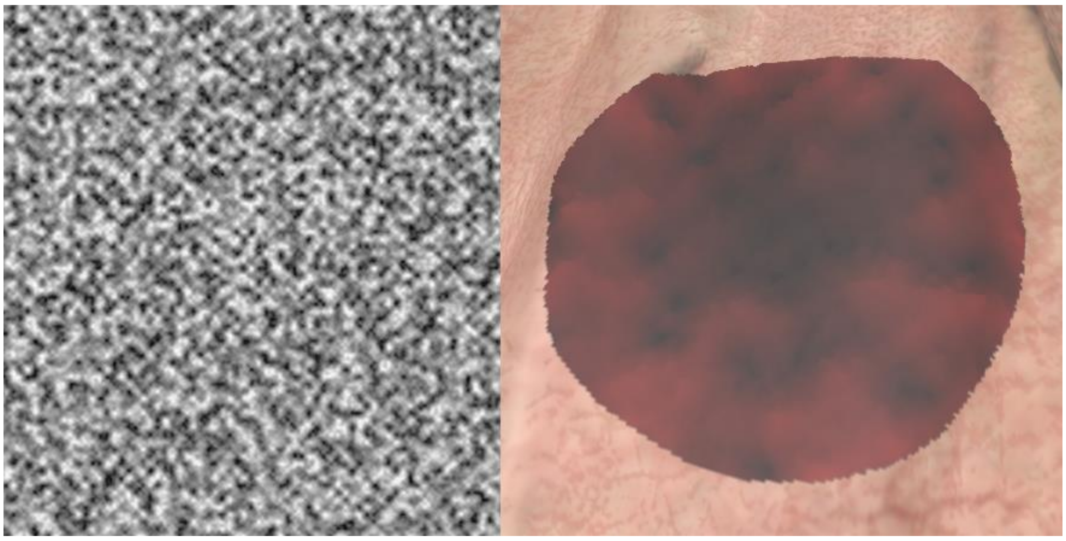
Figure 4.11 – Red scab details

The lighter yellowish colors are created using the height texture seen on the right of figure 4.10. The color is created by linearly interpolating between two colors, a lighter color (0.78 red, 0.48 green, 0.27 blue) and a darker color (0.39 red, 0.12 green, 0.11 blue), and using the height values from the height texture as interpolation values. The final scab color is created by blending the lighter yellowish colors and the darker red colors together in a same way that was done in the scab-forming phase (section 4.3.3), as shown in equation 4.7.

$$finalColor = innerColor + (gradientColor * (white - innerColor)) \tag{4.7}$$

Where $innerColor$ the lighter yellowish color as described above, $gradientColor$ is the darker red color as described above and $white$ is a pure white color.

As seen in the reference image on the left side of figure 4.12, a red glow is also visible around the wound itself, just like the vasodilation and the scab-forming phase. The glow around the wound in the scab phase is simulated in the same way as those phases do.

To add a hint of reflectiveness, the scab phase also generates a reflection texture. This is a greyscale texture indicating the reflectiveness of the surface it is used on. The whiter the color, the more the surface should reflect light. The reflection texture for the scab is created using a noise texture. It is a low frequency noise texture with high persistence, low bias and high gain. The noise has 10 octaves, 1.0 persistence, 0.25 bias and 0.95 gain.

Combining all these effects together gives the result as shown on the left side of figure 4.12. This is a rendered image containing the red glow, the displaced mesh vertices via the height map and the applied yellow and red scab color.

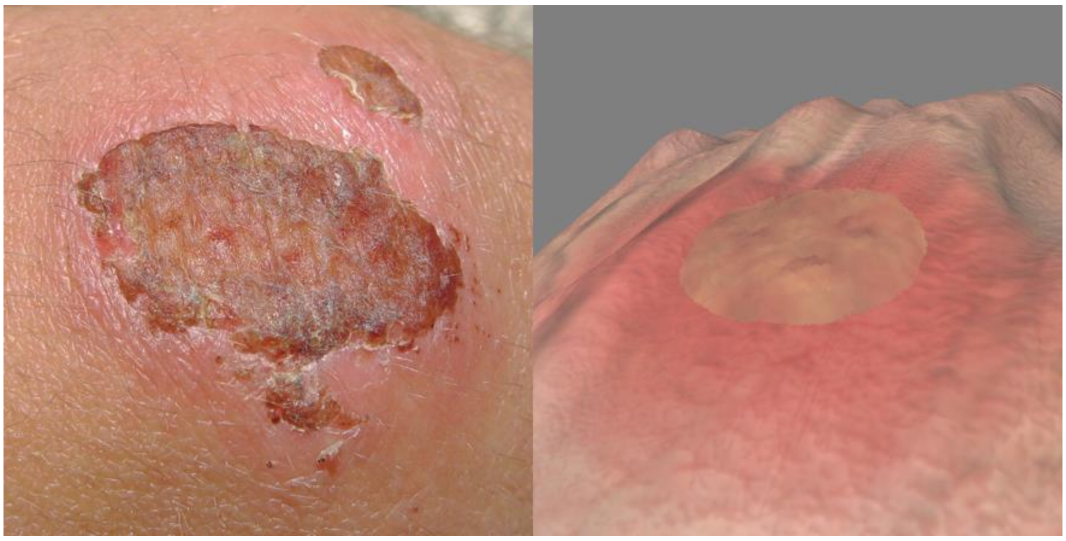
Figure 4.12 – Scab comparison

4.3.5 Maturation

Maturation is the last part of the simulation. Instead of fading from the scab to maturation like other key positions, maturation starts immediately after the scab has disappeared. A scab on a real wound would slowly let go of the skin and fall off. This would require that the mesh of the hand that makes up the scab in the simulation would slowly let go of the hand mesh. However, detaching meshes are not supported by the proposed framework and the wound simulation, so the falling off will be skipped. Instead, the simulated maturation phase starts immediately after the simulated scab phase with no interpolation. This is achieved by placing the key position of the maturation phase so close to the scab phase that the actual interpolation between the scab and maturation phases are not visible anymore.

As mentioned in section 2.1, a wound can become 40 to 80% smaller after contraction. This is where we begin the maturation phase. The scab has just fallen off and the wound is 40 to 80% smaller than its original size. During the maturation phase, the wound keeps getting smaller until it eventually disappears. To be able to simulate this disappearance, we make use of the interpolation value that is used to interpolate between key positions. This interpolation value starts at 0 at the beginning of the maturation phase and ends with 1 at the end of the maturation phase (which is also the end of the entire wound timeline). At each position in the maturation phase, the key position that represents the maturation phase is rerendered.

The rendering process of the maturation phase begins by calculating the maximum diameter of the input wound shape. This is the distance the wound must shrink in order to be disappear at the end of the wound timeline. This distance is called the closing distance of the wound and its calculation is shown in algorithm 4.3.

Algorithm 4.3 Calculate closing distance (*H*)
Input. A hull *H* consisting of a list of points in the plane.
Output. The closing distance of the wound represented by hull *H*.

```
1. maxDist ← 0
2. For each edge e in H do
3.     For each edge f in H do
4.         Let v be a vector from edge e to edge f
5.         maxDist ← max( maxDist, length of v )
6. Return maxDist
```

The key position of the maturation phase outputs one color texture. This texture is created by a series of gradients drawn inside the input shape of the wound. There are two gradients: one that runs from an outer gradient border to an inner gradient border and one that runs from the same inner gradient border to the center of the input shape of the wound, as seen in figure 4.13. These gradients are calculated differently than the gradient rendering technique provided in section 3.3.3, because that the rendering process of these two gradients is a bit more complex than the standard gradient provided in that section.

As seen in the reference image [15] on the left side of figure 4.14, there is still a small border of red glow around the wound and the inside has a white tint to it. The gradient that runs from the outer gradient border to the inner gradient border simulates the outer half of the red glow: it fades from the skin color of the mesh to the red color that the glow around the wound has. This is the same color red used for the glow around the wound in the previously described phases. The second gradient that runs from the inner gradient border to the center of the input shape of the wound fades from the same red color of the first gradient to the whiter tint (0.79 red, 0.41 green, 0.41 blue) seen on the inside of the wound.

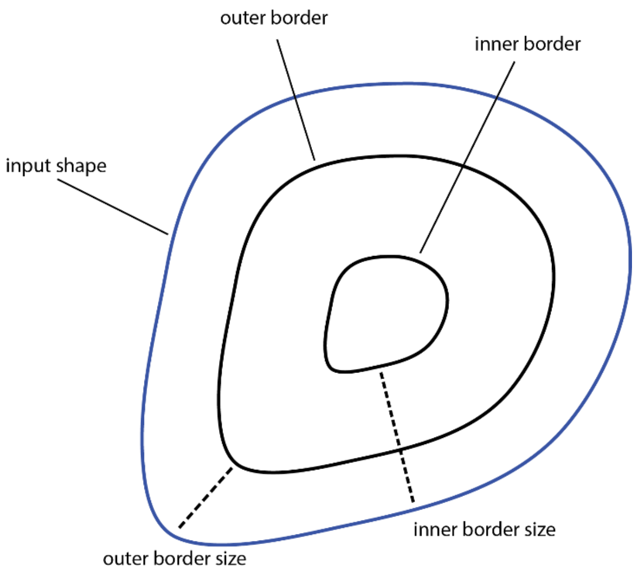

Figure 4.13 – Gradient borders of the maturation phase

The thickness of these outer and inner gradient borders depends on the position the maturation phase is in (interpolation value). When the maturation phase starts, the outer gradient border has a thickness of 0 and when the maturation phase has ended the outer gradient border has a thickness of half the closing distance of the wound. As mentioned before, when the maturation phase begins, the wound is 40 to 80% of its original size due to constriction. This means that the size of the inner gradient border varies between 40 and 80% of the maximum size of the outer gradient border. The calculation of the outer and inner gradient borders is shown in equation 4.8.

$$\begin{gathered} maxOuterGradBorder = closingDistance * 0.5 \\ outerGradBorder = maxOuterGradBorder * position \\ innerGradBorder = outerGradBorder + maxOuterGradBorder * rand(0.4, 0.8) \end{gathered} \tag{4.8}$$

Where $maxOuterGradBorder$ is the maximum size of the outer gradient border, $outerGradBorder$ is the size of the outer gradient border, $position$ is the position the maturation phase is in, $innerGradBorder$ is the inner gradient border size and $rand(0.4, 0.8)$ is a function that outputs a floating point number between 0.4 and 0.8, inclusive.

The algorithm for drawing the maturation phase is given in algorithm 4.4. Running this algorithm gives the rendered maturation phase as seen on the right side of figure 4.14. On function and two constants are used in this algorithm:

Lerp	This function calculates the linear interpolation between the two input values using the interpolation value (the third input value)
WOUND_RED	This is the red color used for the first (outer) gradient as described in the section above.
WOUND_WHITE	This is the white color used for the second (inner) gradient as described in the section above.

Algorithm 4.4 Draw Maturation phase gradients (*H*, *closingDist*, *outerBorder*, *innerBorder*, *skinTex*, *shapeTex*, *outTex*)

Input. A hull *H*, closing distance *clostingDist*, outer gradient border *outerBorder*, inner gradient border *innerBorder*, skin color texture *skinTex*, input shape texture *shapeTex* and an output texture *outTex*

Output. The drawn maturation phase on texture *outTex*

```
1.  For each pixel $\rho$ in outTex do
2.      minDist ← Minimum distance from $\rho$ to H using algorithm 3.3
3.      If pixel $\rho_{shape}$ in texture shapeTex with pixel coordinates of $\rho$ is white do
4.          If minDist <= outerBorder do
5.              Set color of $\rho$ to color of the pixel with same coordinates in skinTex
6.          Else if minDist <= innerBorder do
7.              minDist ← minDist – outerBorder
8.              lerpVal ← minDist / (innerBorder - outerBorder)
9.              pixColor ← Lerp( pixel with coordinates of $\rho$ in skinTex, WOUND_RED,
                    lerpVal )
10.             Set color of $\rho$ to pixColor
11.         Else do
12.             minDist ← minDist – innerBorder
13.             lerpVal ← minDist / ((closingDist * 0.5) - innerBorder)
14.             pixColor ← Lerp( WOUND_RED, WOUND_WHITE, lerpVal )
15.             Set color of $\rho$ to pixColor
16. Return outTex
```

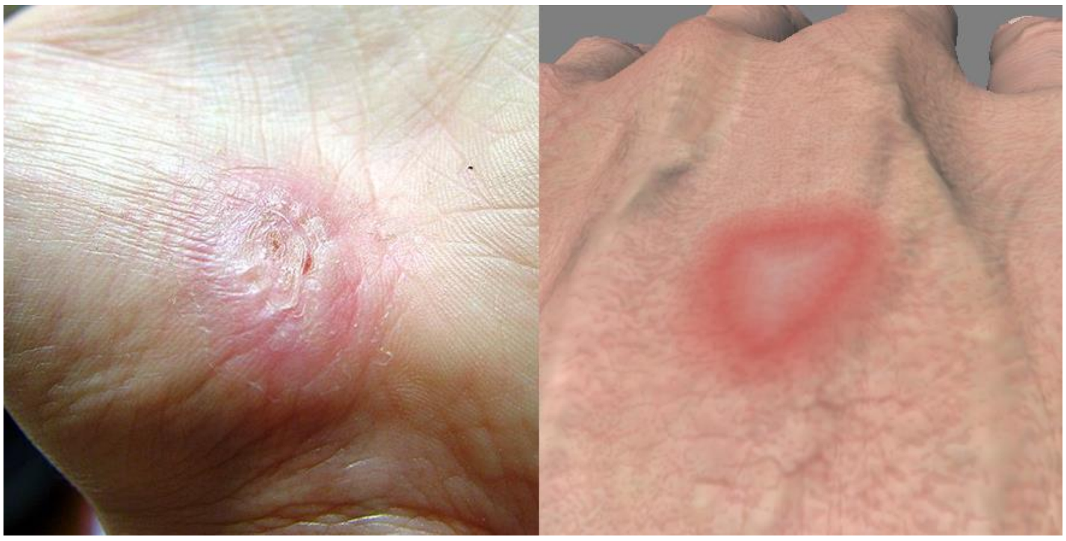
Figure 4.14 – Maturation comparison

4.4 Performance

Several performance measurements have been done on the in this chapter described wound simulation system. The specifications of the test system are shown in table 4.1. The measurements are done using NVIDIA Nsight 4.2 on Microsoft Visual Studio 2012.

GPU	NVIDIA Geforce GTX 780 with 3GB GDDR5 memory
CPU	Intel Core i7-4790 Quad core running each core at 3.60 GHz
Installed memory (RAM)	32 GB

Table 4.1 – Test system specifications

Table 4.2 shows the average time it takes to produce the output of the key position of each wound phase. This is the average processing time taken from 20 samples (drawing a wound 20 times). It also shows the average processing time of the entire wound healing timeline. As one can see, the fastest wound phase to render is the vasoconstriction phase, which is not surprising, because the only thing the vasoconstriction phase does is show the lower skin texture of the hand mesh (see section 4.3.1). The most expensive wound phase to render is the scab phase. This phase has an average of 175 ms to render. This probably has to do with the fact that this phase outputs the most textures (3) and thus uses more techniques to generate its output.

Wound phase name	**Average processing time (ms)**
Vasoconstriction	2
Vasodilation	105
Scab-forming	126
Scab	175
Maturation	97
All phases	943

Table 4.2 – Generating wound phases processing time

Measurements were also done after the wound healing timeline was rendered, as seen in table 4.3. As mentioned before in section 4.2, the user can scroll through the wound healing timeline with a slider. The measurements in table 4.3 were collected over a timespan of rendering for 60 seconds. As one can see there is not much difference between when the user is actively scrolling and when the user is not.

	Max ms/frame	**Avg. ms/frame**	**Min FPS**	**Avg. FPS**
No scrolling	1.072	1	933	1000
Scrolling	1.467	1	678	1000

Table 4.3 – Frame time measurements

4.5 Conclusion

In this chapter an implementation has been described of a system that can simulate the healing process of an abrasion. It uses a pre-made hand mesh which includes four textures modeling the skin of the hand. The user of the system can draw the desired shape of the wound on the hand mesh after which the simulation generates the entire wound healing timeline of an abrasion using the shape that the user has drawn. The user can use a slider to scroll through the wound healing timeline in real-time. Measurements have shown that creating the entire wound healing timeline takes little time (see table 4.2). Rendering the wound timeline and scrolling through the timeline can be done in real-time (see table 4.3).

Chapter 5

Conclusion

The goal of this thesis is to create a real-time visual simulation of the healing process of an abrasion. A system is presented that is used to implement this visual simulation. The system is able to generate a visual representation of the whole healing process using a collection of procedural generation techniques. The user can scroll through the healing process in real-time. Because the system is real-time, it can be very useful in software that require real-time feedback, such as games or virtual training software for medical personnel. For example, the wound simulation software can be incorporated in medical training software where the system first generates a wound with a random size. The student can then scroll through the wound healing timeline to see how it looks. This can be a cheap alternative to having to show real wounds, which can be hard to come by if one wants to show the students a specific type of wound.

The main advantage of the described system is that it very modular. It is relatively easy to change the visual result of the system, by adding or adjusting key positions. Because of this high modifiability, the system can also be used for other types of wounds, for example burn wounds, or for the simulation of non-wound phenomena, for example for modeling the ground of a tundra that changes over time (grass or moss grows and disappears, etc.).

5.1 Future work

Although our system is the first system capable of simulating the wound healing process, still a lot of open questions remain.

The system described in this thesis uses exclusively textures for the end result, which is a disadvantage. Any special geometric changes are not supported. This means that, for example the effect of a scab slowly detaching itself from a wound cannot be simulated, without major modifications to the system. One adaptation to the system that can improve both flexibility and realism is the addition for geometry generating and modifying techniques. For example, to add support for scabs falling off the wound, a technique could be developed that would create a copy of the scab’s geometry, just before it starts to detach itself from the wound. This new piece of geometry is separate from the original mesh, so it could be bend and shaped to simulate the detaching process of the scab.

When looking at the texture generation techniques, there is always room for more procedural texture generation techniques. For example, more types of noise could be added ([16] and [17]). This gives more freedom to the artist designing the output of the key positions. He will have more tools to create the output he wants the key position to have. Currently the result of the system is highly dependable on the skill of the artist designing the output of the key positions. If the person designing the output of the key positions is not really skilled in creating shaders that produce realistic looking textures, it can highly affect the realism of the output of the system itself. This, of course, depends on the desired art style and is only a problem when realism is the desired effect. A tool for artists can be developed to make the system more accessible to artists that are not skilled in creating shaders.

The hull generation technique (section 4.2.1) also leaves room for improvement. The most obvious of improvements is to add support for concave hulls, and therefore also the support for drawing concave shapes. There is less research done on the calculation of concave hull than there is on the calculation of convex hulls, but there are some promising techniques. Moreira and Santos [18], for example, proposed a k-nearest neighbor approach that can calculate both convex hulls and concave hulls, depending on the k value. Very recently, Rosén, Jansson and Brundin [19] developed a fast concave hull generation algorithm they called the Gift Opening algorithm. According to their report, the concave hull for data sets with over 10^7 points can be calculated within a few seconds on an Intel Core 2 duo PC.

Section 4.2.2 describes a naïve way of creating a binary texture of the convex hull. However, this technique is not very efficient as it checks for every pixel in the bounding box if that pixel lies within the convex hull or not. The efficiency of this technique can easily be improved by making use of a property of the convex hull: A direct line can be drawn between two arbitrary points inside the convex hull, without having that line intersect the edge of the convex hull. This means that if one would take every pixel that lies on the edge of the top half of the convex hull and drew a line straight down to the edge of the lower half of the convex hull, it would create the binary texture of the convex hull, as seen in figure 5.1.

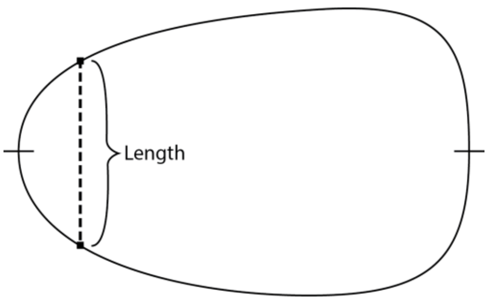

Figure 5.1 – Creating binary texture improvement

Some lighting techniques can also be added to the simulation framework. One technique that could really increase the realism is the addition of Subsurface Scattering or sometimes called Subsurface Light Transport. Section 2.2.4 in the related work chapter already described two of these methods that are able to realistically render skin. The wound simulation as described in chapter 4 already uses multiple skin layers (upper skin texture and lower skin texture), meaning the system can easily be modified to allow skin rendering using Subsurface Scattering. The simulation framework also currently does not feature shadow rendering. Adding shadows to the system can greatly increase the feel for depth and thus increase realism.

The simulation system described in chapter 4 is designed to simulate the healing process of an abrasion wound. This is a shallow, open wound that has minimal bleeding, if it bleeds at all and thus is was a good candidate to simulate. Other wound types can also be simulated using the proposed framework in chapter 3, however. Most of these wounds, like incisions, lacerations and puncture wounds involve a big change in the surface of the skin, meaning a hole or a cut will be present. One could use the height textures to displace the vertices of the mesh inwards to create the hole or cut. This will only generate geometric detail on one axis (the one that is normal to the surface of the mesh), so it might still not be enough. Further research is needed to allow for the simulation of holes or cuts. Bruises can be simulated using the current system, however, because bruises do not involve big visible changes to the surface of the skin. Blood simulation can also be an interesting addition to the system. Blood in a wound flows and eventually clots, which also adds to the whole wound healing process.

Lastly, the reference images shown in section 4.3 that are used to make a visual comparison for each phase are not validated images by doctors or persons by a medical background. These images are picked after making an educated guess that these images represent the corresponding wound phase. Some of the phases may have an incorrect representation of what the phase should look like, because the wrong reference image was used. For future work it would be interesting to develop a corpus of wound healing visual data, in collaboration with medical professionals. Such a corpus could then be validated and used as a basis for more precise simulation.

Bibliography

[1] J. Seevinck and M. Scerbo, “A simulation-based training system for surgical wound debridement,” *Stud. Heal.*, 2005.

[2] W. Stadelmann, A. Digenis, and G. Tobin, “Physiology and healing dynamics of chronic cutaneous wounds,” *Am. J. Surg.*, p. 13, 1998.

[3] A. Vlachos, “Rendering Wounds in Left 4 Dead 2,” in *Game Developers Conference*, 2010, p. 29.

[4] C. Lee, S. Lee, and S. Chin, “Multilayer structural wound synthesis on 3D face,” in *International Conference on Computer Animation and Social Agents*, 2011, no. Casa, p. 44.

[5] P. Hanrahan and W. Krueger, “Reflection from layered surfaces due to subsurface scattering,” *Proc. 20th Annu. Conf. Comput. Graph. Interact. Tech. - SIGGRAPH ’93*, pp. 165–174, 1993.

[6] E. D’Eon and D. Luebke, “Advanced Techniques for Realistic Real-Time Skin Rendering,” in *GPU Gems 3*, Addison-Wesley Professional, 2007, p. 1008.

[7] Wikipedia, “Kernel (Image Processing),” *Wikimedia*, 2014. [Online]. Available: http://en.wikipedia.org/wiki/Kernel_(image_processing). [Accessed: 16-Jan-2015].

[8] Aphrodite3d, “Realistic 3D Male Hand,” *TurboSquid*, 2013. [Online]. Available: http://www.turbosquid.com/3d-models/obj-realistic-male-hand/749416. [Accessed: 16-Jan-2015].

[9] M. De Berg and M. Van Kreveld, *Computational geometry*, 3rd ed. Springer, 2000, pp. 6–7.

[10] Berteun, “Abrasion on hand,” *Wikimedia*, 2005. [Online]. Available: http://commons.wikimedia.org/wiki/File:Abrasion_on_hand_20050906.jpg#file. [Accessed: 16-Jan-2015].

[11] Jpbarrass, “Hand Abrasion - 32 minutes after injury,” *Wikimedia*, 2008. [Online]. Available: http://commons.wikimedia.org/wiki/File:Hand_Abrasion_-_32_minutes_after_injury.JPG. [Accessed: 16-Jan-2015].

[12] Jpbarrass, “Hand Abrasion - 16 hours 45 minutes after injury,” *Wikimedia*, 2008. [Online]. Available: http://commons.wikimedia.org/wiki/File:Hand_Abrasion_-_16_hours_45_minutes_after_injury.JPG. [Accessed: 16-Jan-2015].

[13] S. Tariq, “D3D11 Tessellation,” in *Game Developers Conference*, 2009, p. 38.

[14] ALEF7, “Scab on knee,” *Wikimedia*, 2011. [Online]. Available: http://commons.wikimedia.org/wiki/File:Scab.jpg. [Accessed: 16-Jan-2015].

[15] Jpbarrass, “Hand Abrasion - 21 days 18 hours 21 minutes after injury,” *Wikimedia*, 2008. [Online]. Available: http://commons.wikimedia.org/wiki/File:Hand_Abrasion_-_21_days_18_hours_21_minutes_after_injury.JPG. [Accessed: 16-Jan-2015].

[16] R. Cook and T. DeRose, “Wavelet noise,” *ACM Trans. Graph.*, vol. 1, no. 212, pp. 803–811, 2005.

[17] S. Worley, “A cellular texture basis function,” *Proc. 23rd Annu. Conf. Comput. Graph. Interact. Tech. - SIGGRAPH ’96*, pp. 291–294, 1996.

[18] A. Moreira and M. Santos, “Concave hull: A k-nearest neighbours approach for the computation of the region occupied by a set of points,” *Int. Conf. Comput. Graph. Appl.*, pp. 61–68, 2007.

[19] E. Rosen, E. Jansson, and M. Brundin, “Implementation of a fast and efficient concave hull algorithm,” Uppsala Universitet - Department of Information Technology, 2014.