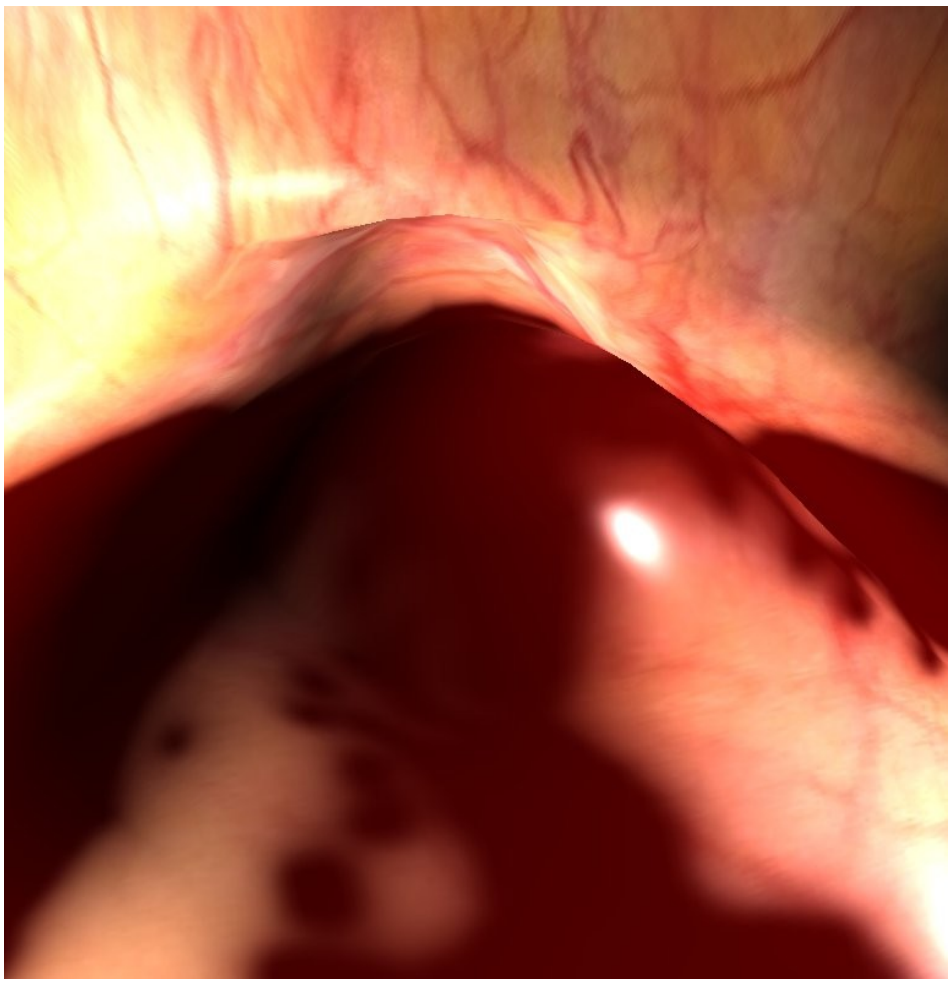
CHALMERS



GPU Based Liquids and Surface Effects

Cauterization and blood flow for surgical simulation

Master of Science Thesis in the Programme MPALG

DANIEL KVICK

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, October 2011

GPU Based Liquids and Surface Effects
Cauterization and blood flow for surgical simulation
Daniel Kvick

Examiner: Ulf Assarsson

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

[Cover: Blood flowing across the surface of a mesh]

Department of Computer Science and Engineering
Göteborg, Sweden October 2011

Abstract

Virtual surgery is a didactic tool used in order to train surgeons without risk to people or animals. To achieve sufficient realism, these simulations require surface effects such as burn marks from cauterization and bleeding from damaged vessels.

We propose efficient methods for generating such surface effects using the Graphics Processing Unit. For each of the two effects, multiple solutions are proposed and discussed. The chosen options are then motivated and explained in detail. Primary focus is placed on simulation of fluid dynamics using particle systems.

In order to simulate cauterization, we propose a GPU-based method using floating-point textures to store temperature and tissue decay. These decay values are used to interpolate between textures which represent different degrees of tissue damage. Efficient approximation of the distance between surfaces and operating instruments is achieved using a three-dimensional distance-field.

Blood flow is simulated on the GPU using a two-dimensional form of Smoothed Particle Hydrodynamics, projected into the texture-space of a mesh. Linked lists are used for efficient representation of hash buckets. We find that our GPU-based version performs significantly better than a CPU-based alternative. In the results section, interactive frame rates are achieved with over 100,000 particles in the system.

Acknowledgments

Writing this thesis has been a serious undertaking and the greatest challenge of my academic career, so far. Though nearly fifty pages were used to summarise the process and results, it feels as if I have barely scratched the surface of what this project has entailed. Naturally, I could never have completed such a task without the help of many kind individuals willing to lend a hand.

First, I would like to thank Ulf Assarsson, without whom I would never have had the opportunity to work on this project. I am especially grateful to him for referring me to Surgical Science and agreeing to serve as examiner for my work there.

I feel the deepest gratitude towards the development team at Surgical Science as a whole, all of which have provided me with constant support and shown more interest in my progress than I could have hoped for. Because of them, I felt part of the team, despite being an intern working on a separate project.

Of course, none of this would have been possible if it were not for Anders Larsson: my supervisor and initial contact at Surgical Science. I am grateful to him for trusting me to work there and arming me with the abundant literature needed for the daunting task of preparing an initial assault on this project.

Many thanks go out to David Löfstrand, who stole countless hours of my time with conversations far removed from what I should have been working on. In retrospect, those chats provided me with the seeds for what seems like a majority of all bright ideas during this project.

I also feel indebted to Göran Wallgren, who helped me keep track of my vectors and never turned down an invitation for a bug hunt (of the software variety), regardless of how busy he might have been.

Finally, I would like to thank Ine Lurquin, without whom I would never have managed to complete this endeavour. Had it not been for her loving support and perceptive proof-reading I would have gone stark raving mad no more than halfway through this venture.

As I finish this thesis, it is my hope that this humble collection of experiences and conclusions may prove useful to others. Though I leave this project behind, I am proud to be a member of the Surgical Science development team and I am looking forward to continuing my work on integrating the developed system into LapSim.

– Daniel *"Agentlien"* Kvick

Table of Contents

1 Introduction

Simulated surgery is an active research topic within computer science [1][2]. Accurate simulations of surgical procedures help surgeons gain familiarity with many tasks and practice many essential skills [3] without endangering people or live animals. In this thesis we present solutions to the problem of creating visually pleasing surface effects for such applications. The focus of this report is specifically on simulation of blood flow and burn marks across tissue surfaces.

1.1 Motivation

Surgical Science is the company behind LapSim, a high-end simulated surgery application for use in the training of surgeons. High visual quality is essential for believability of the simulation and immersion for the user [4]. If the simulation does not look realistic, the willing suspension of belief is broken. This makes it more difficult to take the training seriously, leading in turn to lower performance and a less effective learning experience.

One important visual aspect which has been neglected in the LapSim system is surface effects such as bleeding and burn marks. These effects provide important visual cues of the simulated situation and are essential for the usability of the system. The current solutions are simple vertex-based methods calculated on the Central Processing Unit (CPU). Their simple design is inefficient and the simplistic visuals are often considered awkward and unrealistic by the users. In order to improve visual quality, and thus immersion for the user, new methods were developed to replace the current surface effects.

1.2 Problem statement

In this paper we describe the design of surface effects which have been implemented for future integration into the LapSim system. While physical accuracy was always considered when making design decisions, the primary focus was on performance and visual quality. There is no point investing additional design time and runtime resources to achieve a level of realism which can only be verified by numerical analysis of the simulation data. All methods were implemented on the Graphics Processing Unit (GPU) to allow efficient rendering of pixel-based effects. The goal was to find visually pleasing results which could be efficiently implemented on the GPU.

The two key surface effects are burn marks left by the operating instruments and haemorrhages caused by damage done to tissues or blood vessels. The following subsections describe these effects and provide a basic introduction to our approach when implementing them. All implementations have been made in separate applications and have yet to be integrated into LapSim.

1.2.1 Burn marks

Cauterization is used in order to minimize bleeding when cutting through tissue or severing blood vessels. When cauterizing tissue, the colour gradually changes. This shifting colour provides a visual cue of the state of the tissue. This helps surgeons achieve the desired results without causing excessive damage to the surrounding tissue.

The cauterization simulation was split in two passes. In the first pass, the distance to the instruments is measured using a distance-field (Section 3.3.2). If this distance is close enough, the temperature of the tissue is increased. Otherwise, the temperature is decreased. If the

temperature is high enough, burn level of the tissue is increased. In the second pass, the burn marks are visualized by using the burn level to interpolate between multiple textures representing different tissue states.

1.2.2 Fluid Simulation

During surgery, the slightest mistake can damage tissues and blood vessels. When this damage is severe enough, it often results in bleeding. Once haemorrhaging has occurred, it is important that the surgeon notices quickly and reacts accordingly. Since bleeding is such a ubiquitous phenomenon, the visual quality of these haemorrhages is important.

Many methods of simulating blood flow were evaluated. The final decision was to use a particle-based simulation where particle movements are simulated in two dimensions. This particle plane is then projected across the tissue surfaces. The resulting particle positions are smoothed using a Gaussian blur and rendered with a blood texture.

2 Previous Work

2.1 Cauterization

Cauterization for virtual surgery is such an uncommon and specific application that it was difficult to find related research without first knowing how to solve the problem itself. Inspiration was mainly drawn from the current solution in LapSim and existing ideas for improvement thereof. In order to achieve realistic results, videos of cauterization were studied and the physics behind heat transfer was explored.

2.1.1 Heat Equation

The heat equation was first described by Joseph Fourier in the early 19th century. The idea behind it is that the change in heat depends linearly on the Laplacian (Section 2.3.1) of temperature [5].

$$\frac{\partial u}{\partial t} = \alpha \nabla^2 u$$

In this equation, u stands for temperature, t denotes time and α is a positive constant. The heat equation was originally developed to model transfer of heat within a system. However, it has turned out that this same equation models many different phenomena in physics.

2.2 Fluid Simulation

Many different systems have been created for describing fluid motion. The most accurate known model is the Navier-Stokes equations and was developed in the eighteenth century [6]. In more modern times, fluid dynamics has become a common problem within areas such as scientific simulations [1] and movie productions [7]. Consequently, many different systems have been developed for simulating fluids. These systems vary widely in the underlying method and the most interesting ones are described below.

2.3 Navier-Stokes Equations

Developed independently by Claude-Louis Navier and George Gabriel Stokes, the Navier-Stokes equations describe the motion of fluids by modelling their velocities as a vector field [8]. The Navier-Stokes equations can be derived from Newton's laws.

2.3.1 Mathematical Background

In order to explain the motivation behind the equations, one must first understand the underlying concepts. The most important of these concepts is the Del operator and how it is used to calculate gradient and divergence. The Del operator is a tuple of operators, used to refer to a set of closely related vector operations [9].

$$\nabla = \left(\frac{\partial}{\partial x_1}, \ldots, \frac{\partial}{\partial x_n} \right)$$

One use of this operator is to calculate the gradient of a function. For any real-valued function $f(x_1, \ldots, x_n)$, the gradient of f is written as the product of Del and f.

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n} \right)$$

This results in a vector field describing the greatest directional derivative. Another use of the Del operator is to calculate divergence. Divergence of a vector field is a measurement of the outward flux. This is calculated as the dot product between the Del operator and the vector field.

$$\nabla \cdot F = \frac{\partial f}{\partial x_1} + \ldots + \frac{\partial f}{\partial x_n}$$

A common combination of these uses is the Laplace operator, $\nabla^2 = \nabla \cdot (\nabla f)$ which calculates the divergence of the gradient of *f*.

2.3.2 Assumptions

When using the Navier-Stokes equations, one must choose which assumptions to make about the fluid and its behaviour. A useful set of assumptions is that the fluid is Newtonian and exhibits incompressible flow. That a fluid is Newtonian means that its viscosity is not dependent on the amount of force applied to it. These assumptions simplify the Navier-Stokes equations somewhat [8].

Technically, a fluid exhibiting incompressible flow is not the same as an incompressible fluid. The difference is that incompressible flow only demands constant density on an infinitesimal scale. An incompressible fluid, on the other hand, exhibits constant density within any finite volume of the fluid. For simplicity, however, we will refer to the property of incompressible flow as incompressibility. An equivalent statement to incompressibility is that the divergence of velocity is zero. To understand this, think about the meaning of divergence when it comes to velocity. Divergence of a vector field is equal to its flux. A non-zero flux means the amount of fluid moving into an infinitesimal volume is not equal to the amount of fluid moving out of it. From this, it follows directly that the density of this volume is changing.

The naïve solution to including the above constraints would be to add them as separate formulas in our system of equations. While this is theoretically valid, it is more elegant to reformulate the original equations. When it comes to incompressible Newtonian fluids, the resulting formula is actually simpler than the original. Assuming an incompressible Newtonian fluid, we get the following form for the Navier-Stokes equations [8].

$$\frac{(\partial u)}{(\partial t)} + u \cdot \nabla u = \frac{-(\nabla P)}{\rho} + v \nabla^2 u + f$$

Here, *u* represents velocity, *t* denotes time, *P* stands for pressure, ρ is density and *v* is viscosity. The last term, *f*, represents all other forces acting on the matter which makes up the fluid. The most notable of these forces is gravity.

2.3.3 Motivating the Equations

The above equation describes the motion of incompressible Newtonian fluids. From a first glance, however, this is not obvious. The formula consists of five terms, at least three of which require a bit of effort to understand. Each of these terms represents a separate aspect of the fluid behaviour, making it relatively easy to explain them in isolation.

$$\frac{\partial u}{\partial t}$$

The first part of the formula describes the acceleration of the fluid. This is simply the derivative of velocity with respect to time.

$$u \cdot \nabla u = u \cdot (\nabla u) = (u \cdot \nabla) u$$

The second part describes convection, which in practice means the transfer of velocity to nearby coordinates of the continuum. Convection is described as the dot product of velocity and its gradient. Equivalently, this formula can be read as a scalar multiplication between velocity and its divergence. Seen from this perspective, the motivation is that the divergence of velocity describes the outward flux of the fluid as a rate of change. Multiplying this by the original velocity supplies the correct direction and magnitude.

$$\frac{-(\nabla P)}{\rho}$$

The third term describes the effect of fluid pressure. The gradient of the pressure describes the net pressure as a vector field. Negating this field will give us the resulting force field. As a last step, dividing these forces by fluid densities gives a measurement of how strongly the fluid is affected by the pressure.

$$v \nabla^2 u$$

As *f* is already explained above, the final term to motivate is viscosity. The viscosity of a fluid means its ability to withstand shear forces. On a particle level, these shear forces constitute friction between nearby particles. The viscosity constant *v* is simply a measurement of how viscous the fluid is. The factor $\nabla^2 u = \nabla \cdot (\nabla u)$ is the Laplacian of the velocity field. The gradient of velocity is a vector field describing relative velocities across the continuum. The divergence of this field thus measures the variation in these relative velocities. The greater this variation, the more internal friction will occur.

2.4 Eulerian Grid-Based

Eulerian grid-based methods are similar to the Navier-Stokes equations in that they model the fluid using properties spanning a system. The difference is in that Eulerian Grid-Based methods describe fluids as a grid of properties, rather than a continuum. This simplifies the procedure by treating the fluid within a cell as a single discrete entity. [6]

2.5 Smoothed Particle Hydrodynamics

One of the most popular approaches to fluid dynamics is Smoothed Particle Hydrodynamics (SPH). In this method, fluids are treated as a collection of particles. The behaviour of the fluid as a whole is then determined by the interactions of these individual particles. Every particle is assigned a static mass as well as a number of dynamic properties. These properties are smoothed across nearby particles by the following equation [10].

$$A_I(r) = \int \left(A(r') W(r - r', h) dr' \right)$$

This defines property A using an integral over the whole space. In this formula, r and r' are coordinates indicating particle positions. The kernel function W is mostly taken to be the Gaussian function. In practical applications, the integral is mostly approximated by a summation over all particles.

2.6 Lattice-Boltzmann

An approach similar to both grid-based methods and SPH is the Lattice-Boltzmann method. The simulated space is divided into a lattice over which particles collide and propagate. This allows for direct modelling of how a single particle interacts with its immediate surroundings. It has been shown that the Navier-Stokes equations can be derived from the discrete Lattice-Boltzmann equations, giving a hint towards their ability to achieve statistically viable results [11].

For further reading on the topic of fluid simulation, please see *"Fluid Simulation for Computer Graphics"* by Robert Bridson [12].

2.7 Graphics Hardware

Over the last few years, the layout and inner workings of modern graphics hardware has changed dramatically. The result is that consumer-grade graphics cards now consist of an architecture for highly parallel general-purpose computation capable of both single- and double-precision floating point operations [13][14]. While some parts of the graphics pipeline remain fixed-functionality, the most essential stages can be programmed using specialized c-style languages such as C for Graphics (Cg) and the OpenGL Shading Language (GLSL)[15].

2.7.1 Programmable Stages

The graphics pipeline consists of a series of stages, each with its own purpose and functionality. While some of these stages are fixed functionality, others are programmable. A shader is a piece of software to be executed during one of these programmable stages [13][14] [15]. The word shader comes from their main use: to describe lighting equations. A set of shaders to be executed together is called a shader program. In the most basic programmable architectures, there are two stages: the vertex stage and the fragment stage.

For every vertex which passes through the graphics pipeline, the active vertex shader is executed. This shader is mainly responsible for setting up output position and passing vertex-specific data to the fragment shaders. Once all vertices of a geometric primitive have been processed, the surface area of the primitive has to be rendered. This area is split up into discrete units, called fragments, to be processed independently in parallel by the fragment shader. During this stage, the final output data of the shader program, typically a colour, is calculated [15].

All shader programs must contain vertex and fragment shaders. While both vertex and fragment shaders have a well-defined original purpose within computer graphics, they can in effect be used for general-purpose calculations. The user can control both the expected input and output of each shader, as well as freely programming the handling of this data. On newer graphics hardware, further stages have been introduced. These stages, such as geometry and tessellation, are considered optional [15].

2.7.2 Floating-point Textures

One of the most interesting changes that have taken place within recent years is the ability to store floating-point data in textures. This allows for efficient transfer of large amounts of data to and from shaders. Another advantage is that, by using floating-point textures to store both input and output, multiple passes can operate on the same input, as well as the output of previous passes, without having to send any data over the graphics bus after the first pass.

2.7.3 Atomic Operations

Another invaluable tool is the ability to perform atomic read-write operations from within a shader. Atomic read-write operations provide a way for shader threads executing in parallel to safely use the same memory. A simple technique which beautifully demonstrates the power of this extension was presented by Yang and McKee in the presentation *"Real-Time Order Independent Transparency and Indirect Illumination using Direct3D 11"* [16]. By atomically reading and overwriting data in the framebuffer, a set of threads cooperate in building a linked list of fragment colours to combine for final output colour. Each thread writes a pointer to some unique memory area and writes, in that area, the fragment data and the data they overwrote. In doing so, each thread will effectively write a pointer to the data written by the last thread to touch the same memory. While presented as a way of performing order-independent alpha-blending, this is a general technique and can be used for many different purposes.

Currently, atomic operations are only available on 32-bit integer types [17]. While this does allow for thread-safe manipulation of pointers and counters, it is a serious limitation. There is no efficient way to atomically operate on floating-point values, let alone vectors. While some problems can be solved by atomically updating pointers to vectors, this approach quickly grows complex in both time and memory. Imagine that a series of operations need to be performed by different threads on the same vector. One could create a stack of operations per vector, and then execute these in a second pass. However, if the modified vectors are used in operations on other vectors, a dependency graph needs to be built. Then, each layer of this dependency hierarchy would require a separate rendering pass. It is easy to see that such a scheme quickly grows beyond affordable complexities.

3 Analysis

This section details the theory of the problems, the method followed and the alternative solutions considered. In presenting the alternatives, our design choices during the development process are identified and motivated.

3.1 Method

3.1.1 Alternatives

The main challenge of the project was to find and evaluate potential solutions for each problem. There are many viable approaches, each with a unique set of advantages and disadvantages. In order to ensure a sound design of the system as a whole, multiple options must be considered and evaluated when designing each component. These alternatives were mainly taken from research papers, which often focus on certain parts of the presented solution while leaving others underspecified. This requires another level of investigation to determine how to implement the underspecified components. Furthermore, the specific context of a project sometimes necessitates alternative solutions or variations on chosen techniques. Specifically, the setting of a GPU implementation often requires alternative solutions to problems. As a result, this report covers a wide array of techniques and variations thereof which were evaluated during the course of the project.

3.1.2 Quick Tests

Implementing a new technique in a serious application requires a lot of work with integration of components and overall design. These aspects, while important, make it very difficult to learn and familiarise oneself with new technology or algorithms; the restrictions of interoperability often cause issues which draw focus away from the new technique itself.

In order to alleviate these issues, most new techniques were first tested using what we call quick tests. The idea is that one does not immediately try to apply the technique to the actual problem for which it was chosen. Instead, the focus is on a dummy problem which is easier to solve. A quick test can then be used to learn and verify ones understanding of the technique at hand. One important aspect of quick tests is that they are fast and simple to implement. Consequently, their design is often naïve and unfit for serious applications. Once a quick test is completed, it is thus important that it is removed from the system rather than used as a base for further implementations.

Whenever possible, algorithms and technologies were investigated using quick tests before integration into the main application was attempted. For example, before implementing the linked-list particle hash function (section 3.7.2), the previously implemented hash function (section 3.7.1) was rewritten using image loads instead of texture lookups and with an image store replacing the final output of the fragment shader. This provided a simple application of the image extension, without having to worry about linked lists or atomic operations.

3.1.3 CPU – GPU Transitions

While the parallel nature of the GPU is the main source of its power, it also complicates implementations. Implementations for the CPU may be significantly slower, but they are also easier to formulate and test. Thus, certain algorithms were first implemented on the CPU and later transferred to the GPU. This has an additional advantage when implementing multi-pass algorithms. With a working CPU implementation available, one can transfer the algorithm one pass at a time to the GPU. This prevents having to write the entire code at once and

makes it easy to verify the correctness of each step in isolation. This was particularly helpful when implementing the GPU-based particle simulation.

3.1.4 Results

An important aspect of the methodology is how to measure and present results. Implemented features are simple to describe in terms of their motivation and practical implications. Describing performance results, however, is a more involved process. First, performance indicators have to be identified. Then, tests must be designed in order to measure these indicators. Finally, the gathered results need to be presented in an intuitive manner.

In the context of real-time rendering, the most interesting performance indicator is frame rate. Unlike theoretical measurements, such as big-O notation, the frame rate tells exactly how an implementation behaves in practice. The drawback of using frame rate is that it is highly dependent on the used hardware. Even graphics cards with similar architecture may produce drastically different behaviour due to varying number of processing elements and cache sizes. Consequently, it is not viable to directly compare results attained on different hardware. Preferably, one would still want to supply performance data over a wide range of hardware. Doing so would help indicate the performance of the method itself, rather than that of the system used. For practical reasons, however, all performance measurements for this report were generated on the same hardware (Section 4.2.1).

When performing a test, frame rates are continuously tracked by tallying the amount of frames per second (FPS). The average of these frame rates is then calculated and saved into a performance log. The tallying of FPS is done using a simple frame counter and a time variable. At the beginning of each frame, the frame rate is incremented. Next, the current clock time is queried and compared to the value stored in the time variable. If the difference between these is larger or equal to a second, the frame rate is saved, the frame counter is reset and the time variable is set to the current clock time.

3.2 Tools and Languages

The main focus of this project was on practical implementations. As such, the choice of development tools was central to the success of the project. Many alternatives were investigated and the ones deemed most interesting are described below. This includes alternatives used in the end, as well as those which showed promise, but were ultimately not chosen.

3.2.1 C++

With the ability to mix high-level object-orientation with highly optimised low-level code [18], C++ is an invaluable tool for graphics programming. As such, Surgical Science uses C++ to develop their products. For this reason, other programming languages were not considered for the main platform. All CPU-based implementations described in this report were written in C++.

3.2.2 OpenGL

The Open Graphics Library (OpenGL) is a cross-compatible framework for programming graphics. It is currently developed by the Khronos Group [19]. The entire specification of OpenGL is free and open to the public. Using OpenGL and its extensions provides access to most features available on modern graphics cards and a large set of utility functions to facilitate their use. All CPU-based implementations described in this report were written using OpenGL.

3.2.3 GLSL
The OpenGL Shading Language (GLSL) is a language for programming shaders. GLSL is developed by the Khronos Group and is designed in tandem with the OpenGL functionality for handling shaders [19]. This simplifies interoperability and provides a common language for describing the interface between shaders and the graphics library. Most GPU-based code implemented for this report was written in GLSL.

3.2.4 CUDA
CUDA is a proprietary language for programming on the GPU [13][14]. It requires the use of an Nvidia card with CUDA capabilities. The original intention behind CUDA was to allow researchers and developers to exploit the parallel processing power of the GPU without having to learn how to use a graphics library. It simplifies GPU computing by removing the need to disguise all data processing as graphics operations. Due to its powerful design and recent popularity, CUDA programming was explored as an alternative to using traditional shading languages such as GLSL. However, it was concluded that this led to little advantage when used in graphics applications. Interoperability between CUDA and OpenGL was found to incur serious overheads. For this reason, CUDA was only used for some of the initial quick tests.

3.2.5 PhysX
Nvidia PhysX is a proprietary library for performing physics calculations on the GPU and requires the use of an Nvidia card with CUDA capabilities [20]. PhysX allows for GPU simulation of physics on rigid bodies, cloths and fluids. In these simulations, user-defined data can be bound to physics objects and collision resolution can be handled on the GPU, in software or a combination of both. PhysX was considered as a base for the blood flow simulation. However, implementing particle physics ourselves meant increased adaptability and full access to the implementation details. One specific constraint of PhysX which was deemed undesirable was the inability to simulate interactions between different fluids. Consequently, we did not use PhysX in the final implementation.

3.3 Cauterization
The creation of burn marks was the first of the two main tasks. The idea was to make a system for visualizing the state of tissue which has been cauterized by the surgeon. While visual quality was a priority, we also wanted to approximate the mechanism of cauterizing as realistically as possible.

3.3.1 Current Solution
In LapSim, the visual geometry contains a low-resolution skeleton mesh. This skeleton is called the dynamic mesh. All collisions are calculated against the edges of the dynamic mesh, rather than those of the visual geometry. Each vertex in the dynamic mesh is bound with weighted edges to the nearby vertices of the visual geometry. In this way, each skeleton edge has an indirect weighted connection to the nearby vertices of the visual mesh. When a hot instrument collides with an edge of the dynamic mesh, the heat of the instrument is scaled using these weights and distributed to nearby vertices. This heat is also scaled by the size of the time step and the resulting value is added to a burn level which is kept for each vertex of the visual geometry. When drawing the geometry, this level is used as colour per vertex and interpolated across the triangles. The resulting visuals are shown in Figure 1.

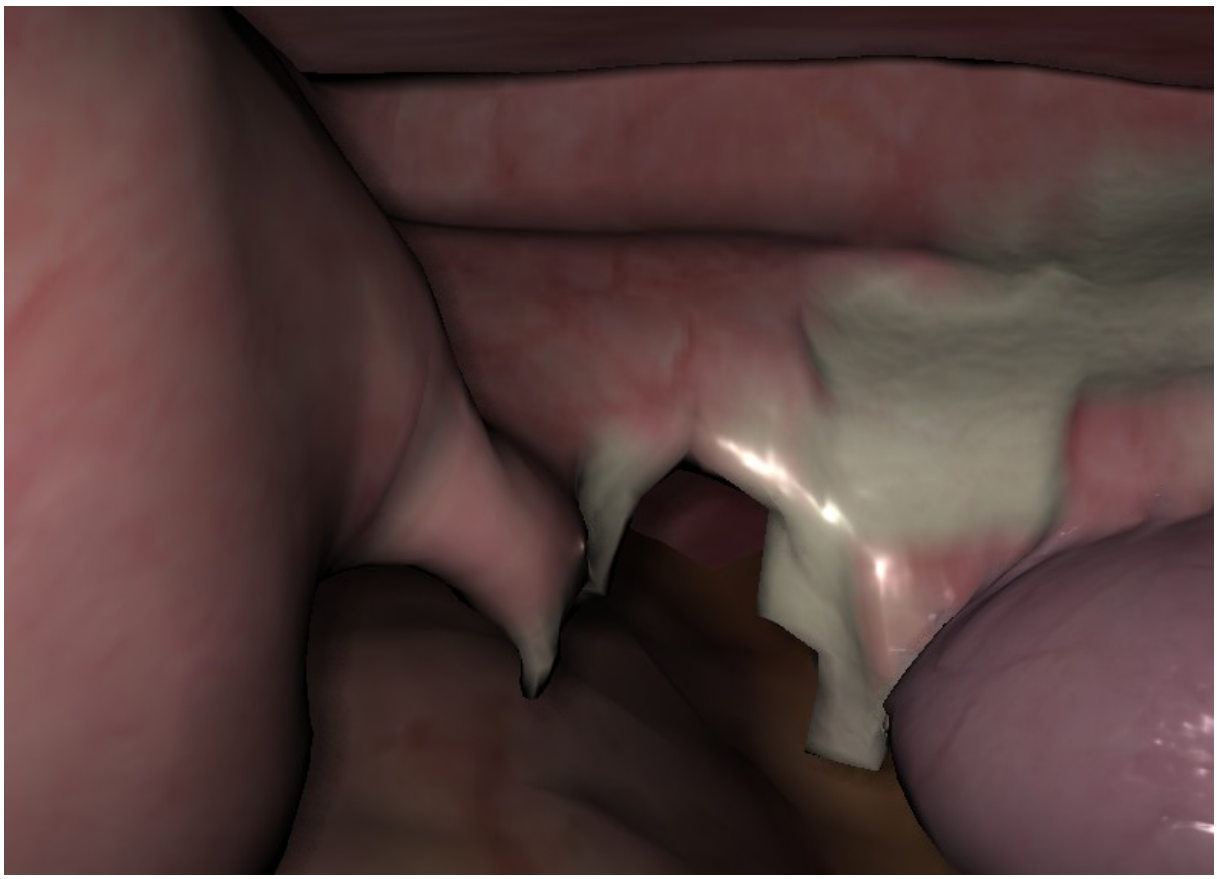


Figure 1: Burn marks (grey) in LapSim across the surface of a fallopian tube.

There are multiple issues with this technique. Firstly, it acts on vertices, which means it is only as detailed as the mesh itself. Secondly, it is implemented on the CPU, which makes it expensive to perform any advanced calculations per pixel. Probably the worst issue, however, is the reliance on the dynamic mesh. Since cauterization is done against the skeleton inside the geometry, burning will mostly leave the same mark on both sides of the surface, regardless of how thick the tissue is supposed to be. In order to alleviate these issues, a better solution was sought.

3.3.2 Distance Field

A distance field is a texture where each texel indicates the distance to the nearest point on a surface [21]. In LapSim, distance fields are three-dimensional textures, centred on the origin of an instrument, and approximate the distance to the surface of that instrument. In instruments with moving parts, each part has its own distance field. The values in a distance field are positive for points outside of the instrument and negative inside the instrument. Given a point in world space, the distance to an instrument can be found in three steps: transform the point using the inverse matrix of the instrument, scale the coordinates to texture space and look up the value at the resulting texture coordinate of the distance field. Points outside of the distance field are first projected onto the surface of the distance field. The resulting value is then set to the sum of the read value and the distance between the original point and its projection. Such an approximation is efficient and generates results of sufficient precision for the actual applications of these distance fields.

Currently, distance fields in LapSim are used to calculate deformations of dynamic meshes. When a vertex of a deformable mesh ends up inside an instrument, the instrument's distance field is used to find how far the instrument has intruded into the mesh. Ray marching is then

performed over the distance field to approximate the nearest point on the instrument surface in the direction of the intrusion. The result of this ray marching is used to translate the vertex, thus deforming the mesh with the instrument.

In our solutions, distance fields are used to calculate, per pixel on a surface, the distance to a hot instrument. When this distance is small enough, cauterization occurs. At first, the distance was used to scale the severity of the burn; however, a binary method which simply burns or not was found to work better in practice.

3.3.3 Burn level

During the cauterization of tissue, the colour changes continuously as the tissue degrades. When a surgeon begins to cauterize tissue, the tissue begins to whiten. Once the burn is severe enough, the tissue rapidly turns black. This level of tissue decay is stored in a floating point texture where values, called burn levels, represent the severity of the burn across a tissue surface. By choosing the resolution of this burn level texture, one can effectively determine the resolution of the cauterization algorithm across that surface. Using this burn level, one can then visualize the burn by interpolating between textures which represent different stages of the cauterization process. In our solution, three tissues were used: one representing healthy tissue, a whitened texture representing mild burns and a dark texture representing badly damaged tissue.

Since surface points are mapped to texture coordinates in the cauterization algorithm, it is vital that texture coordinates are unique. If two distinct surface points map to the same texel, burning around either will affect both. Consequently, meshes which do use tessellating textures or repeated coordinates must be assigned a separate set of texture coordinates for use in the cauterization algorithm. Alternatively, one can make the artist responsible for avoiding such conflicts in texture mapping and use the same texture coordinates for both skinning and cauterizing.

3.3.4 Temperature

In reality, the instruments used for cauterizing do not directly damage the tissue. Instead, they induce a current which increases the temperature of the tissue. Sufficiently high temperatures, in turn, cause damage. This has implications which affect the way cauterization works in practice. For one, it takes a while before the temperature is high enough to cause burn marks. Additionally, the rate of decay increases throughout the duration of the cauterization. Finally, temperatures spread across tissue surfaces.

Temperature can be represented by a second floating point texture. Cauterizing is then modified so that it increases temperature, rather than directly affect the burn level. At high enough temperatures, burn level is increased in a way which depends linearly on the temperature. The realism of this heat model can be improved further by smoothing heat values across neighbouring texels. Doing so gives the burn wounds a smooth look, rather than a sharp edge. It also leads to temperatures spreading across the surface of a mesh.

3.3.5 Smoothing Issues

Most calculations described so far are done independently per pixel and require only the uniqueness of texture coordinates across the surface of a mesh. In fact, only the spreading of heat via smoothing requires knowledge of surrounding pixels. Thus, the algorithm is entirely insensitive to discontinuities or seams in texturing when smoothing is disabled. Since smoothing is done using a straightforward texture lookup, heat transfer across texture seams

gives rise to artefacts such as discontinuities in the burn marks. Possible solutions to these issues are discussed in section 6.1.2.

3.4 Fluid Simulation

Our largest task was to develop a new method for fluid simulation which could create detailed and realistic results without incurring significant costs in time and memory. As the current solution is CPU-based and LapSim itself is CPU-bound, it was deemed affordable to run a more complex simulation on the GPU.

3.4.1 Current Solution

At present in LapSim, bodily fluids are simulated per vertex of the visual geometry. When the amount of fluid in a vertex exceeds a certain threshold, part is transferred to some neighbouring vertices. To ensure that fluid does not flow against gravity, it is only transferred along an edge if the dot product of this direction and the direction of gravity is non-negative. When drawing, these fluid values are used to set a colour per pixel which is then interpolated across the surfaces. Vertices with fluid are given an alpha value of 1 while others are given an alpha of 0. This way, fluid colours are interpolated smoothly across triangles.

Figure 2: Fluids in LapSim. Left: Blood covering the liver and bile leaking onto the gall bladder. Right: Blood running across a severely damaged fallopian tube.

Unfortunately, this method still suffers from issues related to its coarse nature. In detailed areas, the small triangles effectively cancel out the interpolation between alpha values, as it is done over such a short distance. This gives rise to sharp edges. Surfaces composed of larger triangles also suffer from visual artefacts: the blood forms large polyhedra with visibly straight edges. As blood flows from one vertex to another, these edges appear to jump forward in discrete steps.

3.4.2 Naïve Solutions

Our first attempt to simulate blood across the surface of a mesh was a simple pixel-based method relying entirely on dot products and other vector operations on the orientation of surfaces and the amount of blood present at neighbouring pixels. After a few quick tests, it was determined that such methods easily produced visually pleasing results, but suffered from significant issues. Most evident was the numerical instability as the amount of blood tended to either diverge towards infinity or converge to nothing. Furthermore, the models themselves were arbitrary and lacking any serious motivation. Finally, this type of solution provided little room for future improvements. It was thus decided that a more advanced and well-motivated approach was necessary. Our attention then turned towards the field of fluid dynamics.

3.4.3 Fluid Dynamics

Currently, the most accurate model of fluid dynamics available is the Navier-Stokes equations. These equations treat the fluid as a continuum, and can be used to find the velocity and pressure of any point inside the fluid. However, solving these equations directly is not efficient enough for real-time simulations. Another obstacle when using the Navier-Stokes equations is that they are not well-understood. In fact, the Clay Mathematics Institute is currently offering a million U.S. dollars to anyone who can prove whether or not smooth solutions always exist that satisfy these equations [22].

There are multiple other Eulerian methods, in particular grid-based variants; however, they typically suffer from complexities beyond what is practical for real-time simulation. Eventually, it was determined that the best approach to simulating fluids would be to use some form of a particle system.

3.4.4 Smoothed Particle Hydrodynamics

One of the best known particle-based approaches to simulating fluids is Smoothed Particle Hydrodynamics (SPH). With recent developments in computational hardware, it has become popular for use in games. In fact, PhysX uses a form of SPH to simulate its particle systems [20]. We thus considered the possibility of using such a proprietary physics library for fluid simulation. Ultimately, we decided that the best choice would be to implement a particle system ourselves.

The basic idea behind SPH is that particle properties depend on the properties of surrounding particles. Each particle property A is determined by the following formula [10].

$$A_s(r) = \sum_b m_b \frac{A_b}{p_b} W(r - r_b, h)$$

In this equation, each particle b has position r_b, mass m_b, pressure p_b and a value A_b of property A. The value h represents the interaction radius of a particle. Using this formula, one can substitute W for any desirable kernel function, making it easy to adapt and extend a chosen implementation. Since the actual simulation of a particle system is done in terms of property changes over time, it is important that these kernel functions are differentiable. The most commonly used kernel functions are the Gaussian and cubic splines.

There are a number of favourable properties to SPH. Probably the most interesting is a guaranteed conservation of mass, which follows directly from the use of distinct particles with invariable mass. Many implementations also use constraints to ensure incompressibility of the fluid. However, these constraints require complex calculations which were deemed too

computationally expensive for our simulation. However, the basic approach was still seen as the best alternative for real-time fluid simulation.

3.5 Viscoelastic Fluid

While incompressible SPH is proven to behave well, it requires solving complex, and thus expensive, equations per particle. The approach which was eventually chosen for particle simulation is based on the article *Particle-based Viscoelastic Fluid Simulation* by Clavet et al. [23] and is another variation on SPH. In this method, density is calculated differently and the incompressibility constraint is approximated, rather than enforced. The algorithm proposed in the article consists of nine steps.

1. Hashing of particles
2. Application of gravity
3. Application of viscosity
4. Application of velocity
5. Adjustment of springs
6. Application of spring displacements
7. Double density relaxation
8. Resolution of collisions
9. Updating of velocity

Steps five and six handle virtual springs between particles; these springs have a rest length which changes with time towards their current length. Together, these two steps help simulate the elasticity and plasticity of certain fluids. After experimenting with different variations on the algorithm, it was determined that these effects were not desirable for the application of interest. Consequently, these two steps were discarded in order to improve performance.
We also decided that the step for resolving collisions was not relevant, as our implementation projects the particles onto geometry surfaces (Section 3.8).

3.5.1 Hashing of Particles

Several steps of the algorithm require smoothing of properties from neighbouring particles. Theoretically, these properties should be smoothed over all particles in the system; a procedure which implies at least an $O(n^2)$ complexity. This is clearly too costly as particle systems often contain tens of thousands or hundreds of thousands of individual particles. Furthermore, the influence two particles exert on each other depends on the distance between the two. This means that distant particles have little impact on each other in practice and may be ignored without noticeable effect. In order to make use of this observation, we want an efficient method for identifying the neighbours of a particle without having to loop over all other particles. This is solved via hashing of particle positions. Using a spatial hashing algorithm, the space of the particle system is divided into a grid of hash buckets containing particles.

3.5.2 Simple Particle Steps

Application of gravity, application of velocity and velocity update are straightforward steps performing simple arithmetic operations per particle. Gravity updates particle velocities according to $\Delta\vec{v}=\vec{g}\,\Delta T$ with g the gravitational vector and T representing time. Similarly, velocity updates positions according to $\Delta p=\vec{v}\,\Delta T$. The final step of the algorithm updates particle velocities to reflect the actual particle movements during the frame. In the paper on viscoelastic fluids, this was done according to

$$\vec{v}=(\ \vec{P}_{new}-\vec{P}_{old}\ )\,\Delta T$$

where P_{old} is the position at the start of the iteration and P_{new} is the resulting position at the end of the iteration. However, in order to model traction against the surface on which particles are simulated, an inertia term I is introduced. As a final improvement, the magnitude of each component is clamped to a maximum value v_m in order to ensure reasonable particle velocities. With these modifications in place, the following equations model the velocity update for each component i.

$$\Delta P_i=P_{new_i}-P_{old_i}$$

$$v_i=\begin{cases} v_m, & \Delta P_i>v_m \\ -v_m, & \Delta P_i<-v_m \\ I\,\Delta P_i, & -v_m\le\Delta P_i\le v_m \end{cases}$$

3.5.3 Application of Viscosity

Viscosity is the first effect applied which uses the particle hashing to efficiently find neighbouring pairs. For any two particles within interaction distance of each other, the inward radial velocity is calculated according to $u=(v_i-v_j)\cdot\hat{r}_{ij}$ where v_i is the velocity of particle i and $\hat{r}_{ij}$ is the normalized vector from i to j. If this value is above zero, the viscosity effect is applied as a pair of impulses between the two particles.

First, the impulse $I=\Delta t(1-q)(\sigma u+\beta u^2)\hat{r}_{ij}$ is calculated where q is the distance between i and j. The constants σ and β are used to scale linear and quadratic terms respectively. The impulse applied on particle j is $I / 2$ and similarly, the impulse $-I / 2$ is applied on particle i.

3.5.4 Double Density Relaxation

Incompressibility is approximated by calculating the fluid density around each particle and performing displacements which work towards achieving the rest density. For each particle, the pressure acting on it is calculated by looping over its neighbours and keeping a running density sum. The density of a particle depends on the number of and distance to its neighbours.

$$P_i = k\left(-\rho_0 + \sum_{j \in N(i)} \left(1 - \frac{r_{ij}}{h}\right)^2\right)$$

Here, ρ_0 stands for rest density and r_{ij} is the distance between particles i and j. The set $N(i)$ is the set of all neighbours to particle i. The pressure constant k is used to scale the measured density. This step relies on the assumption that all particles have equal mass. Next, the calculated pressure is used to apply a displacement on each neighbour. The magnitude of this displacement is proportional to the distance between the particle and its neighbour.

$$D_{ij} = \Delta T^2 P_i \left(1 - \frac{r_{ij}}{h}\right) \hat{r}_{ij}$$

In order to avoid clustering of particles, a second pressure term, near-pressure, is introduced. The near-pressure acts as an exclusively repulsive force which ensures that the particles of low-viscosity fluids do not form tightly packed clumps by pulling strongly on a small set of neighbours.

$$P_i^{near} = k^{near} \sum_{j \in N(i)} \left(1 - \frac{r_{ij}}{h}\right)^3$$

The near-pressure constant k^{near} is analogous to k for pressure. The following algorithm for calculating pressures and applying the resulting displacements is used in the original paper [23].

```
1.for each particle i
2.  ρ := 0
3.  ρ^near := 0
4.  // compute density and near-density
5.  foreach particle j in neighbors(i)
6.    q := ρ_ij/h
7.    if q < 1
8.      ρ := r+(1-q)^2
9.      ρ^near := ρ^near +(1-q)^3
10. // compute pressure and near-pressure
11. P := k(ρ-ρ_0)
12. P^near := k^near ρ^near
13. dx := 0
14. for each particle j in neighbors(i)
15.   q := r_ij/h
16.   if q < 1
17.     // apply displacements
18.     D := Dt^2(P(1-q)+P^near(1-q)^2) r̂_ij
19.     x_j := x_j+D/2
20.     dx := dx-D/2
21.   x_i := x_i+dx
```

An important thing to note is that both double density relaxation and the viscosity step apply correctional impulses directly on particles while still looping over them. This gives different results on the CPU and GPU. On the CPU, the behaviour of this method is difficult to predict precisely as it depends on the order in which the particles are processed. When calculating on the GPU, this becomes even more complex. Not only does it depend on the order in which particles are processed; it also depends on which particles are processed in parallel. This means the results may vary between GPU models with different numbers of processing units or different scheduling of threads. In fact, the scheduling of GPU threads may even differ between different runs on the same machine. However, the original algorithm is itself a results-oriented approximation and our GPU implementation still generates visually pleasing results which seem numerically stable. Accordingly, while it is important to be aware of these differences, they pose no real issue.

3.6 GPU Implementation

After choosing the underlying model, the next step was to implement this on the GPU. The article on viscoelastic fluid simulation was very practical and made it easy to understand how to implement it as intended. Still, adapting this algorithm to the GPU carries a unique set of challenges. The main difficulties concern data representation, communications between processing units and parallel execution of particle threads. The result was an implementation consisting of one GLSL shader program per pass and a set of functions in C++ for handling and executing these shaders.

3.6.1 Workflow

At first, the viscoelastic fluid simulation was implemented on the CPU. This was a straightforward and relatively simple process. Once this implementation was done, a final version was produced by successively replacing each step in the algorithm with a GPU implementation. In doing so, the correctness of each step and the potential differences in behaviour between the implementations could be analysed one pass at a time. Such an approach provided a significant advantage over trying to implement the entire algorithm directly on the GPU; whenever a problem occurred, it was immediately evident in which pass the problem originated. Integration issues were largely avoided as each component was integrated into a working whole during its development, as opposed to having to integrate all parts with each other simultaneously.

3.6.2 Data Representation

One of the key problems when implementing the algorithm on the GPU was determining how to represent and store all necessary data. The primary concern was data transfers between CPU and GPU. During the GPU passes, all data needs to be present in graphics memory. As this is a large amount of data, it would be advantageous to keep it local to the GPU, avoiding unnecessary data traffic. Moreover, one would like to store particle data so that the properties of a particle are accessible from within a shader, given only the particle ID.

These concerns were addressed by saving all particle-specific data in floating point textures. Due to hardware and software limitations, these textures need to be two-dimensional in order to support large amounts of particles. Particle IDs are then mapped to two-dimensional texture coordinates as a vector, where I is the particle ID and W is the width of the texture.

Said ID is a simple unsigned integer in the range $[0, P - 1]$ where P is the number of particles in the system. This way of assigning IDs facilitates efficient storage and access of particle properties.

3.6.3 Vertex Shader

After the first hashing step, each of the remaining steps in the algorithm consists of a loop over the set of all particles. All such steps are implemented as single graphics passes using the same vertex shader. This shader is a simple one which, given the particle ID, looks up the corresponding texture coordinates and sets this as output position for the vertex. These IDs and texture coordinates are then passed to the fragment shader. As a result of all particle data being stored as textures in graphics memory, the only particle-specific input to the vertex shader is the particle ID. The list of particle IDs is also stored on the graphics card, meaning no particle-specific data has to be sent from the CPU to the GPU. Therefore, only uniform variables such as algorithm parameters, number of particles, screen size and transformation matrices have to be sent over the graphics bus.

This uniform treatment of algorithm steps means that the only differences between passes, from the perspective of the CPU, is which fragment shader is in use and which texture is set as render target. Hence, a function was written which, given this information, binds all textures and calls the appropriate shader. On the CPU-side of the implementation, each step is then written as a single call to this function.

3.6.4 Neighbourhood Search

Graphics passes such as viscosity and double density relaxation require iterating over all neighbouring particles. Given a particle's hash position, it is guaranteed that all neighbouring particles are found in the nine surrounding hash buckets. For this to work, texture coordinates must be treated as being modulo the texture dimensions. In OpenGL, this is done simply by setting texture wrap mode for each dimension of the texture to GL_REPEAT. A particle's hash bucket and its eight neighbouring buckets can then be found at texture coordinates $[x_0+x_i,y_0+y_i]$ where the particle itself has hash position $[x_0,y_0]$ and x_i and y_i are integers in the range $[-1,1]$. It is thus trivial to enumerate these buckets using a double loop over x_i and y_i.

3.6.5 Particle Creation

Particle creation is handled by the CPU, as this is where the logic resides which determines when and where to spawn new particles. During the development, a shader was written for spawning particles. The idea was to increase performance by minimizing CPU involvement and data traffic over the graphics bus. Testing revealed that this method actually caused an increase, rather than decrease, of computation time. The reason for this was that in spawning particles on the GPU, particle data had to be generated on the CPU and sent as vertex inputs to the graphics memory; the shaders then transferred the given data to the target textures. On the other hand, the CPU-based method simply transfers data directly to texture memory. As such, both methods require the same CPU computation and bus transfer, but the GPU implementation required an additional rendering pass. With these conclusions reached, the shader was removed, leaving the spawning of particles to the CPU.

3.6.6 Differences between Implementations

Adapting the particle simulation to the GPU leads to a number of differences in the underlying algorithm. These differences are mainly caused by the parallel nature of the GPU as well as lacking support for atomic read-write operations on vector types. Of special interest are the viscosity and double density relaxation steps. During these steps, the original implementation loops sequentially over all neighbours for each particle. On graphics hardware, this will be executed in parallel for each particle. Additionally, the target variables are read from one texture and the updated values are written to a separate buffer. This means that the behaviour will differ somewhat from the original CPU implementation, where

positions are updated during the loop and these new values are possibly read back during a later iteration.

The largest difference, however, is in the symmetry of applied impulses and displacements. In the original paper, all impulses and displacements are immediately applied on neighbouring particles, with the sum of these applied to the particle itself at the bottom of the outer loop. Due to limitations in current hardware, this cannot be efficiently achieved on the graphics card. The reason for this is the lack of an atomic read-write operation on vectors. In order to compensate for this discrepancy, the GPU version removes the constant 0.5 from the viscosity and double density equations. While, technically, this is not equivalent to the original formula, it does in practice lead to a more realistic result.

3.7 Hashing

While the article on viscoelastic fluids [23] did mention the need for particle hashing, it offered no insights on how to perform this hashing. Usually, this choice is limited to selecting an appropriate hash function. However, if hashing is to be done on the GPU, the more important question is how to perform this function and store its result in an efficient thread-safe manner.

A simple hashing procedure which worked well in practice was to index each bucket by an n-dimensional array corresponding to the position ranges of its contained particles. Given a particle, each component X_i of its position is hashed according to floor(X_i) mod D where D equals the maximum interaction distance between two particles. This vector is then used as index in order to access the hash bucket.

Once a hash function was chosen, the next task was to find a method for performing the hashing on the GPU. This is in fact an active research topic in which several interesting articles have been written over the last few years. Several methods were investigated [1][24] [25] and two of these were implemented for further evaluation. One option which was not evaluated was to utilize CUDA for hashing via scattered writes. This decision was based on earlier experiments in which using CUDA would severely damage performance (Section 3.2.4).

3.7.1 Spatial Binning

The first explored method of hashing particles on the GPU was inspired by *Efficient Spatial Binning on the GPU* [24]. Using this method, the hash grid is split into a set of *N* layers, each represented by a texture. The idea is that a bucket is represented by one texture coordinate across multiple textures. A separate texture tracks the size of each hash bucket. Using a bucket index as texture coordinates; the ID of the *n*th particle in the bucket can be identified by reading from the texture representing the *n*th layer of the hash grid.

In order to generate this set of textures, particle IDs are rendered in *N* passes. During the *n*th pass, the *n*th layer texture is bound as rendering target. During all but the first pass, the *n-1*th texture is bound as an input texture. Using the particle hash as output position, the particle ID is written into the correct bucket. The depth of a pixel is set to the written particle ID over the number of particles. Thus, with depth testing set to pass the lowest value, the particle with the lowest ID is written for each texel.

In the fragment shader, the particle ID is compared to the one written at the same position in the previous layer. If the previously stored value is not smaller, the fragment is discarded.

This effectively sorts the particles in a bucket, ensuring that different particles are written to each layer. Stencil buffering is used in order to avoid writing to a bucket once all its particles are rendered. When a particle is stored in a layer, the iteration number is written to the stencil buffer. Stencil testing is set to pass particles where the stencil value is equal to the number of the previous iteration. Originally, all stencil values are set to zero, which causes all particles to pass the stencil test during the first iteration. If the stencil buffer was not used, the layer after the last particle of a bucket would be empty, causing the following layer to render the first particle again. An alternative method to this stencil test is to clear the texels of each layer to a value larger than any particle ID. This way, whenever no particle is written, the following pass will discard all particles of that bucket due to the comparison of the particle IDs and this higher value.

One drawback with this technique is the memory usage. With a maximum bucket size of N, a total of N hash textures are required. There is also the possibility of overflow if N is chosen to be too small. In fact, we found that hash buckets tended to contain very few particles on average, with a few buckets containing a very large number of particles. This means that, even with a conservative bucket size, a majority of the texture memory allocated will mostly stay unused. Additionally, this method requires a total of N rendering passes, making the time complexity of this method questionable.

3.7.2 Linked List Hashing

Another method for hashing the particles was devised in order to address these issues. By using a linked list format, memory use was minimized and the hash buckets were rendered in a single pass. The idea was inspired by *"Real-Time Order Independent Transparency and Indirect Illumination using Direct3D 11"* [16] in which a linked list of fragments is used to achieve order-independent transparency. Creating the linked list requires atomic read-write operations on textures. For this purpose, the OpenGL extension *EXT_shader_image_load_store* [17] was used.

In the linked list method, two textures are used. The head texture stores the head pointer of each linked list while the node texture contains all nodes of the linked lists. In the original paper, each node consists of two pieces of information: the fragment data and the next pointer. A global counter is kept in order to keep track of the number of nodes written so far and is used to select memory locations for each node. In our application, this scheme was simplified significantly. We know beforehand how many particles are to be written and can use this information to allocate a node texture of appropriate size. By using particle IDs as texture indices, there is no need to separate data and next pointers. The head pointer is itself the data to identify the first particle in each bucket. Similarly, the next pointers in the node texture uniquely identify both a particle in its bucket and the memory location of the following node. Finally, this use of IDs as pointers eliminates the need for a fragment counter as each particle has already been assigned its memory location implicitly. As a result, our data structure becomes very compact and efficient in both time and space. One single texture lookup is used to both dereference the next pointer and read the next particle ID.

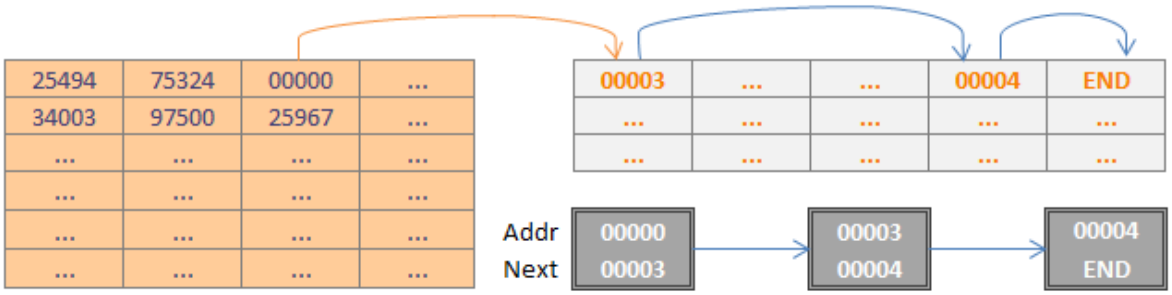


Figure 3: Linked List Hashing. A texel of the head buffer (left) points to the first of three texels in the node buffer (top right) constituting a linked list. This linked list is visualized at the bottom right. Note that the address of an element is itself the data stored in that node. The terminator END can be any value greater or equal to the number of particles.

While this technique is simple to implement and offers less complexity in both time and memory, it comes with another type of cost: so far, the necessary OpenGL extension is only usable on the latest generations of graphics cards. This means that linked list hashing is unavailable to all users with graphics hardware which is slightly dated or from a lower price class. As this is still a large portion of potential users, spatial binning should be kept as a fall-back for machines on which linked list hashing cannot be used.

3.8 Surface Projection

The main goal of this particle system was to simulate fluid running across the surfaces of geometry representing organic tissue. This requires some way of binding particles to these surfaces; a problem for which two solutions were identified. One way of handling interaction between fluids and meshes is by collision response. When particles penetrate a surface, impulses are applied to cancel out the excessive forces. This collision response can also be used to make particles cling onto a wall and run along its surface, rather than drop off into the air. The primary drawback of such an approach is that it requires expensive collision detection between simulated particles and the visual mesh; not the dynamic mesh used in other dynamics. This is further complicated by the fact that particles are rendered on the GPU while all dynamics and collision detection in LapSim is CPU-based.

Hence, an alternative solution which avoided such problems was selected. Rather than simulating them as points moving in three-dimensional space, particles were handled in two dimensions and projected across the geometry surfaces. All particle calculations are then performed in texture-space and particle positions are mapped to texture coordinates.

3.8.1 Coordinate system

Projecting the particles onto a surface means mapping particle positions to texture coordinates. The choice of this mapping has profound effects on the algorithm itself. The first thing to note is that the method of binding positions to the surface of a mesh may change the very nature of the simulated space.

In particular, one must carefully consider the choice of how to handle particles moving over the edge of the texture. One choice is that these particles are no longer drawn; this means the simulation is carried out in an open space, of which only a portion is visualized. Alternatively, the positions can wrap around; the surface is then simply connected. Doing this lets particles flow across certain texture seams, allowing natural simulations across closed meshes. It is essential that the texture wrap mode and shader implementation agree in terms of this choice.

Simulating in free space but treating texture coordinates as modulo the texture size means that particles which are too far from each other to interact may still be drawn next to each other.

Our choice was to use an open particle space. We also treat positions directly as texture indices, which simplifies the hashing. In order to avoid pointer issues and minimize computations, particles outside of the texture area are discarded during the hashing, viscosity and double density steps.

3.8.2 Texturing

When particles are simulated in texture space, it is important to consider how texture coordinates are mapped across mesh surfaces. Any seams which do not connect opposing edges of the texture may lead to particles seemingly jumping or disappearing altogether as they cross the seam. Another source of artefacts is stretching and compression of texture coordinates. The speed at which particles move is dependent on the scale of texture coordinates, which can vary across different surfaces. This same scale also determines the size with which particles are drawn. In fact, not only can this scale vary across surfaces, the scale can also differ between the two dimensions of the same surface. Many of these artefacts are hidden by the blurring and smoothing effects of the visualisation. Still, extreme stretching and compression of textures can lead to elongated particles or significant differences in apparent simulation speed between different surface areas. For the purpose of this report, such concerns are considered the responsibility of the model and texture artists.

3.8.3 Gravity

With particles simulated in two dimensions and projected onto the surface, gravity equations can no longer be carried out in the simulation space. In order to give a realistic behaviour, the direction and magnitude of gravity for a particle needs to depend on the orientation of the surface onto which it will be projected. This is carried out by the following system of equations.

$$\vec{g}_u = \vec{u} \cdot \vec{g}$$
$$\vec{g}_v = \vec{v} \cdot \vec{g}$$
$$\vec{g}' = (\vec{g}_u, \vec{g}_v)$$

In this set of equations, *g* describes gravity in world space, *u* and *v* describe the projection into world space of a unit vector along the respective axes of the particle's texture space and *g'* is the resulting gravity vector applied to the particle velocity.

Before carrying out these equations, one first needs to generate the appropriate *u* and *v* vectors. This is done by two simple GPU passes over the mesh. The first pass stores, at each texture coordinate, the corresponding vertex position. In the next pass, *u* and *v* are calculated as follows.

$$\vec{u} = \Delta u = 0.5\left(\vec{V}_{1,0} - \vec{V}_{-1,0}\right)$$
$$\vec{v} = \Delta v = 0.5\left(\vec{V}_{0,1} - \vec{V}_{0,-1}\right)$$

The position $V_{u,v}$ is attained by an addition of *(u,v)* and the vertex position stored at the current texture coordinate.

3.9 Particle Visualisation

Once the particle system is simulated correctly and a mapping between positions and texture coordinates has been chosen, the one remaining issue is the choice of how to visualize these

particles when drawing the mesh. While particles are simulated as discrete entities, sets of nearby particles should be rendered as contiguous units. The idea is to give smooth visuals and make the blood seem like a fluid substance, rather than individual particles. This leads to an issue of how to identify the fluid surface.

3.9.1 Marching Squares

One solution which was briefly considered was to use the marching squares algorithm. In marching squares, the surface of a mesh is approximated by dynamically creating a set of contours which approximately enclose the original mesh. A grid is used and the corners of each cell are sampled to find whether or not they contain any geometry. Each possible configuration of corners maps to one contour image. Filling each cell with the corresponding image generates a visualisation of the fluid surface. For arbitrary precision, the algorithm can be run recursively down to any predetermined depth for each corner where geometry was found. The downside with marching squares is that checking if a cell contains particles can be very costly. In our application, the procedure can be sped up significantly by aligning the dimensions of the grid with that of our hash textures.

3.9.2 Position Smoothing

In the end, we chose a simple solution of smoothing particle positions. When drawing blood, particle positions are first rendered into a floating point texture: after clearing all texels to 0, each texel corresponding to a particle position is set to 1.0. In the next pass, a square area around each texel is sampled. Dividing the sum of these samples by the size of the sampled area produces an average over said area. The bigger the sample area, the smoother particles will look. However, the complexity of this procedure is quadratic with respect to the side of the sampled area.

$$b = \frac{\sum t_i}{d^2}$$

Here, d is the side of the quadratic area sampled and t_i stands for the i^{th} neighbouring texel. The resulting blood value b is in the range [0, 1] and describes the density of blood in the sampled area. When drawing the blood, b is used to modulate the blood colour while $1\text{-}b$ is used for the surface texture. Thus, particles near each other blend together to form a blood-coloured surface. Furthermore, the edges of this surface fade towards the colour of the surface texture. This gives rise to a smooth transition at the edge of puddles and streaks of blood.

This method alone suffers from serious visual artefacts. As particles move within a fluid, spots of lower density cause flickering across the surface. This effect is alleviated by slightly modifying the output of the first visualisation step. If a value v corresponds to whether any particle position maps to the given texel, we instead write $\max(v, kb)$ where k is a fading constant and b is the current blood value in that texel. In doing this, values are also smoothed over time, hiding minor temporal variations in surface density. Using this approach, it is sufficient to sample only the nine immediate neighbours of each texel.

3.9.3 Hash Position vs. Real Position

In our implementation, the hash map and target texture have the same resolution. This means that, when rendering, particle positions and their hash positions give equal precision. As an effect, the first step is done by rendering, for each pixel, a Boolean value representing whether or not the corresponding hash bucket is non-empty. It is also possible to have a higher resolution on the final drawing texture than what is used for the hash texture. In such a case,

one must instead loop through the hash buckets and, for each particle, render it at its actual position. Doing this will result in higher resolution, but would make visualisation more costly.

4 Results

4.1 Realism

As discussed in section 1.1, the primary motivation of this project was to increase the realism of certain surface effects. In this context, realism implies faithful representation of the processes involved as well as their visual results. For any particular frame in isolation, it is primarily the method of visualisation which determines the perceived realism of a model. When observing an effect over time, however, the behaviour of its underlying model becomes important. For example, the modelling of temperatures has no direct effect on the visuals of any one frame. Still, it was found that modelling temperature greatly improved the perceived realism of the cauterization simulation. With this in mind, the realism of the surface effects was evaluated in how well both their behaviour and visuals approximated reality.

4.1.1 Cauterization

Unlike the current solution in LapSim, our new model supports visualisation of the multiple stages of tissue damage. In the current implementation, colour fades towards white, but never blackens. In our solution, serious tissue damage does lead to blackening of the burned tissue. More importantly, the floating point representation of tissue decay makes it easy to modify the visual behaviour as required. In a manner of seconds one can reconfigure the order and duration of stages, or even insert new stages.

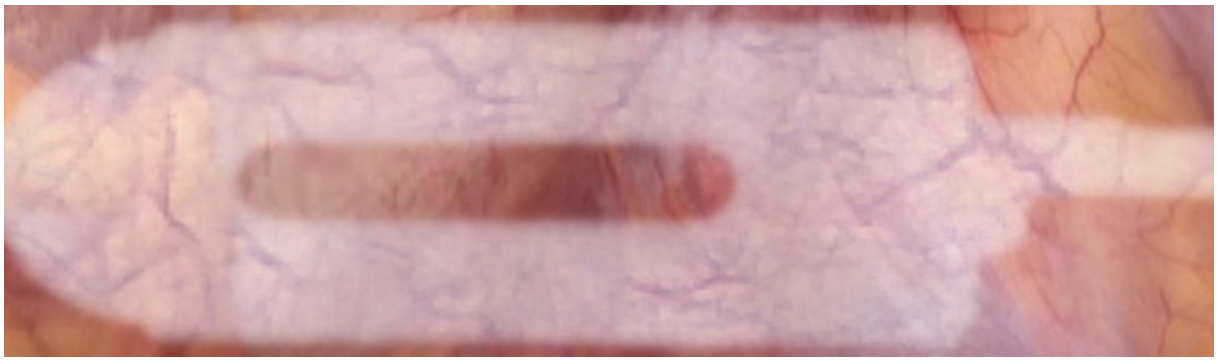


Figure 4: Tissue which has been cauterized using the presented method. The silhouette of the tool used is visible in the resulting burn mark.

Another advantage of our solution is that the shape of burn wounds can closely approximate the shape of the instruments used to cauterize. This can be clearly seen in Figure 4. Currently in LapSim, collisions are detected between the low-resolution dynamic mesh and a few points on the instruments. Thereafter, tissue decay is spread to all surrounding vertices. As a consequence, the resulting marks bear no resemblance to the instruments used to perform the cauterization. On the other hand, our solution uses high-resolution textures for storing burn values and distance field lookups to replace the collision detection. This leads to an effective approximation of collision detection between every point on the surface against every point on the instruments.

One of the most important improvements with the new model is that it is based on temperature, rather than direct manipulation of tissue decay. While the current model is simplistic and should be improved before integration into LapSim, it fulfils its purpose as proof of concept.

4.1.2 Fluid Simulation

The solution presented in this report is a significant improvement over the current fluid simulation used in LapSim. Unlike the current ad-hoc solution, it is based on a well-motivated physical model. It is also pixel-based, allowing much higher resolution than a vertex-based approach. The most evident signs that our fluid simulation is well-behaved lies in its high-level behaviour. As particles travel across the surface of a mesh, they interact to form streaks and puddles of blood. As these collide, one can witness waves travelling across the boundaries of the fluid. This type of behaviour corresponds well with what we would expect from fluids in reality.

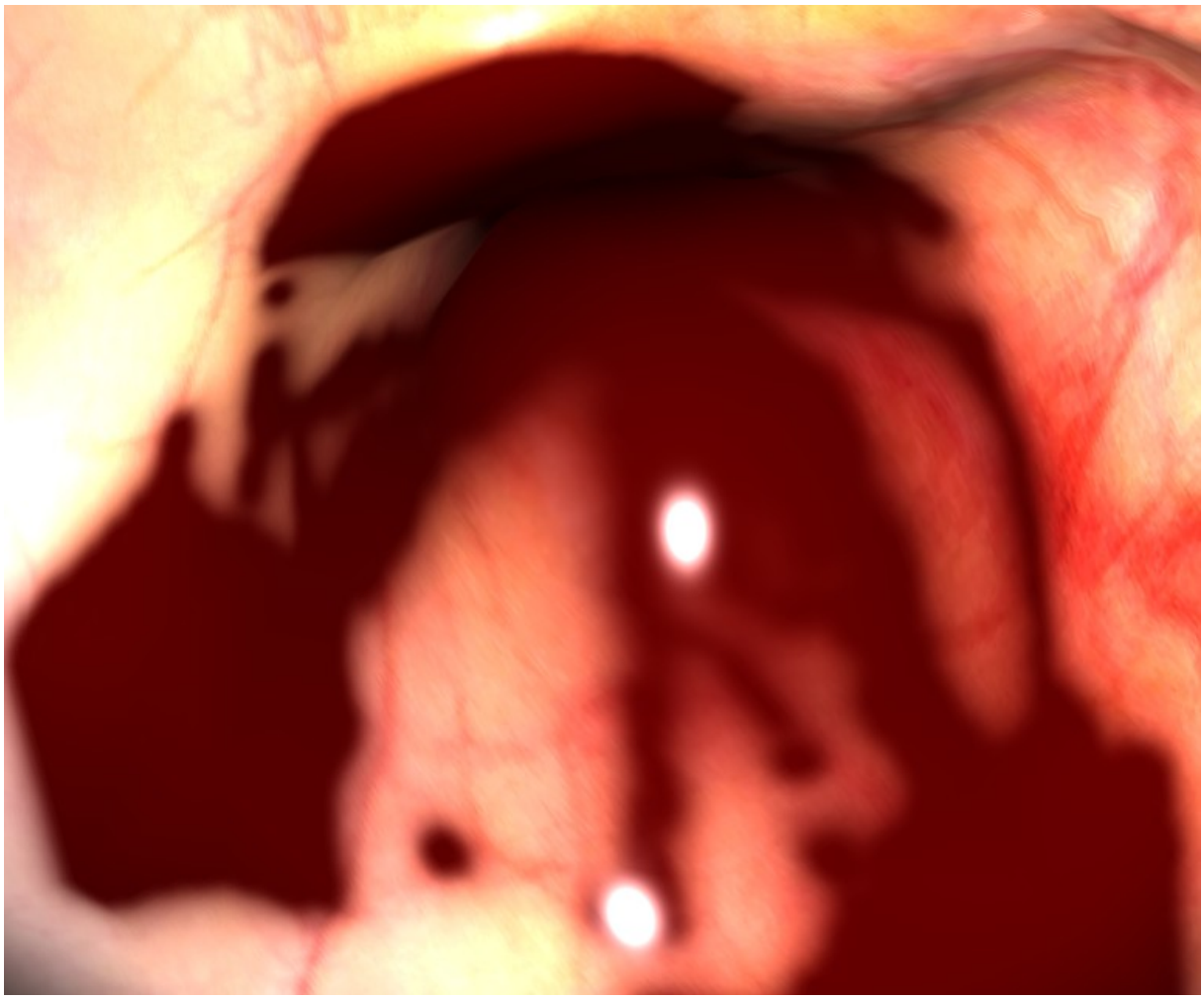


Figure 5: Blood forming streaks and puddles as it runs across tissue.

The visualisation itself also satisfies our demands for realism. Individual particles are not directly visible. Instead, the abstract shape of the fluid is rendered, visualising the aforementioned streaks and pools of blood. Figure 6 shows how distinct particle positions are blurred together to form a solid surface. Furthermore, the blurring gives a sense of fluid density which corresponds well to the number of particles in each area.

The most obvious artefact of this visualisation is an apparent glow around the boundaries of puddles and streaks of blood. This effect comes from the smoothing of particle positions. Other versions which do not produce this type of glow were explored, but their sharp edges made the blood look unrealistic and cartoon-like. While the glow is obvious in still images, it is less noticeable when the simulation is running.

One concern during development was that sinks in the normal map would lead to shaky behaviour. Conceptually, a particle which enters a sink could oscillate between the different sides of its gravitational pit. In fact, this behaviour can be witnessed in single particles. This is not a problem, however, as the inter-particle forces will either hide or cancel this effect even with small numbers of particles around a sink.

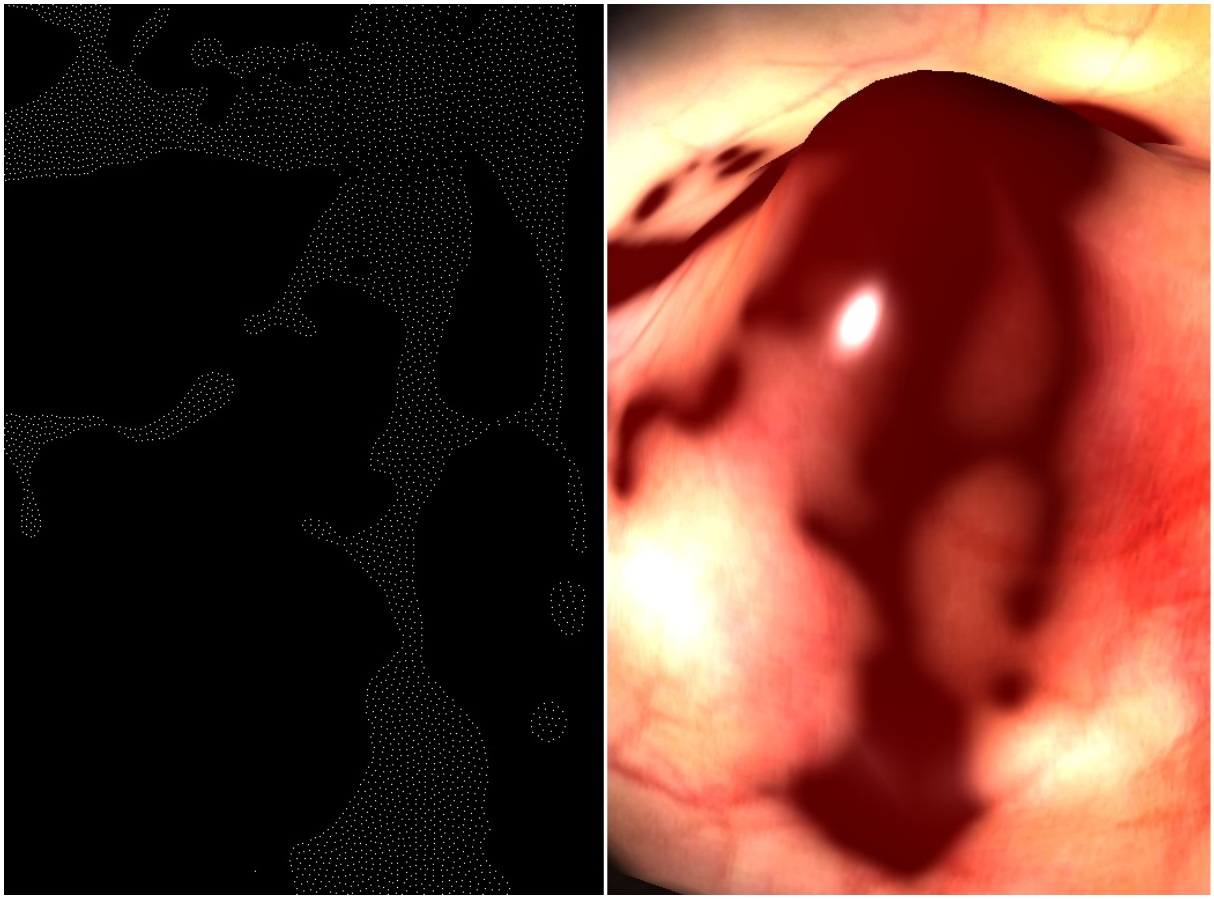


Figure 6: Rendering of particle positions in particle space (left) and final result of rendering particles with blurring (right).

4.2 Performance

When evaluating the results of a real-time graphics implementation, performance is a key point of interest. Performance is usually evaluated in terms of frame rate, measured in frames per second (FPS). These measurements can vary widely depending on the simulated situation, which leads to the problem of how to give a fair performance estimate of a nontrivial system. With many variables affecting the outcome, it is impossible to measure the full behaviour of any reasonably complex system. In order to mitigate this problem, experiments were performed to determine the most relevant variables. Guided by these efforts, independent performance measurements were performed targeting the most interesting aspects of the system behaviour.

Our implementation of cauterization was found to have a frame rate of around 400 FPS. While this performance is acceptable, it is perhaps somewhat low considering the simplicity of the simulation. The reason for this is that the current implementation uses immediate mode for rendering. Using vertex buffer objects instead could increase frame rate significantly. The

reason this was not done within the scope of the project was that the achieved frame rate was, nonetheless, beyond sufficient. As the cauterization shaders contain almost no branching, this frame rate is also solid and displays no mentionable dependency upon any parameters. In view of this, the remainder of this section focuses on the fluid simulation, for which performance is a much more interesting issue.

4.2.1 Performance Measurements

All performance data was gathered by resetting the simulation and recording the frame rate for thirty consecutive seconds, starting shortly after the moment of initialisation. At the time of initialization, the desired amount of particles was spawned in a rectangular grid centred on the origin of our mesh. Doing this gives the particles enough time to settle, while still recording frame rates from the initial phase during which the most activity occurs. The reason that benchmarking does not start immediately after the moment of initialization is that, for large amounts of particles, many particles are initially located in the outermost hash buckets. This may lead to very long linked lists, causing significantly slower evaluation of the first few frames. Including these frames in the measurements would introduce an unfair bias to the performance results.

The hardware used during all simulation was a PC with 4 GB RAM, an NVIDIA GTX 460 graphics card and an Intel i5 CPU clocked at 2.8 GHz. All CPU implementations were written as a single thread and thus utilize a single core.

4.2.2 GPU vs. CPU

The main idea of this project was to achieve efficient particle simulation by exploiting the parallelism of modern graphics hardware. As such, it is interesting to compare results between our final implementation and the CPU-based version. In a series of tests, we measured frame rates for both versions while successively increasing the number of particles. In an attempt to give a fair comparison, the spring steps were removed from the CPU version. Furthermore, visualisation of particles was turned off, as this was never implemented for the CPU and is not part of the simulation model itself. The result of this comparison is visualized in Figure 7. It is immediately obvious that the GPU version scales significantly better than the CPU version. Interactive frame rates can only be achieved for up to 750 particles on the CPU and at 2500 particles, the frame rate has dropped to 7 FPS. At this point, the GPU simulation still reaches a frame rate around 1800 FPS, more than 250 times greater. An interesting side note is that, after this point, CPU frame rate decreases very slowly and does not dip below 1 FPS until 40000 particles are used.

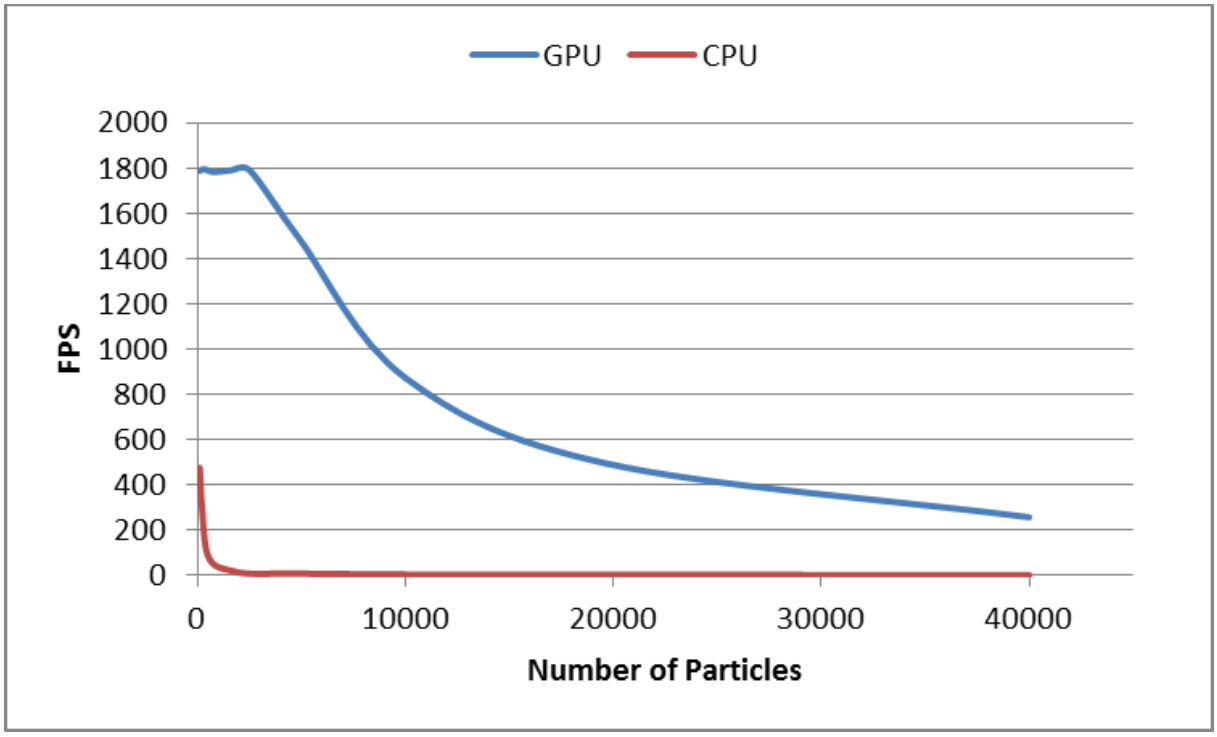


Figure 7: Comparison of frame rates between the two particle system implementations.

This remarkable difference in performance may seem surprising. After all, the source codes for the two variants exhibit the same complexity and differ only in a few details. However, there are clear reasons why such a difference should, in fact, be expected. The task of simulating this particle system is inherently parallel. Large numbers of particles are simulated by performing identical calculations on each. The biggest difference in control flow between two particles is the number of iterations when looping over all neighbours. Consequently, we have almost ideal conditions for exploiting the parallel architecture of modern graphics hardware. Consider that the CPU-based version performs these computations one particle at a time, using a single core with clock frequency of 2.8GHz. On the other hand, the GPU used has 336 CUDA cores clocked at 1.6GHz, all working in parallel. Putting things in this perspective, the great difference in performance is indeed understandable.

4.2.3 Buffer Size

One of the most important variables in terms of memory complexity is the size of the used buffers. In our current implementation, the same resolution must be used for all particle property buffers as well as both hash buffers. Consequently, the choice of buffer size affects everything from the resolution of hashing to the maximum number of particles supported in the system. The necessary minimum resolution depends on the scale of geometry as well as the desired level of detail.

Within our test application, it was found that buffers should have a resolution of at least 256×256 texels to achieve acceptable visual quality. The best results were attained using textures of 512×512 texels. For benchmarking purposes, a resolution of 1024×1024 texels was also used when measuring performance.
As can be seen in Figure 8, performance scaling is very similar between different buffer sizes. In fact, the choice of buffer size does not seem to limit performance in any realistic scenario. The base costs of certain operations do depend on buffer size, which may lead to a noticeable

difference in frame rates before any particles are added to the system. However, this gap quickly closes as particles are added to the system, making these differences irrelevant for practical applications.

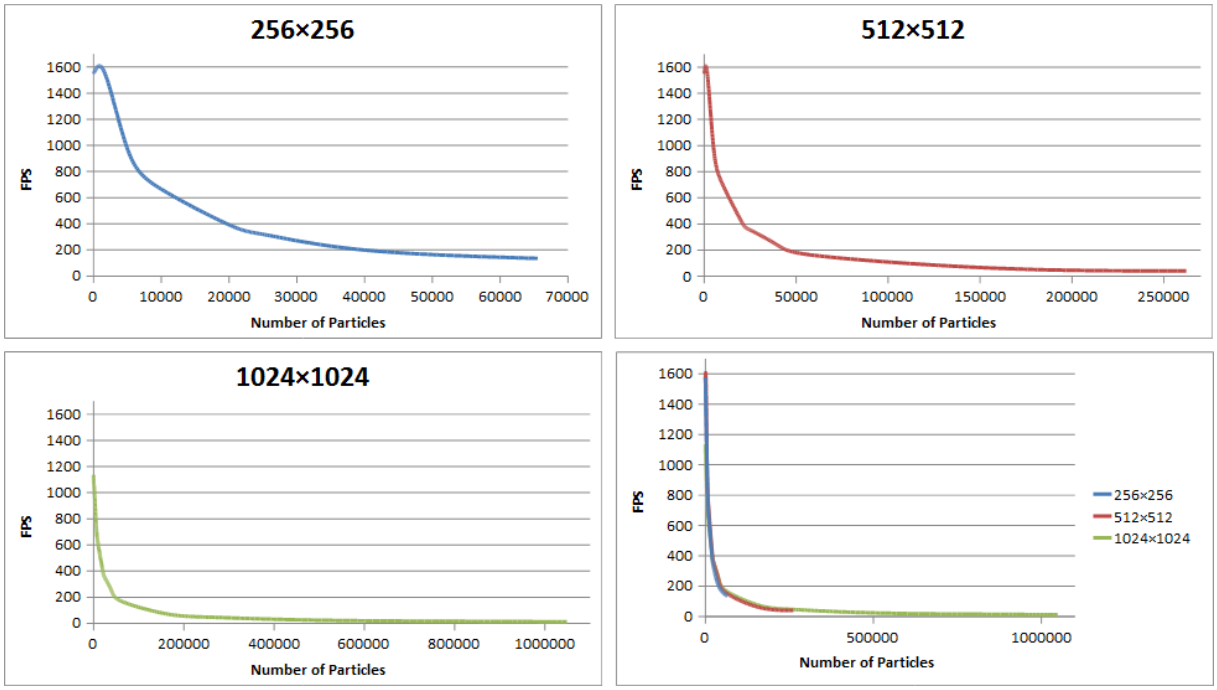


Figure 8: Frame rates for varying buffer size

In theory, performance of the visualisation step does depend mainly on the dimensions of the hash textures. In practice, however, this is negligible in context of the much more complex particle simulation. While there is some smoothing involved, the visualisation is still simple enough that its impact on the frame rate is negligible.

This insensitivity to buffer size is a positive result. It means buffer size can be chosen according to memory limitations and mesh dimensions, without having to consider the effect on performance. An interesting thing to note is that, with low resolutions, it is not even possible to add enough particles to reach below interactive frame rates. With the greatest possible amount of particles in the system, the three resolutions tested led to frame rates of 119 FPS, 38 FPS and 12 FPS respectively.

4.2.4 Particle Spawning

One serious bottleneck in the current implementation is the cost of spawning new particles. Every time new particles are added, the buffers for position and velocity have to be read from the graphics card, modified and written back to the graphics card. This means a substantial amount of data is moving across the graphics pipeline every frame that new particles are spawned. During our benchmarks, it was found that the frame rate could drop by several hundred frames per second due to this data transfer. Unfortunately, there is no way to avoid transferring particle data to the graphics card during this process. However, it is possible to minimize the amount of data transferred. When spawning new particles, their positions and velocities do not depend in any way on particles already in the system. Nor does the spawning of new particles immediately influence the properties of existing particles. Consequently, there is no need to update the entire particle property textures when adding only a few particles. As a future improvement, one could send only the texture rows corresponding to the new particles. This would also mean one no longer has to transfer current particle properties

to the CPU in order to spawn new particles. Implementing these improvements would save a lot of unnecessary traffic over the graphics bus, preventing sharp drops in frame rate.

4.2.5 Density

A key realization regarding performance was that density is probably the most important factor affecting the frame rate of the simulation. The reason is that, even with low density, the traversal of linked lists in the node buffer constitutes a majority of the computation time. As density increases, so do the lengths of these lists. In special cases, where large numbers of particles occupy a small area, frame rates can drop below interactive frame rates even with relatively few particles present.

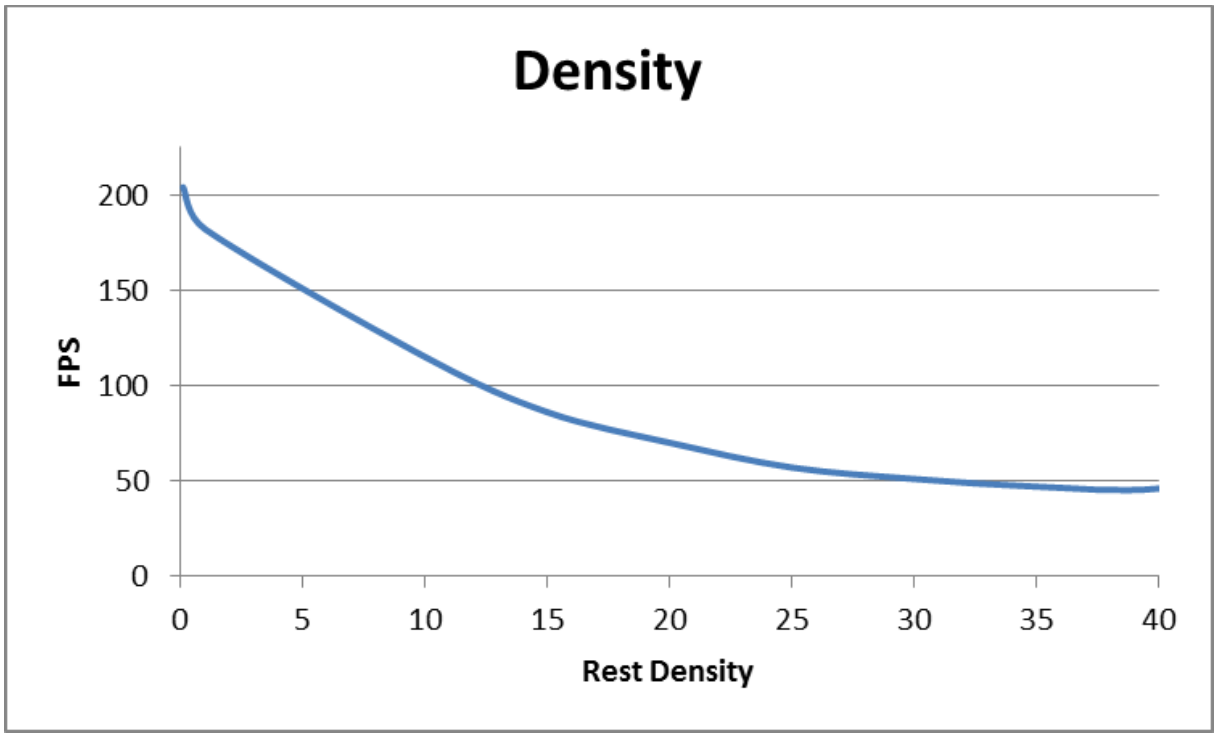


Figure 9: Frame rate dependence on density in a system of 256×256 particles using a buffer size of 512×512.

These occurrences are rare in realistic scenarios and usually last only a short time, while the fluid approaches rest density. However, a high enough rest density would mean that simulations would tend towards these situations. To confirm the effect of long lists in the hash buffer on performance, we ran the same simulation many times with varying rest density. During these tests, we used a texture size of 512×512 and spawned 256^2 particles. The results, visualised in Figure 9, show a clear downward trend as rest density is increased. This confirms the proposition that particle density, and thus the average length of hash lists, has an immediate effect on performance of the simulation. It is important to realise that this variation of rest density was used merely as a way of indirectly influencing the length of linked lists during hashing. Not only does high rest density slow down the simulation, it can also lead to numerical instabilities. During our experiments with density, we found the simulation to be unstable for rest densities above 30. If a higher density is desired for simulation purposes, one should also consider changing resolution of hash textures.

5 Discussion

Designing and implementing the described systems implied a large number of choices. Primarily, we had to identify and select an approach to each problem. Within the context of each algorithm, we also had many options for how to deal with its individual components. For every choice we made, there is also an opportunity to do things differently. It may be that some other path would have led to significantly better results. Such possibilities are difficult to judge, and can never be entirely excluded. What we can say, however, is that we achieved our goals and that we are satisfied with the results.

There are obviously things which, in hindsight, we would have done differently. Still, such speculations should always be taken with a grain of salt. There are indeed mistakes which we genuinely could have avoided. It is difficult, however, to tell these from the suboptimal decisions caused by lack of experience. Experience we could not have attained before making these decisions.

On the other hand, results are not the only things of interest. It is at least as fascinating to have a look at the process itself. Especially when it comes to predictions of how a task will unfold as you work with it. You can learn a lot from the discrepancies between your expectations and the actual turnouts.

5.1 Cauterization

From the very beginning of the project, it was formulated as an implementation of fluid dynamics across a surface, with the possible addition of cauterization marks. This attitude reflected not the relative importance of the tasks, but the expectations on their relative difficulties.

Now that the project is completed, we can safely say that such beliefs were indeed well-founded. The entire cauterization part of the project was performed as a short digression in-between the research and implementation phases of the fluid simulation. This does not mean that it was rushed. The original idea was to work intermittently on the two; it just so happened that the first period of work on cauterization was enough to fulfil our goals. We then decided that what we had achieved was sufficient and that further effort was not warranted.

As has been mentioned above, it is difficult to properly learn new techniques and frameworks while using them to solve a complex problem. For this reason, the simpler task of cauterization served as an excellent learning platform in preparation for the fluid simulation. On an abstract level, both simulations are very similar, the fluid simulation is simply more complex. This allowed us to try our hand at some of the necessary techniques, before having to implement them in the more complex simulation.

5.1.1 Shader Implementation

In particular, the cauterization simulation provided simple tasks to help familiarize ourselves with the necessary methodology for creating multi-pass algorithms using the latest versions of OpenGL and GLSL. The reason that this was non-trivial is that the latest versions of these systems introduce a new way of thinking about shader programming. Previously, GLSL has supplied standardized variables for shader I/O, along with the ability to define supplementary inputs when necessary. The new way of thinking is that the interfaces between OpenGL and shaders, as well as between different shaders during a rendering pass, are entirely determined

by the programmer. As a consequence of this new outlook, practically all predefined variables of GLSL have been deprecated. This, naturally, leads to a period of adaptation and experimentation while getting used to this new way of thinking about shaders.

The relative simplicity of our cauterizations provided an excellent way to experiment with this new methodology, while still working on an actual problem.

5.1.2 Texture Facilities

Something which we thought would prove difficult was the use of general data textures. In particular, we expected issues with the recent support for three particular properties: dimensions which are non-power-of-two, floating-point data and three-dimensional texture sizes. It was not so long ago that working with such textures was only possibly through non-standard extensions. Using such extensions was often difficult and error-prone. As such, it was a surprising relief that, with current technology, these facilities are as easy to use as any other type of texture. Having investigated this meant less worry and more straightforward work when implementing particle systems on the GPU.

5.1.3 Temperature Modelling

While the visual quality of the end result is pleasing, the chosen heat model is too simplistic. The current model does not allow for much spread without numerical instabilities, leading to little actual utility of the modelled temperatures. During development, a lot of time was spent optimizing parameters for visual quality. This time would have been better spent developing a better model for heat transfer. The reason for this lapse in judgement was the late realization that temperature would be an important component of the final solution. When the subject of heat transfer was first encountered, it was deemed out of scope and treated as a possible future extension. If instead, it had been considered a vital part of the application, it would have been possible to reach a more mature solution within the scope of this project.

5.2 Fluid Simulation

5.2.1 Theoretical Models

One of the principal challenges with this project was the complexity and wide variety of mathematical models used to describe fluid motion. Guided by an abundance of research results and articles, we were quickly able to gauge alternatives and determine a feasible solution. However, it seems now that we would have benefited from a more cautious approach to this process. The chosen method was definitely a good alternative. Still, there is a feeling that perhaps it would have been better to spend more time studying the different alternatives before settling on a particular candidate. In essence, a better understanding of the mathematics involved could have simplified the design and helped better motivate some of the important decisions. On the other hand, attaining such knowledge with no practical experience of fluid simulations would have taken considerable effort. This, of course, means it would have taken more time; time stolen from the implementation phase.

Nonetheless, we could have afforded taking a closer look at other SPH variants. Implementing viscoelastic fluid as described by Clavet et al. [23] took little more than a day of work. Perhaps, it would have been well worth it to attempt an implementation of some other carefully selected option. Doing so could have helped highlight the differences between approaches in terms of computational complexity. At the very least, failing to implement a second algorithm could have confirmed our belief that, as fluid simulations go, viscoelastic fluid is relatively easy to implement.

5.2.2 Modification of Algorithms

A large part of our work consisted of adapting algorithms to fit the setting of our project. In doing so, we learned that the details of such techniques are not written in stone. Within a given setting, it is even quite possible to improve upon established techniques. A good example of this is how the linked list scheme could be simplified by knowing the number of particles and assigning IDs ahead of time.

Possibly the best choice made during the entire project was to first implement viscoelastic fluids on the CPU. Not only did this help us in developing the GPU version, it also made it very easy to test variations on the algorithm. We knew from the start that it would not be possible to implement this algorithm, without modifications, on the GPU. Having a CPU version made it very easy to test what results different simplifications would give. A surprising result was that very few details of the original algorithm are actually necessary to achieve seemingly realistic fluid motion. The steps we eventually decided to leave out were the ones involving the springs which provide the elasticity and plasticity of the fluid. This may sound strange, in context of using the viscoelastic fluid model. However, the primary reason for choosing this model was its simplicity, not the elasticity from which it derives part of its name.

5.2.3 Graphics Programming

Implementing solutions for graphics hardware is a unique experience and this project has taught us that modern GPUs are powerful indeed. Their parallel nature means they can operate on huge amounts of data. Additionally, many algorithms scale beautifully on the GPU. This means there is very good motivation for wanting to use graphics hardware for general-purpose computations. However, getting even a simple algorithm to work can be an immense task. It is generally not possible to set breakpoints in shader code or step through it in any way. The only way of testing a shader is by executing the entire graphics pipeline and monitoring the results. This often makes it difficult to find where a bug lies and how to solve it. From our perspective, this hints at a need for future development of the tools involved. Using CUDA, it is already possible to set breakpoints in code which is run on the graphics card. Similar faculties would be invaluable when developing shaders. Currently, the difficulty of writing code for graphics hardware is a serious bottleneck in the development of scientific simulations and electronic entertainment. With better tools to support shader development, such projects could progress at a remarkable speed.

5.2.4 Visualisation and Surface Projection

It was surprisingly easy to achieve sufficient visual quality when rendering the fluid. Our original idea was to create the fluid surface by rendering particle positions and then blurring the resulting image in a series of three passes. By using such a simple method, we hoped to get a hint at how difficult the task of visualization was. Surprisingly, a simplified version of this original scheme turned out to provide sufficient visual quality. Once a working procedure had been developed, a long succession of small improvements were found. During this process, it became clear exploring alternative methods of visualisation could constitute a Master's thesis of its own.

5.3 Utility Libraries

In creating any kind of complex system, there will always be tasks that have to be repeated many times. In something like a graphics application, most operations are variants on the same algorithm on the same kind of data. This means the CPU side of the code contains many different sections setting up the same type of data for the same type of operations. When

doing the same things many times over, writing repetitive code may take up a lot of time. Worse yet, the resulting mess easily causes as well as hides bugs.

In these cases, it is invaluable to have implemented utility classes and functions for the most recurring data types and the operations on them. You only need to implement such utilities once. From then on, you can rely on it working well wherever it is used. One of the most important lessons learned during this project was that, if you find yourself being bothered or worried about repeatedly writing the same code, or hunting the same bug, it is worth it to take a step back and write utilities for code reuse. Another positive effect is that using such utilities is generally much cleaner, resulting in code which is easier to read.

5.3.1 Level of Abstraction

Something which occurred multiple times with several of these utility classes was an uncertainty regarding the proper level of abstraction and generality. When first designing the Frame Buffer Object (FBO) and shader program classes, it was not clear how general these needed to be. There are many variables and modes of usage which can be taken into account. It is difficult to predict what level of customization is actually necessary. At the same time, every unused feature takes time to write and complicates the design. Accordingly, the first implementations of all utility classes were simplistic and only provided support for the patterns actually occurring in the code.

From the beginning, we were aware that these patterns might be broken in the future, requiring a rewrite. In the end, the necessary generality was continuously underestimated when designing, and redesigning, these classes. This does not mean it would have been better to make a more general design from the beginning. These classes were designed to be as simple as possible and, as a result, took very little time to implement. Furthermore, it is only by practical use of a system you learn what you need from it. While using the classes, one quickly learns which restrictions and interfaces impose impractical constraints. Simply put, you easily notice what code is bothersome to work with. When the time came that a generalization was necessary, this experience mostly made the redesign task trivial.

5.3.2 Design Process

As concluded above, a suitable level of abstraction for utility libraries needs to be continuously redefined during the course of a project. Accordingly, a suitable strategy is to create simple utilities which offer little more than is needed at the time when they are written. When some utility class no longer satisfies the user's needs, it should not be given a quick patch. Instead, the overall design should be reconsidered, creating a new tool which is often both more powerful and easier to use. In making such choices, a utility library tends to grow naturally into an increasingly elegant solution to the current problem.

It is important to exercise some caution during this development process. The described methodology is only suitable for fairly trivial classes for interfacing with the underlying frameworks. Larger components of the system should be carefully crafted at the earliest possible design phase. If some utility class grows unreasonably large, this may be a sign that the concept it embodies is something more than such an interface. Another possibility is that what should have been multiple related classes has been put into a single unit. In this case, the best choice is probably to refactor the affected utilities.

6 Future Work

While the systems described in this paper are completely implemented and fully functioning, they are not necessarily the final result. There are many potential improvements, extensions and ideas which can still be explored. These modifications, while relevant, were simply deemed to lie outside of the scope of this project. Some improvements will be implemented as our solutions are integrated into LapSim. Other augmentations have been deemed unnecessary for our particular application, but are still presented here for the sake of completeness. As these ideas are not yet fully developed, their purpose is explained below along with only a vague description of how to implement them.

6.1 Cauterization

6.1.1 Heat equation

In the described solution, heat is increased while cauterizing and dissipates with time. This generated heat is then spread to neighbouring pixels. Balancing the cooling rate and amount of spread is non-trivial. If the two are not carefully configured, heat will either dissipate instantly or grow uncontrollably due to a positive feedback loop. In fact, we did not manage to attain satisfactory level of spread without temperatures diverging. Such issues can be solved by using the heat transfer equations as described by Fourier (Section 2.1.1), rather than an ad-hoc method. With this in place, amount of spread and heat given from instruments can be easily configured without risk of numerical instabilities.

6.1.2 Smoothing Issues

While the smoothing issues described above were considered acceptable, they can be solved. One can generate a texture containing neighbouring texture coordinates. During the heat transfer, this texture can then be used to find the neighbouring heat values for smoothing. The additional computation required by such a method amounts to one additional texture lookup per neighbour for each pixel. This does not count the time required to generate such a lookup texture, as this may be pre-compiled offline. Memory complexity is similarly increased by one integer per neighbour for each pixel. It may be possible to reduce this by using a more advanced scheme, only storing references to those neighbours which lie across texture seams. On the other hand, such an optimization would again require additional computations.

6.2 Fluid Simulation

6.2.1 Particle Lifetime

A simple extension which would give much to the simulation is a lifetime counter for each particle. This could be implemented simply using an integer texture which is initiated to a lifetime value when new particles are spawned and, if non-zero, decremented each frame. Particle steps could then simply terminate if the remaining lifetime was found to be zero.

6.2.2 Adaptive Texture Mapping

It is not always possible to ensure that texturing is made sufficiently uniform for the described method to give good results. By rendering texture coordinates to a texture, the scale of each texture dimension can be calculated per pixel. With these factors available, mapping of positions to texture coordinates can be made adaptive to ensure stretching and compression artefacts are minimized. To minimize computation time, a multi-pass algorithm which samples these written coordinates may instead be used to calculate new, uniform coordinates. This way, position mapping only has to be reconsidered when the mesh is deformed.

At other times, it may be impossible to utilize some portions of the texture or to avoid unfavourable texture seams. Such issues can be treated as discussed in section 6.1.2.

6.2.3 Three-Dimensional Particles

Technically, the implemented particle system supports more than two dimensions. The only real issue is how to solve hashing in three dimensions; the memory complexity of a three-dimensional texture is rather restrictive. In some situations, one axis may be identified along which a coarser level of hashing is necessary. One example is the direction of gravity, in which blood tends to move quickly unless it is stuck to a surface. In such a situation, the hash texture may be set up to use only a few slices in this third dimension, giving a coarser hashing along the least important dimension. When hashing without linked lists, this may lead to performance issues as one must iterate over these hash layers, thus increasing memory cost by a factor L, the number of slices used in the third dimension. Another choice is to hash particles in two dimensions, while simulating them in three. Such an approach may lead to large amounts of unnecessary computations when particles at different depth are placed in the same hash bucket.

The approach which seems to show the most promise is simulating surface-bound particles as described in this report. Liquid not in contact with a surface, however, could be simulated separately using three-dimensional calculations in world space. Due to the presence of gravity, such spurts or drops would only be in free fall for a short amount of time before colliding with a surface. The small amounts of liquid simulated in 3D can then be hashed using only a few vertical texture slices.

6.2.4 Collision detection

Assuming a satisfactory three-dimensional hashing was devised, it would be possible to simulate particles directly in space, rather than projected onto a surface. This way, effects which are currently not supported, such as dripping and pooling, would immediately follow. However, this would require the particle system to interact with the geometry of the scene.

A popular approach to this is to generate surface particles. When particles collide with a surface, new immovable particles are spawned to represent this surface [1]. The particle system will then automatically compensate and particles will flow around the surface, rather than through it. Unfortunately, this idea relies on the view of particle interactions as an equation system to be solved by a numerical solver. With the simplified model of visco-elastic fluids, this is not the case. A simpler solution would be to apply an impulse to any particle colliding with a surface. This impulse would be directed along the normal of the surface and have a sufficient magnitude to move the particle out of the mesh.

6.2.5 Geometry Discontinuities

Issues may arise when the simulated 2D space does not correspond well to the surface of the target mesh. More specifically, when there are discontinuities in either the mesh itself or the mapping of texture coordinates onto the mesh. In LapSim, the most common case is that the discontinuity exists purely in the mesh. For instance, this kind of discontinuity occurs when a user makes an incision in a dynamic surface. This means the mesh now contains a gap while the texture corresponds to the surface as it would be if this gap were to be sealed. If these discontinuities are not considered, particles travelling across the gap will seemingly disappear from one end and instantly reappear at the other. Similarly, if the discontinuity exists in the texture mapping, particles may disappear in one frame only to appear at a seemingly disconnected point some frames later.

There are multiple ways of handling this kind of discontinuity issues. The simplest way is to mark discontinuities in the alpha layer of the particle position textures. With this in place, the simulation itself can prevent particles from crossing a discontinuity. The drawback of this solution is that it instead would lead to particles at the edge of an incision lining up, rather than flowing into the wound. To solve this, one may allow particles to transfer from one surface to another. This would mean keeping track of which surfaces are in contact and how their respective texture coordinates map onto each other.

A more general but expensive solution is to dynamically remap texture coordinates used for simulation in order to remove discontinuities altogether. The area, in world space coordinates, of the gap should then be used to estimate the necessary texture area to represent the gap. There would be no need to change particle positions which map to a surface texel with horizontal coordinate less than the first gap texel on its line. The remaining positions would be incremented by the width of the gap at their corresponding line of the new surface texture.

6.2.6 Obstacle Map

During an actual surgery, the blood does not flow over the surface like a wave which covers everything. Some details extrude from the surface, causing the blood to flow around, rather than covering them. One way of getting such a visual effect, without having to implement collision detection, would be to use an obstacle map. This obstacle map would be a static texture containing, for each texel, a single value indicating whether or not that point is obstructed. For compactness, this could be stored in the alpha component of the normal map. In the simplest case, the obstacle component would be a binary value and blood would only be drawn at unobstructed positions. By setting up a suitable obstruction map, texture artists can make certain details seem like they stick up above the surface of the blood. Alternatively, the obstruction value could be interpreted as a minimum blood value required for blood to be drawn at that position. This would make thick concentrations of blood flow over low obstacles, while thinner droplets would seem to move around them.

6.2.7 Improved Visualisation

The simple visualisation technique implemented for this report has one noticeable issue: as the blood is simulated and visualized across the surface of a texture, it looks flat. This can be alleviated by using the blood value as a depth map. This way, the density of blood in an area is used to calculate displacement along the surface normal. There are many methods of achieving such results. One technique is to tessellate the interesting areas and use a geometry shader to displace the resulting vertices. Another idea would be to implement some form of parallax mapping [26].

The current use of a fading constant was originally motivated by the wish to have blood linger slightly after leaving an area. The intent was to allow the use of fewer particles to produce the visual effect of a constant stream which lingers somewhat after the particles are gone. While it did work very well to solve other issues, it did not achieve this original purpose. As such, a future expansion would be to devise a method which truly accomplishes this. The easiest method would probably be to make the blood texture double-buffered and combine both buffers during visualisation.

Furthermore, the simple method of blurring causes some visual artefacts, discussed in Section 4.1.2. This type of issues may be removed by using a better function for smoothing of blood values.

6.2.8 Particle-specific Properties

Another goal, which was implemented on the CPU but not transferred to the GPU, was the ability to mix fluids with different physical properties. This is done by having constants such as viscosity be particle-specific. This allows for interaction of different kinds of fluid and naturally gives rise to effects such as segmentation of water and oil. On the CPU, interactions between two particles were handled by using the averages of their respective constants. In the context of virtual surgery, particle-specific properties are interesting in order to simulate the interaction between blood and other bodily fluids, as well as water. With particle-specific properties introduced, one would also have to introduce new properties for visualisation. The simplest such would be to draw inspiration from the way fluids are currently visualized in LapSim. Each particle would be assigned its own colour, depending on what type of fluid it represents. When multiple fluids are found in the same spot, their relative concentrations could be used to blend their colours.

6.2.9 Transfer of Properties

After implementing particle-specific properties, this can be taken one step further by implementing explicit dilution of fluids. This would mean interpolating properties between interacting particles over time. Consider a particle system containing water particles and blood particles. As they interact, the denser blood becomes watered out and washes away. A drawback of this method is that it makes it impossible for diluted particles to shift again later, as they may do in real life. Finally, during our experiments with particle-specific properties, the desired effects were found to occur even without this modification. Our experiments showed that as long as one uses a sufficient number of particles, the large-scale interaction will resemble the natural dilution which we wanted to achieve. This means we will get the intended effect without having to continuously change the properties of individual particles.

6.2.10 Coagulation

Another feature which was considered is that of coagulation. This effect is of particular interest for surgical simulations and could be handled by a continuous shift in particle properties. The rate of this shift would most likely be inversely proportional to the velocity of the particles. This way, moving particles require more time to coagulate and blood would stay fluid until it came to a near halt, much like in real life. Combining this with transfer of properties, water could be made to dissolve the coagulated particles, restoring their original properties. However, coagulation does not mix well with the more general dilution described above, as this would produce particles which are a mix of water and blood. If these are allowed to coagulate, we are in effect allowing water to coagulate. If, on the other hand, they are not allowed to coagulate, blood which has been in contact with water will not coagulate. This may be seen as another argument against the transfer of properties between particles.

7 References

1: T. Harada, S. Koshizuka, Y. Kawaguchi. *"Smoothed Particle Hydrodynamics on GPUs*", Proc. of Computer Graphics International, pp. 63-70, 2007
2: J. Lombardo. *"Real-time Collision Detection for Virtual Surgery*", Proceedings Computer Animation, pp. 26-28, 1999
3: C. Langelotz, M. Kilian, C. Paul and W. Schwenk. *"LapSim virtual reality laparoscopic simulator reflects clinical experience in German surgeons*", Langenbeck's Archives of Surgery, Volume 390, Number 6, pp. 534-537, 2005
4: W. Jin et al. *"Improving the Visual Realism of Virtual Surgery*", Stud Health Technol Inform. 111, pp. 227-233, 2005
5: J.R. Cannon. *"The One-Dimensional Heat Equation*", Cambridge University Press, 1984
6: M. Müller et al. *"Particle-Based Fluid Simulation for Interactive Applications*", Proceedings of SIGGRAPH/Eurographics, pp. 154-159, 2003
7: RealFlow [http://www.realflow.com], 2011
8: J.H. Ferziger and M. Peric. *"Computational Methods for Fluid Dynamics*", Springer, pp. 12-13, 1995
9: H. M. Schey. *"DIV, Grad, Curl, & All That: An Informal Text on Vector Calculus*", W.W. Norton & Company, 1997
10: D. J. Price. *"Smoothed Particle Hydrodynamics and Magnetohydrodynamics*", arXiv:1012.1885, 2010
11: S. Chen and G. D. Doolen. *"Lattice Boltzmann Method for Fluid Flows*", Annual Review of Fluid Mechanics Vol. 30, 329-364, 1998
12: Robert Bridson. *"Fluid Simulation for Computer Graphics*", A K Peters, 2008
13: J. Sanders, E. Dandrot. *"CUDA by Example*", Addison-Wesley, pp. 5-7, 2010
14: D. Kirk, W. Hwu. *"Programming Massively Parallel Processors*", Morgan Kaufmann, pp. 3-42, 2009
15: T. Akenine-Möller, E. Haines and N. Hoffman. *"Real-Time Rendering*", A K Peters, pp. 11-45, 2008
16: J. Yang and J. McKee. *"Real-Time Order Independent Transparency and Indirect Illumination using Direct3D 11*", Proceedings of SIGGRAPH, 2010
17: NVIDIA. *"EXT_shader_image_load_store", 2010*
18: B. Stroustrup. *"The C++ Programming Language - Special Edition*", Addison-Wesley, pp. 8-9, 2008
19: OpenGL [http://www.opengl.org], 2011
20: PhysX [http://www.nvidia.com/object/physx_new.html],
21: M. W. Jones, J. A. Bærentzen, M. Sramek. *"3D Distance Fields: A Survey of Techniques and Applications*", IEEE Transactions on Visualization and Computer Graphics, Volume 12, Issue 4, pp. 581-599, 2006
22: Clay Mathematics Institute [http://www.claymath.org/millennium], 2011
23: Clavet et. al. *"Particle-based Viscoelastic Fluid Simulation*", Symposium on Computer Animation, 219-228, 2005
24: C. Oat, J. Barczak, J. Shopf. *"Efficient Spatial Binning on the GPU*", AMD Technical Report, 2009
25: S. Lefebvre and H. Hoppe. *"Perfect Spatial Hashing*", ACM Transactions on Graphics, Volume: 25, Issue: 3, 579-588, 2006
26: T. Kaneko et al. *"Detailed Shape Representation with Parallax Mapping*", Proceedings of ICAT, pp. 205-208, 2001