CHALMERS

Real-Time Fluid Dynamics for Virtual Surgery

Lars Andersson

Master's Thesis
Engineering Physics Program

CHALMERS UNIVERSITY OF TECHNOLOGY
Department of Computer Engineering
Göteborg 2005

Abstract

This thesis report documents the work performed to investigate and implement a number of algorithms for real-time visualization of blood and smoke in the context of virtual surgery applications. More specifically, the effects that we want to implement are blood squirting in free space, blood drops running down a surface, blood dissolving in a fluid and smoke generated by a tissue burning instrument. The work started with a period of initial research to get a better overview of the subject and the possibilities available. This was followed by experiments and implementation of the most promising methods. An uncoupled particle system rendered as a set of spheres is used for the blood splash and drip effects. Blood trails are rendered directly onto the surface texture. The results are not very realistic visually or physically, but may still be usable to give the user visual cues about what is going on in the simulator. The simulation of blood dissolving in a fluid is based on a real-time approximation of the Navier-Stokes equations and is more physically correct. It allows quite realistic real-time interaction with a fluid in two dimensions. The smoke effect is implemented as a simple particle system and probably looks realistic enough to be usable. However, the simulation is not physically based and needs to be modified if interaction with the smoke is required.

Sammanfattning

Den här rapporten beskriver arbetet som utförts för att undersöka och implementera ett antal algoritmer för visualisering av blod och rök i realtid. Mer specifikt är effekterna vi vill åstadkomma blod som skvätter och droppar som rinner utmed en yta, blod som löser sig i en annan vätska och rök från ett värmeinstrument. Arbetet började med en period av efterforskning för att få en bättre överblick av ämnet och tidigare arbeten. Detta åtföljdes av experiment och implementering av de mest lovande metoderna. Ett okopplat partikelsystem med partiklar renderade som sfärer används för skvätt och droppeffekterna. Blodspår renderas direkt till texturen på den yta som blodet befinner sig. Resultaten är inte speciellt realistiska varken visuellt eller fysikaliskt, men skulle ändå kunna vara användbara för att ge visuella ledtrådar om vad som händer i den simulerade omgivningen. Simuleringen av blod i vätska är baserad på en realtidsapproximation av Navier-Stokes och är mer fysikaliskt korrekt. Modellen tillåter realtidsinteraktion med vätskan i två dimensioner. Rökeffekten är implementerad som ett enkelt partikelsystem och ser antagligen tillräckligt realistiskt ut för att vara användbar. Den är dock inte fysikaliskt baserad och behöver förbättras om realisktisk interaktion med röken krävs.

Contents

Chapter 1

Introduction

Virtual surgery simulators are becoming an important tool in the education of surgeons and medicine students. Techniques and procedures can be practiced in a virtual environment before conducting the corresponding operations on real human beings. One of the most important applications is simulation of minimally invasive surgery, also called laparoscopy. The basic idea of laparoscopy is trying to avoid serious damage to the tissue surrounding the actual area of medical significance. Instead of cutting through muscles and other tissue lying in the way of the organs or joints of interest, a fiber optic camera and surgical tools are inserted through a small number of portals in the skin. The medical procedure can then be performed by a surgeon operating the instruments through the portals using visual feedback from the inserted camera. This has often proved to reduce the unnecessary tissue damage, physical pain and time of recovery significantly.

However, not being able to see the affected area directly makes things much harder for the surgeons. Successfully using a television screen or computer monitor as the only visual feedback during surgery requires a great deal of practice. This is a situation where virtual surgery simulators can be of great value. Instead of practising on animals or plastic models which may have both ethical, technological and economical drawbacks, the affected environment and the medical procedures can be simulated and visualized using mainstream PC hardware. Visual feedback is provided by rendering and updating the simulated environment in real-time. Haptic feedback is provided by a set of artificial surgical tools. The artificial tools are manipulated by the practicing surgeon and affected by physical models of the simulated environment to provide a realistic feeling of interaction. With a simulator like this the user may practice and repeat any procedure as much as is needed to gain the skills required.

1.1 The Goals of the Thesis

This thesis project was suggested by Mentice AB in Gothenburg, a company developing simulators for medical procedures such as laparoscopy. The main purpose of the project is to investigate and evaluate different methods and algorithms for real-time simulation and visualization of dynamic, free surface fluids, with an emphasis on virtual surgery applications. After the initial research, one or more of the most promising methods will be implemented using OpenGL for visualization. In addition, the possibilities of accelerating these algorithms using modern programmable graphics hardware will be briefly examined.

If successful, the technology developed could be used to visualize effects such as floating and dripping blood, smoke and water irrigation. To be more specific, a few distinct effects that we want to investigate and preferably implement have been lined up.

- **Blood splashes.** A solid splash of blood, for example emanating from a damaged blood vessel, moving in free air before colliding and interacting with the surrounding tissue.
- **Blood drops.** Drops of blood from a smaller or lower blood pressure cut running down a tissue surface, probably leaving a streak of blood behind.
- **Blood in fluid.** A transparent, saline water solution is sometimes injected into the operation space during minimally invasive surgery. Blood mixing and dissolving in this fluid should be simulated and visualized.
- **Smoke.** A tissue burning instrument is sometimes used during surgery, for example to make cuts or to make blood coagulate in order to prevent excessive bleeding. The fourth goal is to animate and visualize smoke generated by an instrument of this kind.

The performance requirements are quite high. A lot of things are going on in a real-time surgery simulation. For example, the tissue-instrument interaction physics and force feedback simulation need to run at about 500 Hz in order to provide the user with a realistic feeling of interaction. This means that fluid simulation and visualization can only use a very limited share of the total CPU and GPU resources available.

Thus, when running on their own, the effects described above probably need to run at a minimum of 100 frames per second or more on modern hardware to be usable in a real world surgery simulator. This puts quite hard constraints on the type of algorithms and the level of realism that can be implemented. The situation is in many ways similar to the one that game developers are facing when trying to add nice special effects to a modern computer game. A great number of subsystems all want their share of CPU or GPU cycles and striking a good balance is an important part of the overall design.

1.2 Work Flow

The work has been divided into three major parts. Initial research, implementation and documentation. The first five weeks of the project were spent on research alone. A large number of research papers, books and articles were located and quickly evaluated in order to acquire a good general overview of the subject. The sources of information that appeared to contain useful information and algorithms were studied in more detail. This part of the project is basically documented in chapter two of this report.

After the initial period of research and investigation, some ideas about the best ways to solve the problems presented fortunately started to pop up. The methods that appeared to be most promising were studied in more detail and experiments was performed. A basic OpenGL based framework for trying out ideas was developed and a number of different algorithms were tested. This work has been documented in chapter three.

The final part of the project consisted of documenting the initial research, the experiments conducted, implementations coded and conclusions drawn. In short, it involved creating the document you are looking at right now.

Chapter 2

Overview and Previous Work

In this chapter some of the most relevant techniques for computational fluid simulation in the field of computer graphics will be presented. Many of these are not suited for interactive simulation but are still included to present a broader overview of the subject. We will also take a quick look at the possibilities of accelerating these techniques using modern programmable graphics hardware. First of all though, we will start with a brief introduction to the basic equations of fluid dynamics.

2.1 Basic Fluid Dynamics

Computational fluid dynamics (CFD) has been an important tool for scientists and engineers ever since the performance of computers reached useful levels during the 1960's. Many technological feats, such as going to the moon and back, modern jet fighter aircraft and nuclear submarines would have been more or less impossible to achieve without the help of CFD [3]. It is a huge subject that has received a great deal of attention and research funding during the last decades, partly due to the obvious military applications. Here we will just scratch the surface and review a couple of the most fundamental equations relevant to fluid simulation in the context of computer graphics and visualization. See for example [2] for an in-depth derivation of the equations presented here.

2.1.1 The Navier-Stokes Equations

The fundamental equations governing the motion of a fluid are the Navier-Stokes equations, derived independently by the French engineer Claude Navier and the Irish mathematician George Stokes in the first half on the nineteenth century. These equations can take many forms depending on the assumptions made. We will assume that our fluids are incompressible and Newtonian [2], and thus the Navier-Stokes equations take the following form

$$\frac{\partial \mathbf{v}}{\partial t} = -(\mathbf{v} \cdot \nabla)\mathbf{v} + \mu \nabla^2 \mathbf{v} - \frac{1}{\rho} \nabla p + \mathbf{f}, \tag{2.1}$$

where $\mathbf{v}$ is the fluid velocity, ρ is the density, p is the pressure, μ is the kinematic viscosity and $\mathbf{f}$ is the external body force. The first term on the right, $-(\mathbf{v} \cdot \nabla)\mathbf{v}$ is called the *convection* term. It basically represents the change of velocity of a fluid particle caused when the particle moves from one region of the velocity field into another region with different velocities. It can be seen as a "transport of velocity" by the velocity field itself. The second term is the *viscosity* term, representing the internal friction and normal stresses generated between fluid particles as they move in relation to each other. The third term is the *pressure* term, stating that particles are pushed in the direction of the negative pressure gradient. The last term represents body forces, such as gravity or a magnetic field, acting directly on the matter constituting the particle.

2.1.2 The Continuity Equation

To describe the motion of a physical fluid the Navier-Stokes equations need to be complemented with an equation assuring that no mass is created or destroyed in the process. This fact is described by the *continuity equation*,

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0. \tag{2.2}$$

It states that the rate of density change of an infinitesimal fluid element equals the total amount of mass per volume entering and leaving the volume occupied by the element. In other words, mass is conserved.

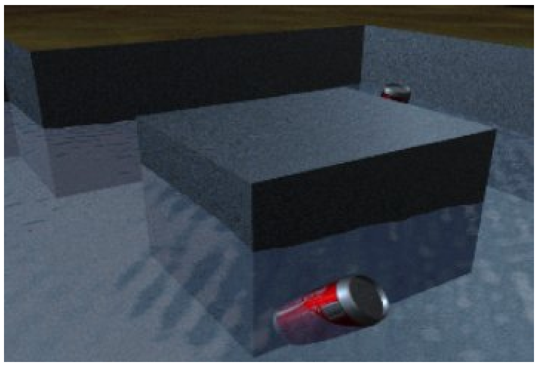
2.2 Grid Based Techniques

Computer graphics generated by grid based (Eulerian) simulation of fluids has been used by the entertainment and special effects industry for at least two decades. One of the earliest examples is the movie "Abyss" from 1989 in which a computer generated water creature moves through a real world submarine vessel. A more recent example is the giant ocean waves hitting New York City in "Day After Tomorrow". Although realistic and impressive, all of these simulations are far away from being able to reach interactive frame rates. Also, apart from being very time-consuming, direct application of standard CFD methods to computer graphics are hard to set up, use and control correctly for people who do not have an in-depth understanding of the underlying principles and equations. In order to come up with something that is actually useful in practice it is often necessary to restrict the simulations in one way or another. Since the beginning of the 90's there has been a lot of research in this direction. A few highlights will now be briefly presented.

2.2.1 Surface and Heightfield Methods

One of the earliest computer graphics related attempts to speed up the simulation of fluids by sacrificing accuracy and generality was done by Kass and Miller [16]. They use simplified versions of the "shallow water equations" in two dimensions and a heightfield representation of the fluid surface to simulate waves in water of varying depth. This method works good for many specialized situations and enables reflection and refraction of waves, net fluid transport and dynamic boundary conditions. On the other hand, due to the 2D heightfield representation, it does not easily permit simulation of truly three-dimensional effects such as breaking waves and interaction with solid objects submerged into the fluid. In short, their method is suitable for simulation of situations such as a calmly flowing river and waves approaching a beach at moderate speed. Raghupathi [33] uses an algorithm based on this method to animate the surface of accumulated blood and irrigation fluid in a minimally invasive surgery simulator.

Another method was presented by Foster and Metaxas [7] in 1996. It solves the full Navier-Stokes equations in 3 dimensions on a low resolution grid to capture a coarse state of the pressure and density fields of the fluid. This is combined with a surface tracking algorithm based on heightfields that follow the evolution of the fluid surface in more detail in order to enable high quality rendering. Boundary conditions such as solid objects and surrounding air are incorporated in the simulation, aligned to grid cell coordinates and treated as fluid cells with special properties. The Navier-Stokes equations are then solved across the entire environment, using a finite difference approximation. Incompressibility is assured by solving the continuity equation iteratively for the resulting velocity field. Their method supports complex fluid behavior such as rotational eddies, vorticity and splashing, as well as submerged objects and obstacles. In [8] the same authors suggest a similar method to simulate the flow of hot, turbulent gases. The work of Foster and Metaxas has many advantages, but unfortunately it hardly reaches real-time performance even on today's hardware. A couple of screenshots generated by this method are shown in figure 2.1 below.


(a) Partially submerged soda cans floating around and colliding with static objects.

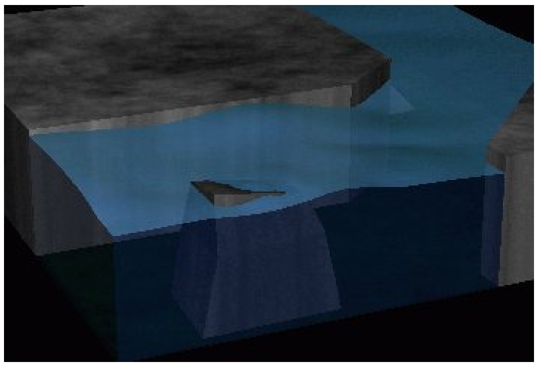
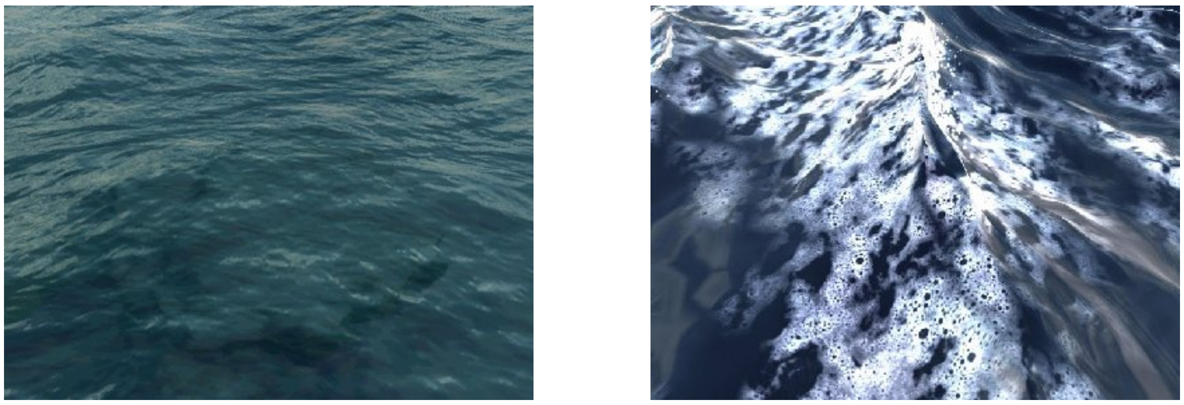

(b) Waves interacting with an underwater rock.

Figure 2.1: Pictures from the work of Foster and Metaxas [7].

A more recent model for animation and rendering of dynamic water surfaces is presented in an article by Jensen and Golias [13]. It is partly based on the same origin as the methods just described above but complements it with other techniques, for example particle systems, to simulate effects such as foam and spray. It also takes advantage of the stable fluid solver algorithms by Stam [39] described below. In addition, modern programmable graphics hardware is used to accelerate some of the calculation and visualization. The result is a fast and flexible simulation of planar water surfaces that could probably run in realtime on today's hardware if implemented correctly. A few screenshots from the article are shown in the figure below.



(a) Dynamic ocean surface. The fine details are rendered using bump mapping.

(b) Simulation of water foam using particle systems.

Figure 2.2: Pictures from the work of Jensen and Golias [13].

2.2.2 Stable Fluids

In 1999 Jos Stam introduced a new and efficient method for solving the Navier-Stokes equations on a grid [39, 40, 41]. Stam used a so called "semi lagrangian" scheme that is unconditionally stable regardless of the size of the time step taken between successive iterations. The "method of characteristics" is used to solve the advection term in a stable manner and conservation of mass is achieved by applying a mathematical result called *Helmholtz-Hodge Decomposition* to the resulting velocity field. Since this method has found many uses in visualization of fluids we will take some time to review it in more detail.

According to the theory of *Helmholtz-Hodge Decomposition* any vector field $\mathbf{w}$ can be decomposed into the form:

$$\mathbf{w} = \mathbf{u} + \nabla q, \tag{2.3}$$

where $\mathbf{u}$ is a mass conserving (or divergence-free, $\nabla \cdot \mathbf{u} = 0$) vector field and q is a scalar field. In other words, any vector field can be expressed as a sum of a mass conserving field and a gradient field. A projection operator P that projects any field into its divergence-free component can be defined as:

$$\mathbf{u} = \mathbf{P}(\mathbf{w}) = \mathbf{w} - \nabla q. \tag{2.4}$$

By applying this projection to the original Navier-Stokes equation (2.1) we get

$$\frac{\partial \mathbf{v}}{\partial t} = \mathbf{P}(-(\mathbf{v}\cdot\nabla)\mathbf{v} + \mu\nabla^2\mathbf{v} + \mathbf{f}), \tag{2.5}$$

where we have used the fact that $\mathbf{P}(\mathbf{v}) = \mathbf{v}$ since the velocity field $\mathbf{v}$ is divergence-free and $\mathbf{P}(\nabla\mathbf{v}) = 0$ since $\nabla\mathbf{v}$ is a gradient (conservative) field. The differential equation (2.5) is the basis of Stam's method. To evaluate the velocity field, starting from an initial state $\mathbf{u}(\mathbf{x}, \mathbf{0})$, a four step procedure is used to advance to the solution by a time step Δt. Let $\mathbf{w_0}(\mathbf{x}) = \mathbf{u}(\mathbf{x}, t)$. The terms are then evaluated sequentially in the following order:

$$\begin{array}{lcll} \mathbf{w_0}(\mathbf{x}) & \rightarrow & \mathbf{w_1}(\mathbf{x}) & \text{Body forces} \\ \mathbf{w_1}(\mathbf{x}) & \rightarrow & \mathbf{w_2}(\mathbf{x}) & \text{Advection} \\ \mathbf{w_2}(\mathbf{x}) & \rightarrow & \mathbf{w_3}(\mathbf{x}) & \text{Diffusion} \\ \mathbf{w_3}(\mathbf{x}) & \rightarrow & \mathbf{w_4}(\mathbf{x}) = \mathbf{u}(\mathbf{x}, t+\Delta t) & \text{Projection} \end{array} \tag{2.6}$$

Body Forces
The force term is easy. The body force, specified as force per unit mass, is simply integrated over the time step Δt using the Euler-forward scheme:

$$\mathbf{w_1}(\mathbf{x}) = \mathbf{w_0}(\mathbf{x}) + \Delta t\mathbf{f}(\mathbf{x}, t) \tag{2.7}$$

Advection
Advection can be seen as a kind of "transport of velocity", or as the velocity field transporting itself. A disturbance in the velocity field propagates according to the advection term $-(\mathbf{v}\cdot\nabla)\mathbf{v}$. It is this term that makes the Navier-Stokes equation non-linear and is thus to be blamed for much of the complexities and difficulties of computational fluid dynamics. It can be solved using standard finite difference methods as described in [7]. However, finite difference discretization of the nabla operator ∇ usually needs quite small time steps to guarantee stability. Stam instead proposed a "semi-lagrangian" technique, based on the *method of characteristics* used to solve partial differential equations. In this case, it can be understood intuitively by considering a single fluid particle $\mathbf{x}$ at time t. During the last time step Δt it was moved to its current location by the velocity field. To find the particle's current velocity, we move backwards in the velocity field from the point $\mathbf{x}$ by Δt. The trace defines a path $\mathbf{p}(\mathbf{x}, t)$ corresponding to a partial stream-line of the velocity field that the particle followed during the last time step. The new velocity of the particle at $\mathbf{x}$ is then set to the velocity it had at its previous location a time Δt ago:

$$\mathbf{w_2}(\mathbf{x}) = \mathbf{w_1}(\mathbf{p}(\mathbf{x}, -\Delta t)) \tag{2.8}$$

Although an approximation, this method has at least two clear advantages over using finite differences. Most importantly, it is unconditionally stable. The velocity can never "blow up" since the maximum velocity of the new field is never larger than the maximum value of the previous field. This enables us to use time steps as large as we please, at the cost of accuracy. Another advantage is that the algorithm is easy to implement and optimize, making it ideal for real time algorithms.

Diffusion

The viscosity term generates diffusion and has a smoothing effect on the velocity field. It has the form of a standard diffusion equation

$$\frac{\partial \mathbf{w_2}}{\partial t} = \mu \nabla^2 \mathbf{w_2} \tag{2.9}$$

This is a well-known equation that can be solved using a number of different numerical methods. As for the advection term, a finite difference scheme and explicit time stepping could be used but would not result in an unconditionally stable solver. Stam instead reformulates the equation as

$$(\mathbf{I} - \nu \Delta t \nabla^2)\mathbf{w_3}(\mathbf{x}) = \mathbf{w_2}(\mathbf{x}). \tag{2.10}$$

After the diffusion operator ∇^2 is discretized, this results in a sparse system of linear equations. This system needs to be solved in order to find $\mathbf{w_3}(\mathbf{x})$. As described in [41], Stam suggests using an iterative, implicit scheme called *Gauss-Siedel relaxation* to calculate the new velocity field $\mathbf{w_3}(\mathbf{x})$. This results in a simple, reasonably efficient and unconditionally stable algorithm.

Projection

In the final step, we need to make sure mass is conserved by projecting the resulting field back to its divergence free component according to equation 2.4. By multiplying both sides of equation 2.3 with ∇ and remembering that $\mathbf{P}(\nabla \mathbf{v}) = 0$ we get the following equations for the projection

$$\begin{aligned} \nabla^2 q &= \nabla \cdot \mathbf{w_3} \\ \mathbf{w_4} &= \mathbf{w_3} - \nabla q \end{aligned} \tag{2.11}$$

The first equation is a variation of the Poisson equation. Like in the diffusion step, the diffusion operator ∇^2 can be discretized and the resulting sparse, linear system can be solved using an implicit, iterative method. When this step has been completed, we have advanced the velocity field by Δt.

Using the computed velocity field, a scalar density field can be evaluated in a similar way. The density field could represent the thickness of hot smoke rising in air, or the concentration of one fluid being mixed with another. A few screenshots from the fluid solver presented by Stam in [41] are shown in figures 2.3 and 2.4.

The stable fluids method was an important contribution to research in fluid dynamics for computer graphics. However, the methods used to solve equation 2.5 are in some ways quite approximate. It would in most situations not be accurate enough to be used in engineering applications where strictly physical behavior is needed. For example, the solver tends to suffer from too much “smoothing” of the density and velocity fields, i.e. the flows dampen too fast compared to reality. However, this is usually not a serious problem in the context of computer graphics. If needed, one way of compensating for the numerical diffusion by introducing an artifical force field is presented in [6]. Another limitation with the solver as described in [39] is that it does not support free surface flows. The fluid simulated is expected to completely fill up the space simulated. The stable fluid algorithm has been used in the context of surgical simulation by Zátonyi et al. [45] to visualize blood dissolving in a transparent liquid.

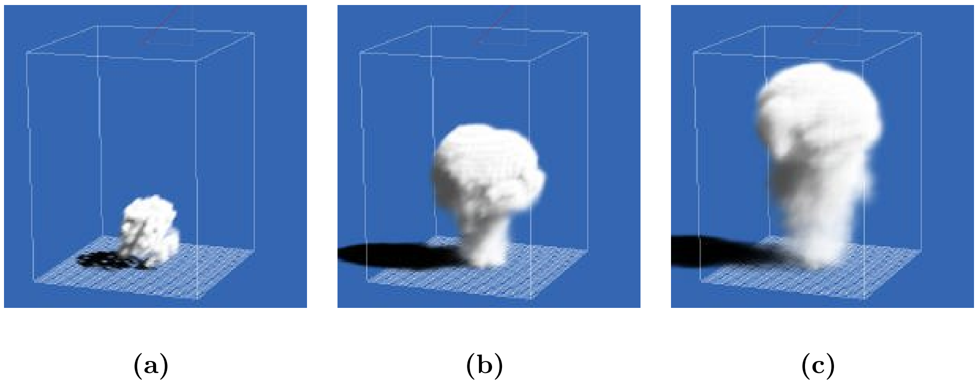

(a) **(b)** **(c)**

Figure 2.3: Pictures from the work of Stam [41], showing the evolution of a rising smoke cloud.

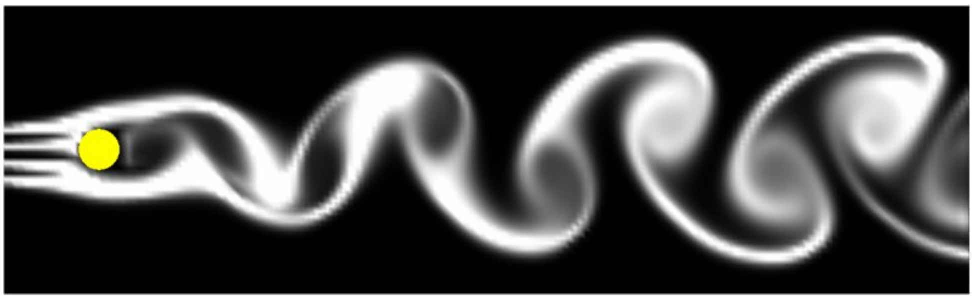


Figure 2.4: Picture from the work of Stam [41], showing the characteristic *Von Karman* vortex street produced behing an obstacle in a moving fluid.

2.2.3 Conclusions

Due to the grid based world description of the models just presented they are not very suitable for simulation of situations when the fluid breaks apart into many separated pieces. Another problem with grid based methods is the efficiency of interaction with dynamic and deformable objects. As the environment changes, the properties of the space subdivision has to be updated, something that tends to be both slow and complicated. To solve these problems, a completely different approach, based on particle systems, can be used.

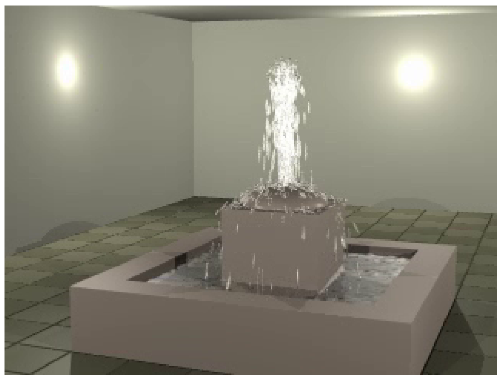
2.3 Particle Based Techniques

Ever since being formally introduced to the computer graphics community by Reeves [34] in 1983, particle systems have been an important tool for real-time graphical effects. Particle systems can be either uncoupled or coupled. In an uncoupled particle system, each particle moves independently of the other particles, usually according to the laws of sir Isaac Newton. In a coupled particle system, the particles also interact with each other in order to simulate more realistic behavior. Another common, but not neccessary feature of particle systems is that they often use stochastically defined attributes, such as initial velocity or particle color.

2.3.1 Uncoupled Particle Systems

Uncoupled particle systems have been used in computer games for ages. One of the very first examples is the old classic game *Space War*, developed on a PDP-1 in 1961, initially using a converted oscilloscope as the display device! Although not really based on any strictly physical principles for the system as a whole, uncoupled particle systems are still frequently used in computer games to visualize everything from splattering blood, flocking birds and falling rain to smoke and explosions. Thanks to the simple principles used to update each particle, a large number of particles can be used which helps making the effects more visually convincing. However, since there is no connection between the particles, uncoupled particle systems are not ideal for simulation of substances in which internal forces have a notable effect on the behavior. In spite of this though, uncoupled particle systems have frequently been used to model this kind of substances with good visual results.

(a) Fluid particles flowing over a bumpy surface using "slide" collision detection and friction.



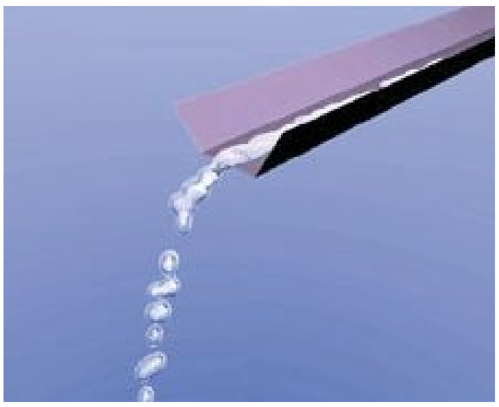
(b) Fountain of water. To emphasize their motion and direction, the particles are stretched out along their velocity vector.

Figure 2.5: Pictures rendered using "Rune's Particle System", an extension to POV-Ray based on uncoupled particles.

2.3.2 Coupled Particle Systems

Coupling the system by letting the particles interact with each other generates many possibilities to make the particle system as a whole behave more realistically. For example, a simple way to make the system act more like a continous fluid would be to let the particles affect each other by some kind of *Lennard-Jones* potential, i.e. by letting the particles repulse each other when they are very close, attract each other at medium distances and letting the force approach zero as the distance is further increased. This simple model may be extended by incorporating functions emulating effects such as internal friction and mechanical damping. This approach was taken by Murta and Miller in [28] to simulate the motion of dripping and splashing fluids. They use *Lennard-Jones* interaction as described above and a simple Euler scheme with small time steps to integrate the resulting particle accelerations.

To enable interaction between the fluid and static objects, ray-polygon intersection is performed for each particle as it moves. If the path of a particle intersects a surface at high speed the particles' velocity along the surface normal is reversed. A random deviation is also added to the velocity vector to encourage more natural, non-deterministic splashing behavior. Fluid particles hitting a surface at low speed are unlikely to possess enough kinetic energy to overcome the adhesion forces generated. Their velocities are instead adjusted to make them slide along the surface. An additional force is added to simulate the friction between the fluid and the surface.


(a) Water running down a metal bar, breaking into separate pieces as it falls through the air.

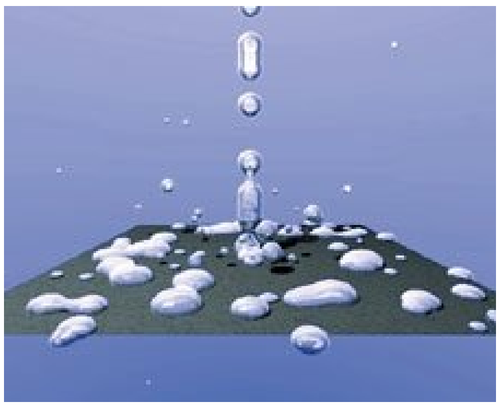

(b) Another example. A splash of fluid splashing and separating into drops as it hits a surface.

Figure 2.6: Pictures from the work of Murta and Miller [28].

Another idea introduced by Murta and Miller is particle splitting and merging. While big chunks of fluid can be represented by relatively few, large particles, small droplets and splashes usually need higher detail. This problem is solved by allowing large particles to split into smaller ones in situations of high dynamic stress. For example, when a particle hits a surface, it is subdivided into about 4-6 smaller particles to catch the fine details of the resulting splash. The reverse process is also used and lets small neighboring particles with similar velocities merge back into a larger particle.

Surface Generation
The particle system is rendered by creating an isosurface of the particle densities. The surface is implicitly defined by all points $\mathbf{x}$ in three dimensions satisfying $f(\mathbf{x}) = c$, for some function f and isosurface constant c. Each particle generates a spherical density field around itself. The function f is defined as the sum of the density field of all particles. In [28] the following form of the total density function f for all particles i is suggested

$$f(\mathbf{x}) = \sum_i \begin{cases} \frac{2d_i^3}{R^3} - \frac{3d_i^2}{R^2} + 1 & \text{if } d_i < R; \\ 0 & \text{otherwise,} \end{cases} \tag{2.12}$$

where R is some cutoff distance and d_i is the distance between $\mathbf{x}$ and the position $\mathbf{p_i}$ of particle i, weighted by the particle size m_i

$$d_i = \frac{|\mathbf{x} - \mathbf{p_i}|}{m_i} \tag{2.13}$$

The implicit surface defined by $f(\mathbf{x}) = c$ can be visualized using ray tracing as in [28], or polygonized using for example the *Marching-Cubes* algorithm [19].

2.3.3 Smoothed Particle Hydrodynamics

Smoothed Particle Hydrodynamics, SPH in short, is another form of coupled particle system. It was invented in 1977 by Lucy [20], and Gingold and Monaghan [10] for simulation of astrophysical phenomena such as star formation and galaxy collisions. The fluid is made up by a large number of particles with properties such as mass, density and velocity. In effect, the particles represent interpolation points occupying a certain region of space rather than real, physical fluid particles. The basic idea is to define a continous field from the values at a set of discrete points, the particles, using a radially symmetric weighting function W called the *Smoothing kernel.* A scalar quantity f is estimated at a location $\mathbf{r}$ by a weighted sum of contributions from all particles according to

$$f(\mathbf{r}) = \sum_j f_j \frac{m_j}{\rho_j} W(\mathbf{r} - \mathbf{r}_j, h), \tag{2.14}$$

where j iterates over all particles, m_j is the mass of particle j, $\mathbf{r}_j$ its position, ρ_j the density and f_j the field quantity itself at the location $\mathbf{r}_j$. The argument h passed on to the smoothing kernel W determines the size of its core radius. As an example, to evaluate the density ρ at any location $\mathbf{r}$, equation 2.14 turns into

$$\rho(\mathbf{r}) = \sum_j m_j W(\mathbf{r} - \mathbf{r}_j, h), \tag{2.15}$$

Derivatives of field quantities, such as pressure gradients, are evaluated by analytic differentiation of the smoothing kernel. We thus get the following expression for the gradient of a scalar field f

$$\nabla f(\mathbf{r}) = \sum_j f_j \frac{m_j}{\rho_j} \nabla W(\mathbf{r} - \mathbf{r}_j, h), \tag{2.16}$$

To simulate a physical fluid we need an equation of state that relates fluid pressure to density changes in the fluid. One of the standard equations of state used in SPH was suggested by Monaghan [22] and takes the form

$$P(\rho) = B\left(\left(\frac{\rho}{\rho_0}\right)^\gamma - 1\right), \tag{2.17}$$

where p_0 is the initial particle density, $\gamma = 7$ and B is a constant. We can see that small variations in density will produce large variations in pressure, which is the behavior we want when simulating nearly incompressible fluids such as water or blood. In order to use this equation the density needs to be evaluated at the center of each particle. This could be done using equation 2.15, but this has a couple of serious drawbacks. In practice, another method based on calculating the rate of change of density at each particle center using the continuity equation is used. This rate of change of density can then be integrated in time for each particle using some standard scheme such as *Runge-Kutta* or *Leap-frog* integration.

In a nonviscous fluid, the force on each fluid element is proportional to the local pressure gradient. An equation of motion based on pressures evaluated using equation 2.17 can thus be derived to update the velocity and location of each fluid particle. Additional force terms such as artificial viscosity and gravity are often added to the final equation of motion. This has been a very brief introduction to the fundamental equations of SPH. See [21] or [36] for more details and full derivations of the equations.

SPH was originally developed for simulation of compressible gases but has been adapted to better approximate free surface flows of stiff, practically incompressible fluids like water [22, 24]. However, using a stiff equation of state to simulate nearly incompressible fluids has a negative impact on the numerical stability of the method, making it necessary to advance the simulation by very small time steps. The popularity of SPH in recent years has resulted in several investigations regarding the stability properties of SPH. Monaghan suggests introducing artificial stresses in the fluid to stabilize the system [23].

Most of the advantages of SPH is related to its Lagrangian (particle based) nature. For example, it is easy to define arbitrary shaped boundaries and there are few restrictions to the way the fluid may deform or separate into disjoint pieces.

Recent work in the field of SPH includes an adaptation for better real-time performance by Müller et. al. [26]. They use specialized smoothing kernels to improve the stability in order to enable larger time steps and to increase performance. Surface tension is also simulated, based on the principles discussed in [25]. Another article by Müller focuses on the application of SPH to virtual surgery [27]. Blood flowing through vessels, free surface splats and splashes are successfully simulated in real-time. The problems solved in [27] have much in common with the goals of this thesis. A few pictures from the work of Müller are shown in figure 2.7 below.

(a) Blood pouring out from a damaged artery. The particle nature of the fluid is easily spotted by noting the individual particles constituting the drops at the bottom.

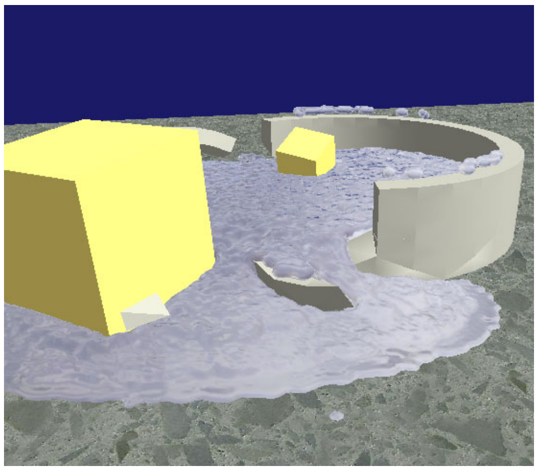

(b) Water leaking out of a broken container. The simulation is running in real time on a fast Athlon XP based system.

Figure 2.7: Pictures from the work of Müller et al. [26, 27].

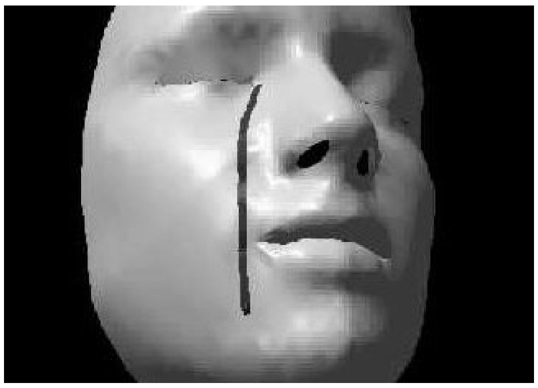
2.4 Other Techniques

Over the years, many specialized techniques for fluid visualization in specific situations have been developed. A short review of the ones most useful for our purposes will now be presented.

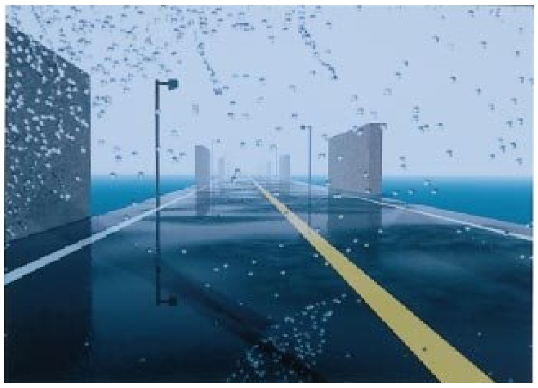
A combination of several different techniques to simulate splashing of fluids hit by solid objects was presented by O'Brien and Hodkins in [30]. They represent the water body as a grid of connected columns. The flow of fluid between neighboring columns is calculated by using physical laws for hydrostatical pressure due to gravity and external forces. The height of each column is simply determined by dividing the volume by the bottom area. The surface model enables external objects to interact with the fluid. Objects that collide with the surface mesh generate forces that are propagated as external pressure to the column based volume model. As a final touch, a particle system is added to simulate spray generated as liquid droplets are disconnected from the surface. This model is in many ways able to simulate, although less accurately and realistically, the same effects as the more computationally expensive model by Jensen and Golias [13] described above.

Basdogan et al. [4] use an auxilary surface to visualize blood from a surgical cut spreading on a tissue surface. The auxillary surface representing the surface of the blood is initialized as a rectangular 2D grid mapped onto the region of interest. The grid is placed just below the tissue surface. When a bleeding occurs the wave equation is solved on the grid to calculate the height of each gridpoint. The blood surface is thus raised above the tissue surface and appears as fluid waves spreading across the tissue surface.

A method specifically targeted for simulation of liquid droplets running down a surface is discussed by Fournier et al. in [9]. The surface on which the droplets are moving is represented as a simple mesh of triangles where each triangle has individual properties such as roughness and wetness. The computation of droplet motion and droplet shape is treated separately. In the case of motion, each droplet is represented by a single particle and is accelerated by forces such as gravity, friction and surface adhesion. The shape of the droplet is computed by using a quite complex model based on physical constraints such as incompressibility and the tendency of droplets to form a surface making minimal contact with the surrounding air and maximum contact with the underlying solid. Due to the detailed model, the calculation of the droplet shape is more than a hundred times more time consuming than calculating the movement of the droplets.



(a) A tear running down the cheek of a crying surface mesh. At the distance used here the sophisticated calculation of the droplet shape is hardly visible.



(b) Water droplets on the windshield of a moving car. The droplets are affected by external forces such as wind, gravity and surface friction.

Figure 2.8: Pictures from the work of Fournier et al. [9] and Kaneda et al. [15].

A similar method is used by Kaneda et al. in [15] to model water droplets on a transparent, curved surface, such as the windshield of a vehicle. Compared to the complex calculation of droplet shape used in [9], a less accurate but much faster way of modeling and rendering the droplets is used, making this algorithm potentially interactive. Another, related technique to simulate simple fluid flow over a bump mapped surface is presented by Jonsson and Hast in [14]. They use the normals stored in the bump maps to affect the small scale movement of the fluid. A couple of pictures from the work of Fournier and Kaneda are shown in figure 2.8.

2.4.1 Procedural Noise

A number of techniques specifically targeted at simulation and visualization of gaseous phenomena such as smoke and fire, as opposed to general fluids including water, has been proposed.

One approach is to model the evolution of gas density using some kind of random noise function or procedural texturing. A frequently used form of smoothed random data called *Perlin noise* was introduced by Perlin [32] in 1985. It is created by summing up functions of different amplitude and frequency, each defined by smooth interpolation between the random values at a set of discrete points. For example, a simple 2D simulation of dynamic smoke could be implemented by a 3D Perlin noise function, where the third dimension would represent time. Another way to use Perlin noise is described in [5]. They use bill-boards painted with Perlin noise textures moved along animated, timedependent spline paths to simulate smoke generated by a tissue burning instrument. These methods often require considerable amounts of tweaking and implementation of ad-hoc features before producing convincing simulations. However, if the right parameters are chosen, noise based algorithms can be very useful.

An attempt to combine procedural noise functions with the skills of a human animator was presented by Stam and Fiume in [37, 38]. They use a diffusion process controlled by density and temperature to evolve particles along a time dependent velocity field, completely defined by the animator. However, in order to capture small scale details and turbulence, the user defined velocity field is perturbed by a procedural noise function. The techniques used to generate the small scale noise velocity field are quite sophisticated in order to guarantee features such as conservation of momentum and total velocity sum equaling zero. Despite not being based on any physical principles this method can give very visually convincing results in the hands of a talented animator. Holtkämper [12] uses a similar approach for the small scale turbulence but employs a regular particle system and Newtonian dynamics to update the system at the large scale. This enables easier simulation of interaction between the gas and solid objects.

2.4.2 Advected Textures

Another way to combine physical simulation with artistic talent is to perform the physical calculations on a quite coarse grid and fill in the details by mapping textures onto the resulting image. The textures could either be based on real photos, hand drawn by an artist or based on procedural turbulence functions. Ideas on how to advect a texture along a dynamic velocity field such as a fluid simulation are presented by Neyret in [29]. The basic idea is to map a grid of polygons onto the simulated fluid surface and let their texure coordinates be advected as massless particles by the simulated velocity field. The texture coordinates are periodically reset back to their original values in order to avoid too much stretching of the texture image. To ensure time-continuity of the animation the texture is scaled by a weight factor fading to zero at the start and end of the period. By blending three sets of textures phase shifted by $2\pi/3$, a constant weight and continous animation is achieved. Neyret also suggests more advanced algorithms based on procedurally generated *Flownoise* to better capture the small scale details of the flow.

2.4.3 The Lattice Boltzmann Model

A completely different approach to fluid dynamics, based on cellular automata, is the *Lattice Boltzmann Model.* Cellular automata was originally proposed by computing legend John Von Neumann as a formal model for self-reproducing organisms. The model defines a spatial lattice of cells having a set of discrete properties. For each cell and time step, these properties are updated locally based on the previous state of the cell and its nearest neighbors. A famous example of cellular automata is "the game of life", a classic screen saver effect of the 90'ties. In the Lattice Boltzmann model, several states are defined at each cell to indicate if microscopic packets representing fluid particles are moving in a set of discrete directions. When taking a time step the states of a cell and it's neighbors are combined and updated according to a set of rules designed to preserve mass and momentum. It can be proved that simulation using a large number of cells statistically satisfies the Navier-Stokes equations for an incompressible fluid. The Lattice Boltzmann model has been used quite frequently in computer graphics research. Wei et al. use it to simulate smoke, fire and other gaseous phenomena [42, 43]. Much of the recent interest is generated by the fact that the algorithm is well suited for acceleration in hardware on modern GPUs.

2.5 Programmable Graphics Hardware

During the last couple of years, consumer level graphics hardware has evolved from being specialized rendering devices into fast, parallelized, general purpose, floating point computational engines. Complex programs executed at the vertex and pixel level can be written in high level languages such as *OpenGL Shading Language* or *Cg.* As an example, an Nvidia 6800 GPU supports 32 bit floating point pixel manipulation and program lengths up to 65536 instructions. A Pentium4 CPU running at 3GHz has a theoretical floating point performance of about 6 GFLOPS. An Nvidia 6800 GPU has an observed performance of 40 GFLOPS. Researchers have recently started to look for ways to use all this floating point power for tasks other than graphics and rendering. The flexibility and programmability of recent GPU's makes it possible to accelerate many different algorithms on graphics hardware. Due to their parallel nature, many fluid simulation algorithms are quite suitable for GPU implementation. An early GPU implementation of Stam's stable fluid algorithms is described by Harris [11]. The density and velocity fields are represented as floating point textures and updated using repeated application of relatively simple fragment programs. A similar but more advanced approach is presented by Lui et al. in [44, 18]. Their work allows simulation in a 3D environment with complex obstacles and boundary conditions. Using GPU's for this kind of simulations is clearly the way to go in the future.

2.6 Conclusions

This chapter has been an attempt to briefly review some of the most common methods for simulating and visualizing dynamic fluids. It is now time to come up with conclusions as to which methods would be most suitable for solving each of the problems described in chapter one. Let us go through one at a time.

2.6.1 Blood Splashes

This is arguably the most difficult of the four problems presented in the first chapter. The blood splashes that we want to visualize have a very dynamic free surface boundary and could easily split into several distinct pieces when hitting an obstacle. A grid based algorithm could thus most likely be ruled out as an efficient real-time solution to this problem. This basically leaves some kind of particle system as the most promising approach. Something along the lines of the work by Murta and Miller, as shown in figure 2.6, would definitively provide a nice visualization, but would also be much too slow on current and near future hardware. On the other hand, using some kind of simple and efficient coupling between the particles should not be impossible. A real-time implementation of a Lennard-Jones form of interaction or a simulation based on smoothed particle hydrodynamics may be fast enough. The work by Müller et al. shown in figure 2.7 is probably still using up a bit too much resources to be used as part of a real world simulator, but it feels like the way to go in the future. One of the most performance critical steps in this kind of effect is generation and rendering of the surface implicitly defined by the set of particles. The *Marching-Cubes* algorithm [19] is a straightforward way to generate a polygonized surface, but to get nice results a relatively high resolution grid must be used and it tends to be quite slow. Alternative methods such as surface splatting [46, 17] or GPU isosurface mesh generation [31] may provide a more efficient way to render the surface. Surface generation however is a little bit out of scope for this thesis. I will thus concentrate on trying to implement some kind of coupled or uncoupled particle system to achieve this effect.

2.6.2 Blood Drops

Drops running down a surface could probably be implemented using the same methods used for blood splashes. Both effects need to interact with the surrounding tissue surface. The situation is a bit more restricted in the case of drops though. This is something that we may be able to take advantage of in order to speed things up a bit. The work by Fournier et al. provides some good ideas but is much too inefficient as presented in [9]. Their sophisticated model for the shape of the drops could be replaced with something much simpler, or even just using a simple nondeformed sphere as the shape. Another idea is to use a vertex-program to stretch the sphere a little bit along the direction of the droplet's current velocity vector. The blood trace left behind moving droplets could probably be implemented by constructing some kind mesh or leaving a set of droplets along the path. A less realistic but more efficient approach might be to actually render a trace of the path into the texture used for the surface in question. This method would result in no performance overhead at all for the blood streaks when the drops have stopped moving.

2.6.3 Blood in Fluid

This is clearly a job for a grid based fluid solver as the effect has no free surface interfaces and presents well defined boundary conditions. Stam's unconditionally stable algorithm appears to be exactly what we need. Running a full 3D simulation using Stam's solver will not be fast enough to be usable in a real world simulator right now. This may be a problem in some situations, but as long as the viewpoint is static a 2D simulation will probably provide quite reasonable visual feedback. A compromise may be to use full grid resolution in the plane perpendicular to the viewing direction, and something like two or three grid layers stacked along the line of sight. Another advantage of Stam's algorithm is that it seems to be quite easily accelerated on modern GPU's.

2.6.4 Smoke

Simulating smoke can be done in many different ways. It's not really obvious to me which method would clearly be the best to achieve the wanted effect. An uncoupled particle system using transparent, textured and animated billboards to render the particles could be made to look quite realistic. But using just an uncoupled particle system would also make it difficult to correctly simulate physical interaction between the smoke and a surgery instrument for example. If physical realism and interaction is needed, Stam's stable fluid solver may prove to be a good solution for this effect as well. A problem could be that the smoke may appear a bit too much like a fluid in liquid form. A possible solution to this is to calculate a low resolution velocity field using Stam's solver and then move animated and textured particles around using this velocity field. Due to its simplicity, efficiency and ease of implementation I have decided to implement the smoke effect using a simple uncoupled particle system.

Chapter 3

Experiments and Implementations

In this chapter, the experiments conducted and the algorithms that have been tried out in practice will be described in more detail. Not all of the methods that were tried worked out as expected but the less successful attempts will also be described shortly in order to document the work performed. Three different methods of simulation, SPH, uncoupled and coupled particle systems were tried out as a way to simulate squirting and dripping blood. Stam's stable fluid solver was used for blood in fluid simulation, and a simple particle system was implemented to simulate smoke.

3.1 Smoothed Particle Hydrodynamics

The first experiment was to implement a simple fluid simulation using Smoothed Particle Hydrodynamics (SPH). As concluded in the second chapter, SPH is still a little bit too CPU demanding to really be useful in a real world surgery simulator, but it would still be an interesting and useful experience to give it a quick try. I first tried to reproduce the results presented by Monaghan in [22], particularly the breaking dam experiment as illustrated in figure 3.1.

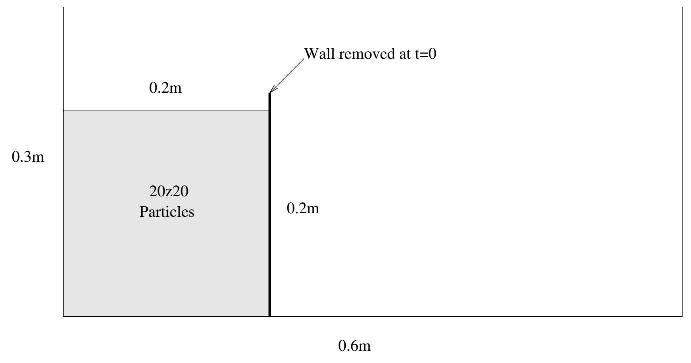


Figure 3.1: Breaking dam simulation at $t = 0$.

The simulation is performed in 2 dimensions according to the basic principles of SPH as outlined in section 2.3.3. A static block of fluid is initially held in place by a wall that is removed at $t = 0$. The standard, normalized cubic spline smoothing kernel as shown in figure 3.2 is used. In 2 dimensions it takes the following form mathematically

$$W(\mathbf{r}_{ij}, h) = \frac{1}{h^2} f\left(\frac{|\mathbf{r}_{ij}|}{h}\right), \tag{3.1}$$

where,

$$f(s) = \frac{10}{7\pi} \begin{cases} 1 - 2s^2/2 + 3s^3/4, & \text{if } 0 \leq s < 1; \\ (2-s)^3/4, & \text{if } 1 \leq s < 2; \\ 0, & \text{if } s \geq 2 \end{cases} \tag{3.2}$$

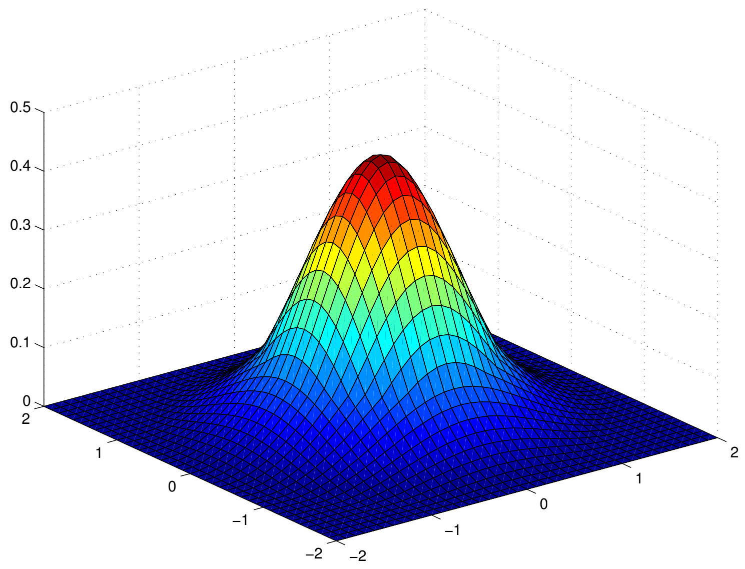


Figure 3.2: 3D plot of the cubic spline smoothing kernel described by equations 3.1 and 3.2.

There are a lot of ad-hoc parameters and constants that can be tweaked in an SPH simulation. Many different configurations have been tried, but here is a list of some reasonable default values for the parameters that turned out to work pretty well in this particular setup.

SPH Parameters:

Initial separation:	r_0	$\approx$	$0.01m$
Gravity:	g	$=$	$9.81m/s^2$
Height of fluid:	H	$=$	$0.2m$
Length scale:	L_0	$=$	$0.06m$
Velocity scale:	V_0	$=$	$\sqrt{2gH}$
Smoothing length:	h	$=$	$1.55r_0$
Velocity of sound:	c	$=$	$\sqrt{200gH}$

Several equations of state were tried out. The equation of state determines the stiffness or incompressibility of the fluid and has an important effect on the stability of the algorithm. For most experiments the usual formula given in equation 2.17 was used.

As mentioned in chapter 2, equation 2.15 could be used to evaluate the density at the center of each particle, but there are better ways to do it. First of all, an extra loop over all particles would be needed to evaluate the densities in this way. In addition, equation 2.15 will produce incorrect results when simulating fluids with free surfaces since the particles near the surface boundary will be assigned densities that are too small. A better way to update the density of particle i is to use the relation

$$\frac{d\rho_i}{dt} = -\rho\nabla \cdot \mathbf{v}_{ij} = \sum_j m_j \mathbf{v}_{ij} \cdot \nabla W(\mathbf{r}_{ij}, h), \tag{3.3}$$

where $\mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j$ and $\mathbf{v}_{ij} = \mathbf{v}_i - \mathbf{v}_j$. The density derivative is then integrated each time step. In this way the density becomes directly available for use in the calculation of pressure forces without an extra iteration over all particles each frame. The equation of motion applied was the standard one as suggested by Monaghan [22].

$$\frac{d\mathbf{v}_i}{dt} = -\sum_j m_j \left(\frac{P_i}{\rho_i^2} + \frac{P_j}{\rho_j^2} + \Pi_{ij} \right) \nabla W(\mathbf{r}_{ij}, h) + \mathbf{F}_i, \tag{3.4}$$

where $\mathbf{F}_i$ is the body force per unit mass acting on particle i, which in this case is gravity. The formula for artificial viscosity Π_{ij} between two particles i and j, also suggested by Monaghan [22], takes the following form.

$$\Pi_{ij} = \begin{cases} \dfrac{-\alpha c \tilde{\mu}_{ij} + \beta \tilde{\mu}_{ij}^2}{\tilde{\rho}_{ij}}, & \text{if } \mathbf{v}_{ij} \cdot r_{ij} < 0; \\ 0, & \text{otherwise.} \end{cases} \tag{3.5}$$

$$\tilde{\mu}_{ij} = \frac{h \mathbf{v}_{ij} \cdot \mathbf{r}_{ij}}{\mathbf{r}_{ij}^2 + 0.01h^2} \tag{3.6}$$

The solid boundaries are represented by a set of specialized boundary particles exerting a Lennard-Jones force per unit mass on the fluid particles according to

$$f(r) = D\left(\left(\frac{r_0}{r}\right)^{p_1} - \left(\frac{r_0}{r}\right)^{p_2} \right) \frac{\mathbf{r}}{r^2} \tag{3.7}$$

where r is the distance between particles, $p_1 = 4$, $p_2 = 2$ and $D = 5gH$.

Implementation

The particles are initially placed on a rectangular 2D grid of cells with side lengths $2h$. Since the smoothing kernel is zero outside a radius of $2h$ only particles in the same and the neighboring cells contribute to the properties of a particle in any given cell. Each cell contains a linked list of free particles and a linked list of boundary particles. After updating the position of each particle a test is performed to see if it has passed on into a neighboring cell. If it has, the linked lists of both cells are updated as appropriate. The *Leap-frog* scheme is used for numerical integration of the density and velocity differential equations. This gives second order accuracy of integration without having to evaluate the density and velocity derivatives more than once per time step. In successive iterations, the variables are advanced by a time step Δt according to

$$\begin{aligned} \mathbf{v}\left(t+\tfrac{\Delta t}{2}\right) &= \mathbf{v}\left(t-\tfrac{\Delta t}{2}\right) + \Delta t \frac{d\mathbf{v}}{dt}(t), \\ \rho\left(t+\tfrac{\Delta t}{2}\right) &= \rho\left(t-\tfrac{\Delta t}{2}\right) + \Delta t \frac{d\rho}{dt}(t), \\ \mathbf{x}(t+\Delta t) &= \mathbf{x}(t) + \Delta t \mathbf{v}\left(t+\tfrac{\Delta t}{2}\right), \end{aligned} \tag{3.8}$$

In the above formula $\frac{d\mathbf{v}}{dt}(t)$ and $\frac{d\rho}{dt}(t)$ depends on the velocity $\mathbf{v}(t)$ and density $\rho(t)$ of each particle at time t. These values are thus *predicted* using the following formulas:

$$\begin{aligned} \mathbf{v}(t) &= \mathbf{v}\left(t-\tfrac{\Delta t}{2}\right) + \frac{\Delta t}{2}\frac{d\mathbf{v}}{dt}\left(t-\tfrac{\Delta t}{2}\right), \\ \rho(t) &= \rho\left(t-\tfrac{\Delta t}{2}\right) + \frac{\Delta t}{2}\frac{d\rho}{dt}\left(t-\tfrac{\Delta t}{2}\right), \end{aligned} \tag{3.9}$$

Due to various factors, one of the biggest problems with the SPH implementation has been to keep the numerical integration stable. To avoid complete numerical blowups, the time step Δt has been kept very small, usually about 0.0002 seconds. The algorithm can be summarized by the following steps:

1. Calculate initial density and velocity gradients using equations (3.3) and (3.4).
2. Calculate initial density, velocity and position. Set $t = \Delta t$.
3. Predict current density and velocity using (3.9).
4. Calculate new density and velocity gradients using (3.3) and (3.4).
5. Update density, velocity and position using (3.8).
6. Render particles using OpenGL.
7. $t = t + \Delta t$.
8. Goto 3

The grid data structure used should theoretically result in $O(n)$ complexity, where n is the number of particles used. To verify this in practice a set of timed test runs were executed, each running 1000 simulation steps with a varying number of particles. The results are shown in figure 3.3 and confirms the expectation of linear complexity. The tests were run on a 933MHz Pentium3 CPU with 256Mb of RAM. However, despite the linear relation between time and number of particles the current performance of the implementation is not very impressive. A couple of hundred particles can be simulated in real-time on the system mentioned, but not much more. Further analysis of the performance and visual results will be presented in the next chapter.

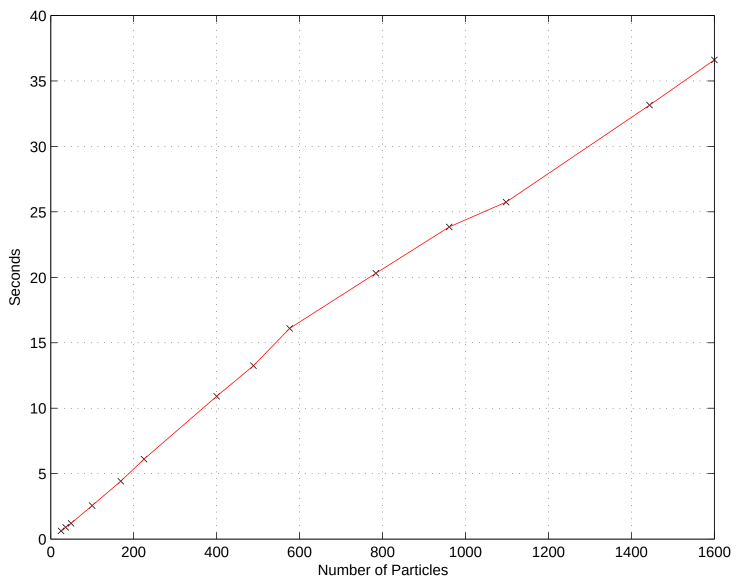


Figure 3.3: Time complexity of the grid based SPH implementation. The relation between computation time and number of particles is clearly linear.

3.2 Particle Systems for Splashes and Drops

As the initial experiments with SPH simulation did not turn out exactly as expected, another approach was tested. In the search for better performance, using a simple, uncoupled particle system seemed like a reasonable thing to try next.

3.2.1 Particle Environment

In order to try out a particle system and its interaction with the surrounding environment, the first thing we need to consider is how to represent this environment in an efficient manner. To get some data to play with a 2D heightfield was first created using Perlin noise functions as described in section 2.4.1. A triangulated 3D surface was then constructed using this heightfield.

One of the most important questions is how to efficiently detect collisions between a particle and its environment, in this case represented by a heightfield and a set of triangles. Testing all triangles for intersection would clearly be much too slow. Several different spatial data structures could be used to improve the situation. I have chosen to implement a grid structure. It is created by splitting up the required portion of space in a Cartesian grid of fixed size cells. The list of triangles is then recursively split up in two pieces along one axis into a set of sublists. These sublists are each split up along the other two axes in the same way. When reaching a “leaf” in the recursion of all axes, the triangles remaining in the current list are added to the grid cell corresponding to that leaf. The resulting space subdivision is illustrated in figure 3.4.

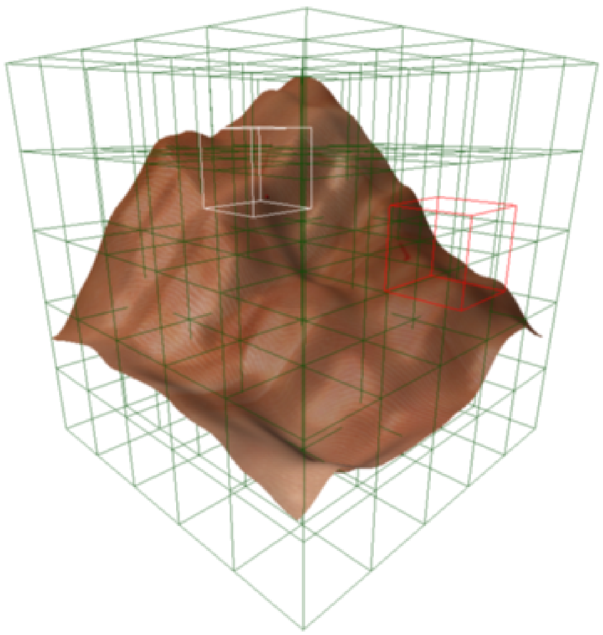


Figure 3.4: Visualization of the spatial grid structure used to speed up particle-surface collision detection.

Triangles overlapping cell boundaries are added to all cells that they touch. The number of triangles ending up in each cell depends on triangle density and grid resolution. Choosing a grid resolution that results in about 10-20 triangles in each cell seems to be a good compromise in this particular case.

Using this structure, the triangles that possibly need to be checked when performing a particle-surface collision detection are quickly located. In short, the origin coordinates of the grid are subtracted from the particle coordinates and the resulting vector is divided by the length of the cell size. This will give the three indices used to find the right cell in the three-dimensional grid matrix.

3.2.2 Collision Detection

Using the grid structure just described, the number of triangles that need to be checked for each particle is narrowed down to about 12-16. When the position of a particle is updated, the ray between its old and new position is checked for intersection with each of the triangles in the current cell. The ray-triangle intersection code used is based on the algorithm described in [1]. It is quite fast, and early rejection tests are used so that the full computation is not executed for non-intersecting triangles. In addition, since the triangles are assumed to represent a surface, it is not very likely that the ray will intersect more than one triangle. This means that the intersection test loop can be exited when the first ray-triangle intersection is found.

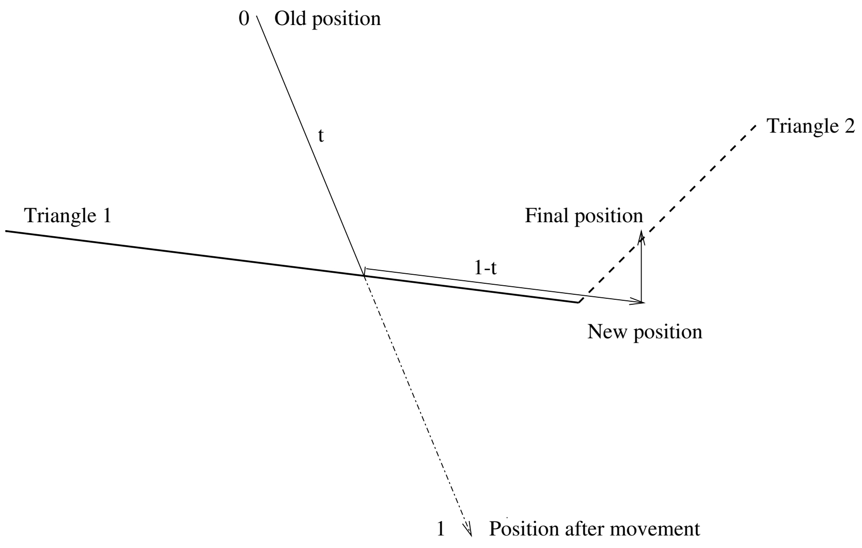


Figure 3.5: Collision detection and response.

When a particle hits a triangle the ray-triangle intersection code returns the fraction t of the original path length that the particle can be moved before hitting the surface. A colliding particle is first moved by this fraction of the original path so that it just touches the surface. It is then moved up a bit along the normal of the surface and after that moved along the surface by the remaining fraction $1 - t$ of the original distance.

However, if the surface has concavities, the second movement should also be tested for collision, or the particle may end up below the surface and eventually pass right through it. This is illustrated in figure 3.5. If the second movement also generates a collision, the resulting deflected path has to be checked in the same way again. For a particle sliding on the surface, this means we will have to loop through and check intersections for all triangles in the current cell at least twice, maybe more. This is a bit too inefficient, but if we only check once, particles will easily slip through the surface. However, we may take advantage of the heightfield created to represent the surface. The real ray-triangle intersection test is only performed once. The elevation of the surface at the new particle location is then calculated by trilinear interpolation of the four heightfield samples closest to the particle. If the particle is located below the surface level it is simply moved up a bit so that it ends up perfectly aligned with the surface. Of course, this is not entirely physically correct, but no part of this simulation really is, and the difference is not noticeable.

Using this combination of ray-triangle intersection and heightfield test works very well. We both get the sliding motion that we are looking for and a stable, efficient implementation that never lets particles slip through the surface no matter how concave it is or how fast they move.

3.2.3 Particle Motion

The motion of the particles in free space is very simple. They are affected only by gravity and accelerated according to Newton's second law, $a = F/m$. When simulating a blood splash for example, a particle emitter is placed at the desired origin of the splash. Particles are then emitted at a constant rate with a velocity vector nearly parallel to the current surface normal. The initial position and velocity are slightly perturbed in a random fashion to make the distribution of particles less uniform and the blood splash spread out.

When the particles hit a surface, friction forces comes into play. The collision response as described above produces completely inelastic collisions. Blood drops never bounce off when they hit the surface. Actually, making drops bounce if they hit a surface with high enough velocity may be something worth trying.

A very simple model for friction has been implemented. When a particle moves on the surface it is affected by a constant force in the opposite direction of its current velocity vector. No consideration is taken to factors such as velocity dependence of the frictional coefficient or variable surface roughness.

3.2.4 Particle Rendering

Shading and rendering a surface represented by a set of point samples in real-time is quite a difficult problem. Ideally, we would like to assign some kind of density distribution to each particle, similar to the one given in equation 2.12. The density of all particles could then be summed up to generate a scalar density field in space. The fluid could be rendered as a fully shaded isosurface to this field. This was the approach taken by Murta and Miller [28] and produces nice results as shown in figure 2.6. Their work was far from reaching real-time performance though. Calculating a density field and generating isosurfaces can be really slow.

The most common way to render a particle system is to render each particle as a possibly animated billboard facing the viewer at all times. This works very well for glowing or transparent effects with a "fuzzy" surface such as fire, smoke or explosions. A fluid existing in free space on the other hand, has a very well-defined surface. To appear realistic it needs to be lit and shaded in order to reproduce visual features such as specular highlights and reflected environment. Despite this, unanimated billboards drawn using a circular texture with alpha smoothed edges was the first method tried to render the particle system. This worked reasonably well for the case when the blood splash was moving in free space but did not look good at all when the drops hit the surface.

After some thought, another way to achieve shading and specular highlights without the need to generate an actual surface was tried. Instead of rendering each particle as a two-dimensional billboard it is rendered as a small sphere in 3 dimensions. The spheres are not very detailed. Since they usually end up quite small on the screen, something along the lines of 100 to 200 triangles is usually enough. The sphere geometry is stored as an OpenGL display list which usually means that they can be rendered very quickly with very limited CPU intervention. The spheres are rendered using gouraud shaded triangles with diffuse and specular lighting.

3.2.5 Blood Trails

When a drop or splash of blood touches a surface it will usually leave a trail of fluid behind as it moves. This effect could of course be implemented by forking off a large number of static particles along the path of each drop. However, more and more static "trail" particles would be accumulated as drops interact with the surface. This would eventually slow down the simulation considerably.

The blood trail effect has instead been implemented by rendering the blood trails directly to the surface texture. The `WGL_ARB_render_texture` OpenGL extension would be ideal for this, but unfortunately it is only available under the Windows OS. Instead, a more portable solution is implemented by using OpenGL `pbuffers`. In short, a `pbuffer` is an offscreen buffer that can be rendered to just like the standard frame buffer. During initialization, a `pbuffer` with the same dimensions as the surface texture is initialized and the original texture image is rendered into this buffer. As a particle moves on the surface during the simulation loop, a circular blood splat is blended into the pbuffer at the location corresponding to the current position of the drop. When all drops have been updated and all splats rendered, the contents of the pbuffer are copied back into the original surface texture using a call to the OpenGL function `glCopyTexSubImage2D`. The surface is then rendered using the updated texture.

Another OpenGL extension, `GL_SGIS_generate_mipmap`, is also enabled to automatically generate new filtered mipmaps when the original texture image is modified by the call to `glCopyTexSubImage2D`. The scaling and filtering of the mipmaps is done entirely on the graphics card on most modern hardware and has a very small impact on the overall performance. By using this method trails after blood particles can be generated very efficiently.

In fact, when there are no moving particles on the surface there will be no performance penalty at all.

3.2.6 Coupling the Particles

The experiments with uncoupled particles turned out quite well. Updating particle states is fast and it looks good enough in many cases. Unfortunately, the system as a unit does not really behave as a fluid physically. This will sometimes produce quite unrealistic results. As an attempt to make things a bit more physically valid, a coupled particle system was tried.

The particles were set up to interact with each other using a Lennard-Jones force function similar to equation 3.7. The shape of the Lennard-Jones curve is shown in figure 3.6 below. As can be seen, the particles strongly repulse each other as they get close to each other, while mildly attracting each other at longer distances. The steep spike of repulsion in the original Lennard-Jones function goes to infinity very fast as the particles approach. If they for some reason get a little bit too close, the resulting force may be huge. This will result in all kinds of stability problems. As an attempt to make things more stable, another kind of inter-particle force was tested. It has the same shape as a Lennard-Jones force on larger distances but uses a cubic spline patch that does not go to infinity as the particles get closer.

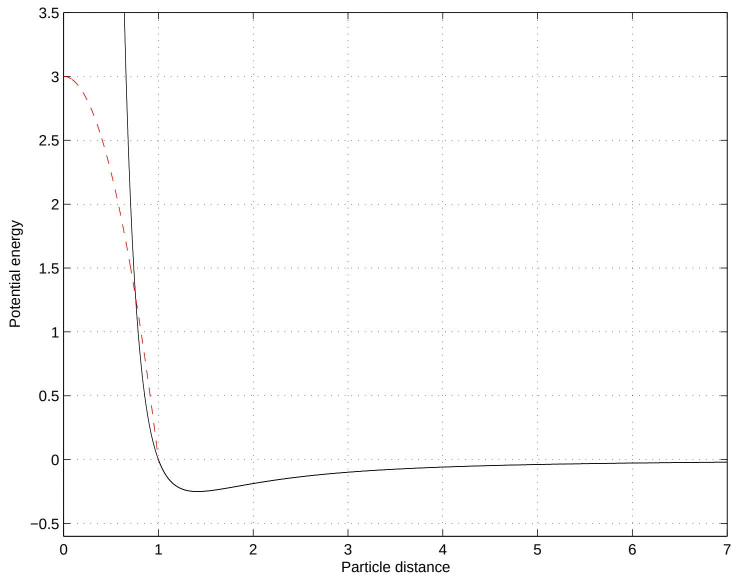


Figure 3.6: Shape of the Lennard-Jones function used for particle interaction. The red, dotted curve shows the spline patch used to avoid very large repulsive forces.

The first version used a very simple $O(n^2)$ approach. As the force on a particle was calculated all other particles were taken into account. Naturally, doing it this way simply becomes too slow even for a particle count as low as about a hundred. Fortunately, as can be seen in figure 3.6, the strength of interaction between particles becomes very small as the distance is increased. In the case of figure 3.6 for example, we could safely ignore the forces between particles that are more than 7 distance units apart.

We thus need a quick way to find the closest neighbors of a particle. This is the same problem as the one that was faced when trying to speed up the SPH implementation. The same solution, a spatial grid structure dividing space into a set of uniform cells also works here. When updating a particle, only particles in the same and the 26 nearest neighbor cells are considered. The cell size is selected so that particles in cells other than the nearest neighbors can be neglected. As particles move, checks are made to see if they have passed on into another cell. If so, the linked lists of particles in each cell are updated.

As a particle is updated, the total force from all neighboring particles is summed up and used to calculate the current acceleration of the particle. The acceleration is then integrated in time using a second order Leap-frog scheme. Several variations of force potential, integration scheme, model of friction etc were tried.

3.3 Blood Dissolving in Fluid

The implementation of a fluid solver based on Stam's algorithms as summarized in section 2.2.2 has been well described in several papers [39, 40, 41]. The current and previous state of the simulation are each stored in a couple of discretized, two-dimensional fields. One scalar field containing the density and one vector field containing the velocity. Both fields are simply stored in a couple of linear arrays of *floats*.

To solve the sparse linear systems that appear in the computation of the diffusion and projection steps a method called *Gauss-Seidel* relaxation is used. It is an implicit, iterative algorithm used to solve systems of the type $\mathbf{Ax} = \mathbf{b}$. Each element of the solution matrix A is updated according to the following formula.

$$A^{k+1}_{i,j} = A^0_{i,j} + a\frac{(A^k_{i-1,j} + A^k_{i+1,j} + A^k_{i,j-1} + A^k_{i,j+1})}{1+4a}, \tag{3.10}$$

where A^0 is the initial state of the matrix at the start of the relaxation process. The constant a is a parameter affecting the convergence rate and needs to be adjusted depending on the size of the time step taken. To take one step towards convergence, equation 3.10 is applied to all elements i, j of the solution matrix A. This process is repeated a number of times until the required accuracy is reached. Stam mentions that using about 20 iterations is usually enough [41]. The effect of the iteration is basically a "diffusion" operation that in this particular case can be used to solve a linear system of equations.

3.3.1 Density Solver

Three main steps are taken to update the density field. The first thing done is to add the contribution from any external sources of density. The density sources for a given frame are simply provided in an array that is initialized by code that calls the solver. For example, in the demo application developed, sources of density can be added by clicking with the left mouse button in the simulation window. Higher densities in this case represent a higher concentration of blood in the otherwise transparent fluid.

In the second step, diffusion is performed by applying Gauss-Seidel relaxation to the density field. The parameter a in equation 3.10 is taken to be $dN^2\Delta t$, where Δt is the current time step, d is a constant of diffusivity and N the resolution of the grid [41]. As the final step, the density is advected by the current velocity field as described in section 2.2.2.

3.3.2 Velocity Solver

The velocity field is updated in much the same way as the density field. Forces, or in other words, sources of velocity are applied instead of density sources. The demo application lets the user manipulate the fluid by creating sources of velocity using the mouse. The diffusion and advection steps are basically identical. However, to satisfy the conservation of mass laws, the final projection step as described in section 2.2.2 needs to be applied to the velocity field. In practice, this step means solving the *Poisson* equation $\nabla^2 q = \rho$. This is another problem on which the Gauss-Seidel solver can be applied.

3.3.3 Visualization

As mentioned, the values of the density field represent the concentration of blood in an otherwise transparent fluid. By default, a grid with resolution 64x64 is used. To visualize the fluid, each grid cell is drawn as a red quad with alpha values at each corner defined by the density at that grid location.

3.4 Smoke

An uncoupled particle system is used to visualize smoke. The same basic particle system code is used for the smoke as for the blood simulation. No collision detection is done though, which makes updating the particles much simpler and faster.

Currently, a very simple model is implemented for the large scale movement of the smoke. Particles are emitted from the surface with a slightly randomized velocity vector. After that, they are simply moved in the opposite direction of the gravity vector, emulating the behavior of a hot gas rising in a colder environment. The movement in the horizontal plane is only affected by resistance from the surrounding medium. The original horizontal velocity is thus decreased during the life time of a particle.

The particles are rendered using a static billboard, rotated so that is always faces the viewer. A color and alpha value is specified for each particle before it is rendered. The constant alpha value depends on the age of the particle. Particles get more transparent as they age. The billboard image is used as a secondary alpha channel. As the particle is rendered the original, constant particle alpha value is multiplied by the alpha value from the billboard texture as shown in figure 3.7.

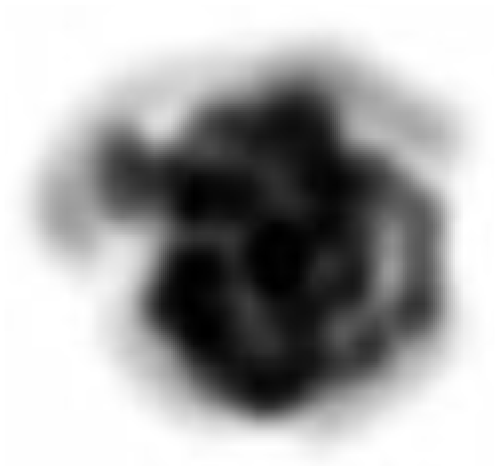


Figure 3.7: An inverted version of the texture bitmap used as alpha channel when rendering smoke particles.

The size of the billboards is also increased as the particles age. This is a simple trick to simulate the process of diffusion of smoke density. In addition to their large scale movement, particle coordinates are also slightly disturbed by a periodical function with varying frequency and amplitude as they move. This makes the rendered billboard images interact and interfere with each other in a constantly varying and pseudo-random pattern, effectively emulating the visual properties of small scale turbulence.

Chapter 4

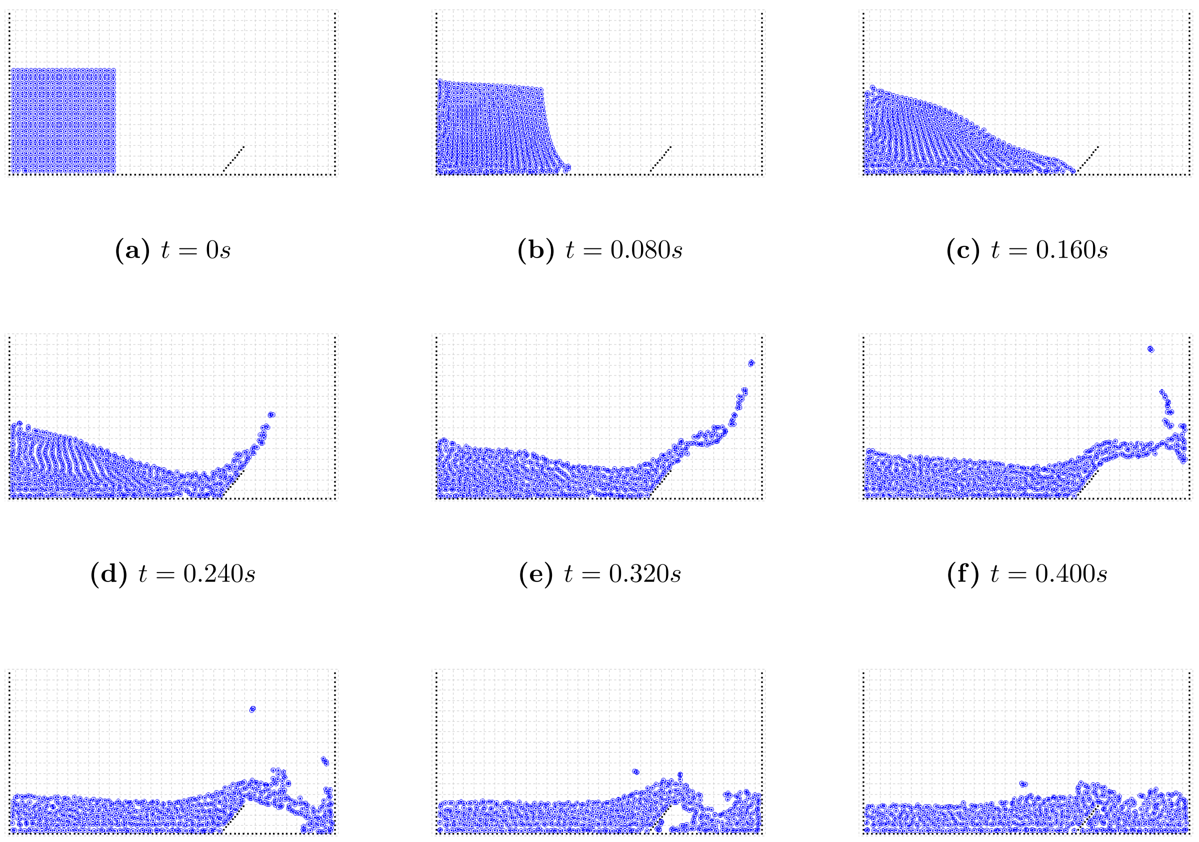
Results and Discussion

This chapter will try to present the results of the experiments described in the previous chapter. Accurately depicting a dynamic, interactive animation using nothing but words and static pictures is not always very successful. Additional screenshots and animations from the demo applications developed can be found on the thesis webpage [3].

4.1 Smoothed Particle Hydrodynamics

The SPH implementation described in section 3.1 did not run very fast, but it did seem to reproduce the basic characteristics of a real fluid in motion quite well. The results of the breaking dam simulation were visually compared to the ones presented in [22]. As far as I can tell, both simulations produced similar fluid behavior on the large scale. Features such as fluid viscosity and mass momentum can be observed quite easily. However, the quantitative accuracy of these effects has not been very well investigated and should not be trusted. It looks like something that behaves similar to water, but the numerical results are definitively not valid for any kind of engineering applications. For visualization purposes though, they may be quite adequate. A set of screenshots from a simulation of a breaking dam with a small downstream obstacle is shown in figure 4.1.

Even if my SPH implementation may produce results that are reasonable enough to be used for visualization, it simply does not run fast enough in its current state. The most basic optimization, i.e. getting rid of the $O(n^2)$ complexity using a grid structure helped a lot, but not enough. The main problem is clearly the stability of the simulation. The time steps in the current implementation must be very small. This means that hundreds of simulation steps need to be executed each frame. If the stability could be improved enough to enable time steps comparable to the wanted frame rate this would definitively be a real-time algorithm. It is not obvious how to improve the situation. Entire Ph.D theses have been written on the subject of SPH stability. The smoothing kernel used has a big impact on the numerical properties of the algorithm. The scheme of integration is also an important factor for stability. I started experimenting with the smoothing kernels and equations suggested by Müller et al. in [26, 27] but did not come up with anything satisfying. Coding an efficient and stable implementation of SPH probably requires a bit of mathematical talent and interest, along with a few "tricks of the trade" best learned from people with a lot of experience in the subject.



(a) $t = 0s$ **(b)** $t = 0.080s$ **(c)** $t = 0.160s$

(d) $t = 0.240s$ **(e)** $t = 0.320s$ **(f)** $t = 0.400s$

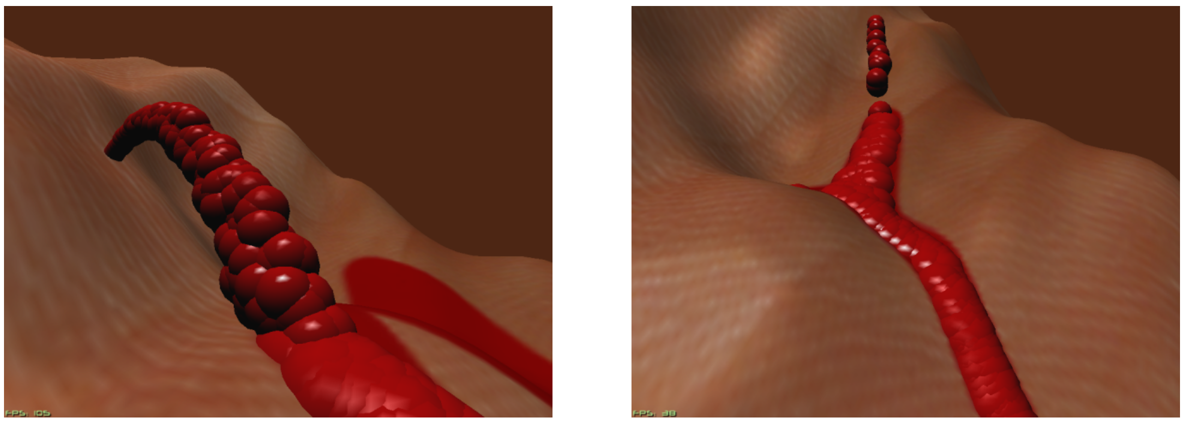
(g) $t = 0.480s$ **(h)** $t = 0.560s$ **(i)** $t = 0.640s$

Figure 4.1: Pictures from an SPH simulation of a breaking dam with a downstream obstacle.

4.2 Particle System Splashes and Drops

Implementing a particle system to simulate blood splashes and drops proved to be a pleasant and intuitive experience compared to SPH. The visual results of the simulation of splashes and flows on a surface are shown in figure 4.2. Although not very realistic, especially not when viewed as a static picture, the resulting animation may still be useful.

Since the particles are completely uncoupled, the system as a whole will behave very unrealistically in many situations. For example, a large splash of blood may float along the surface down into a "valley" in which all particles will end up at the same location on the bottom, basically reducing the size of the fluid body to that of a single particle. The same goes for fluid floating over a bump or hill. Since there is no internal forces of tension or viscosity, the particles will spread out a lot more then they are supposed to. This is definitively a serious problem. However, there are also situations in which the simple uncoupled particle system works pretty well.



(a) A blood splash moving in free air and splashing down on a surface.

(b) A stream of blood peacefully flowing down the orange hills.

Figure 4.2: Screenshots from simulation of blood splashes.

As can be seen in the picture to the left, the trick of rendering fluid particles using 3D spheres as a quick and dirty way to shade the splash of blood is very apparent. This problem is not quite as easy to spot when the splash is animated, but it is still looking much too bumpy. The demo shown in figure 4.2 uses a maximum of 400 particles at a time. Rendering more than 400 spheres starts to put some pressure on the rendering pipeline on a standard video card. Using more particles would probably improve the situation when it comes to the visual flaws generated by rendering the system as a set of spheres. However, using some kind of splatting technique to reconstruct and shade the fluid surface may be more efficient and might produce better results than simply adding more spheres.

To simulate small drops of blood running down a surface a single, slightly larger blood particle was used for each drop. When a particle moves on the surface the collision detection code makes its centerpoint stay just above the surface level. Thus only the portion of the sphere that is above the surface will be visible. This will make the spherical particle look slighly smeared out on the surface, just the way we want it to look. The render to texture technique described in section 3.2.5 is used to make the drop leave a trail of blood on the surface as it moves.

The experiments with coupled particles was quite interesting and almost produced useful results. The general behavior of a system of particles coupled by a Lennard-Jones potential is in most situations much more realistic than the behaviour of an uncoupled system. For example, the fluid wants to maintain a certain volume and will not compress or diverge as easily. However, in the same way as for the SPH simulation, stability problems and small time steps made the simulation too slow. A video showing a coupled particle system in action is available on the thesis webpage [3].

4.3 Blood in fluid

The simulation of blood dissolving in a transparent fluid using Stam's fluid solver algorithms worked out really well. It does however require quite a bit of CPU power. Simulation on a 64x64 grid, including visualization, runs at about 60 frames per second on a 933MHz Pentium3. On a more recent CPU this should be fast enough to be integrated into a real world simulator. If it still does not run fast enough, the simulation could probably be done on a grid of even lower resolution, but rendered at higher detail by interpolating densities in between grid points. A screenshot from a simulated bleeding is shown in figure 4.3 below. The background image and the simulation are both 2D.

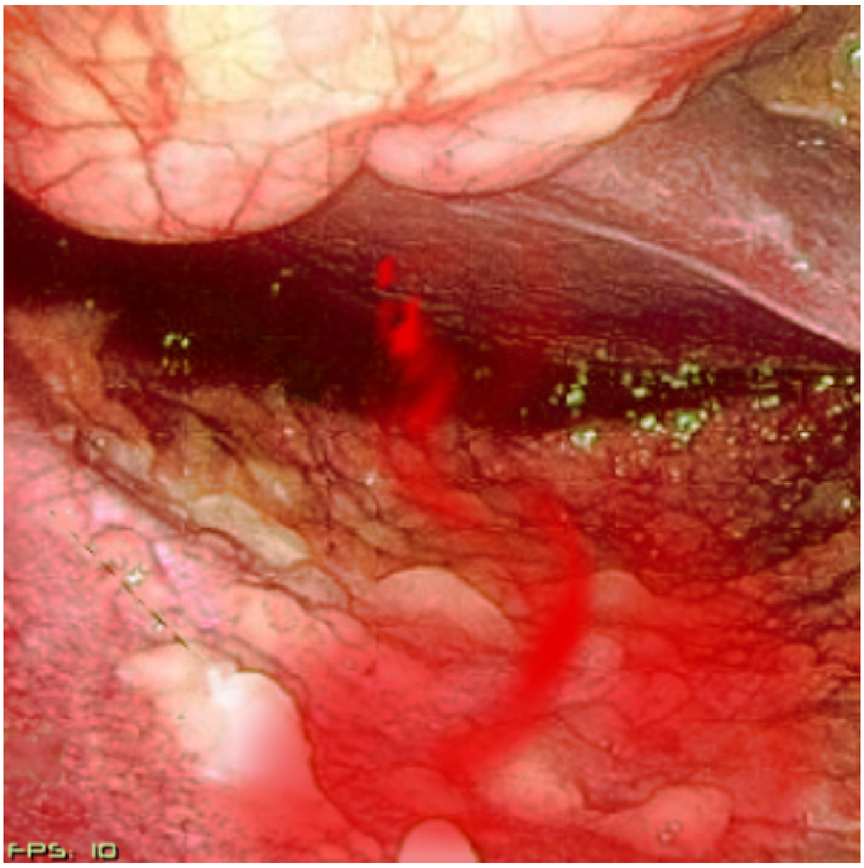
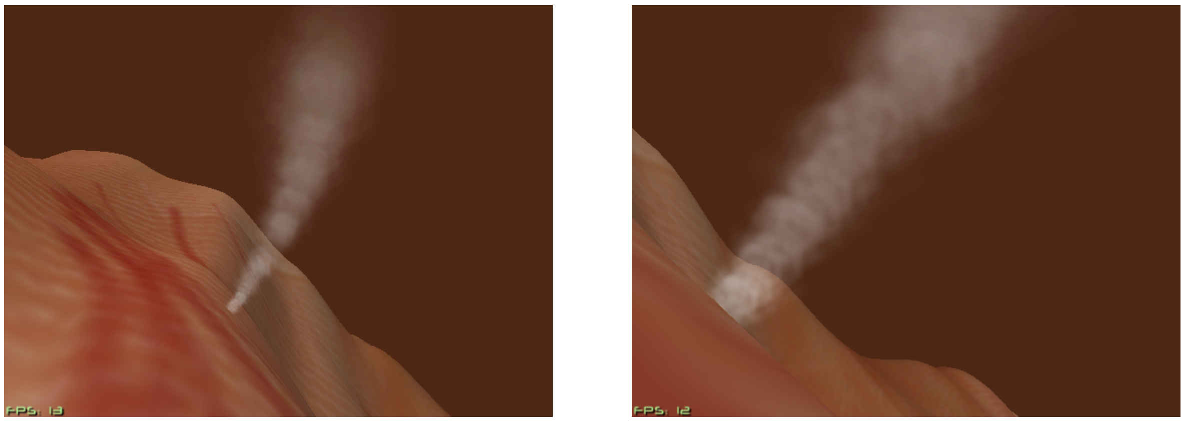


Figure 4.3: Simulation of blood from injured tissue dissolving in a transparent fluid.

A simple demo application has been developed that lets the user initiate a bleeding and manipulate the velocity and density fields using the mouse. It should be quite easy to implement effects such as interaction between the dissolved blood and a surgical instrument by simply letting the instrument affect the velocity field. In order to achieve realistic simulation of interaction between the fluid and various instruments the fields probably needs to be represented and updated in all three dimensions. As an approximation, using a very low grid resolution along the depth axis may work pretty well. If we need to translate and rotate the camera arbitrarily however, a full 3D simulation is probably necessary. This is still too slow to implement on a CPU though, but it may work just fine on a modern GPU.

4.4 Smoke

Despite using a very simple, uncoupled particle system and no collision detection, the smoke effects generated look quite good. The overlapping and interfering billboards emulate small scale turbulence in a visually convincing way. A couple of screenshots from a smoke simulation are shown in figure 4.4.



(a) A smoke source on the surface generates a rising cloud of white smoke. Horizontal air resistance makes the cloud turn upwards.

(b) A closer look. Note the small scale details generated by overlapping billboards. When animated, these nicely emulate the visual properties of small scale turbulence.

Figure 4.4: Screenshots from simulation of smoke.

The large scale movement of the smoke particles is not very realistic though. As mentioned in section 3.4, the particles are simply moved upwards with a constant speed, and slowed down in the horizontal plane to simulate the resistance and velocity diffusion created by the surrounding medium. If more realism is needed, the particles could instead be moved around by a dynamic velocity field updated using Stam's fluid solver. A more physical model for the force generated by the temperature difference between the smoke and the environment is also needed.

A couple of hundred billboards is usually enough to create a nice smoke effect. Updating positions and other particle characteristics on the CPU is cheap and could easily be done on several thousand particles each frame. Rendering all particles on the other hand can be quite slow. When the system is viewed from a distance and each billboard covers only a small portion of the screen area rendering performance is not an issue. When moving close to a smoke system though, each billboard may cover all or big parts of the rendering window. Rendering a couple of hundred screen sized billboards quickly consumes huge amounts of fillrate and slows down animation even on the fastest of GPU:s. This needs to be taken into consideration if using a system like this in real life. Perhaps the size of the billboards could be checked and several large, transparent ones could be combined into a single, less transparent billboard before rendering.

Chapter 5

Conclusions

In this chapter I will try to briefly summarize the most relevant insights that have been the result of my work. A few suggestions regarding possible directions of future work based on the experiences made will also be given. One thing that I have realized is that computational fluid dynamics is an extensive subject. At one point I had almost a hundred different research papers covering large parts of my tiny room, but it still felt as I had barely scratched the surface. As a matter of fact, it still does.

First of all, have the goals of the thesis as specified in the first chapter been achieved? Well, a few separate solutions to the specific problems mentioned have indeed been presented and implemented. Let us take a look at each problem at a time and discuss how useful the suggested solution could be in a real world situation.

5.1 Splashes and Drops

The visual and physical realism of the splash and drop effects could definitively be improved. In a situation where realistic, physically based interaction with a fluid or visual realism is crucial for a valid simulation experience, the methods described in this report are simply not accurate enough.

In other situations however, with proper adjustment and tweaking, I actually believe that the algorithms and methods examined and implemented for free surface fluids could be useful in a real life virtual surgery application. The effects may not be very realistic, but they are still valid enough to provide the user of a simulator with visual cues and feedback that is helpful to understand what is going on in the simulated environment. A blood splash may look and behave more like a bunch of red apples than a real life fluid, but in the context of a virtual surgery simulator it should be pretty obvious that the visual effect represents blood rather than apples. However, this part of the work is the one that was least successful. Alternatives to the suggested and quite naive implementation should definitively be examined. For someone with the numerical and mathematical skills required, implementing a coupled particle system fast enough for practical real-time use should not be impossible. Combining this with GPU based methods for surface generation, such as surface splatting, would probably produce quite nice results. That is the direction I would take to improve this effect.

5.2 Blood in Fluid

The blood in fluid effect is the one described in this report that gets closest to physical reality. Using a high resolution grid in three dimensions should produce quite accurate results, at least visually. A 3D simulation running on the CPU will probably be too slow for use in a real application, unless a multiprocessor system is used and a separate CPU could be dedicated to fluid simulation alone. However, the best way to run this kind of simulations is definitively on a modern GPU. The GPUs released during the second half of 2004 should be fast enough to run a 3D simulation with decent grid resolution in real-time. Implementing a GPU version of Stam's fluid solver would be the next step to improve this effect.

5.3 Smoke

Although not strictly based on physical principles, the smoke simulation looks quite good but also needs some tweaking. The length scale of the current test implementation is not correct for virtual surgery. It looks more like a smoke cloud from a large fire or chimney on a cold day rather than a tiny smoke stream generated by a tissue burning instrument. This could be improved to look better by adjusting diffusion rate, small scale motion frequency etc. Experiments would probably be the quickest way to come up with something useful. As mentioned, if more physical realism is needed the model described could be combined with a real fluid solver updating a velocity field used to move smoke particles around.

Bibliography

[1] T. Akenine-Möller, E. Haines, *Real-Time Rendering*, A K Peters, 2002.

[2] J. D. Anderson, Jr, *Computational Fluid Dynamics, The Basics with Applications*, McGraw-Hill, 1995

[3] L. Andersson, *Thesis Project Web Page, Real Time Fluid Dynamics for Virtual Surgery*, `http://www.dd.chalmers.se/~f99laan/exjobb/`, 2004.

[4] C. Basdogan, C. H Ho , M. A Srinivasan, *Simulation of Tissue Cutting and Bleeding for Laparoscopic Surgery Using Auxiliary Surfaces*, Precedings of the Medicine Meets Virtual Reality Conference, 38-44, January 1999.

[5] H. K, Çakmak, U. Kühnapfel, *Animation and Simulation Techniques for VR-Training Systems in Endoscopic Surgery*, Computer Animation and Simulation 2000, Proceedings of the Eleventh Eurographics Workshop, 173-185, 2000.

[6] R. Fedkiw, J. Stam and H. W. Jensen, *Visual Simulation of Smoke*, In SIGGRAPH 2001 Conference Proceedings, Annual Conference Series, 15-22, August 2001.

[7] N. Foster, D. Metaxas, *Realistic Animation of Liquids*, Graphical Models and Image Processing, 58(5), 471-483, 1996.

[8] N. Foster, D. Metaxas, *Modeling the motion of a hot, turbulent gas*, Proceedings of the 24th annual conference on Computer graphics and interactive techniques, 181-188, 1997.

[9] P. Fournier, A. Habibi, P. Poulin, *Simulating the Flow of Liquid Droplets*, Proceedings of Graphics Interface 98, 133-142, 1998.

[10] R. A, Gingold, J. J Monaghan, *Smoothed Particle Hydrodynamics: Theory and Application to Non-Spherical Stars.* Monthly Notices of the Royal Astronomical Society, 181, 375-389, 1977.

[11] M. J, Harris. *Fast Fluid Dynamics Simulation on the GPU*, GPU Gems, Addison-Wesley, 2004.

[12] T. Holtkämper, *Real-time Gaseous Phenomena - A Phenomenological Approach to Interactive Smoke and Steam*, Proceedings of the 2nd international conference on Computer graphics, virtual Reality, visualisation and interaction in Africa, 25-30, 2003.

[13] L. S Jensen, R. Golias, *Deep-Water Animation and Rendering*, Gamasutra, `http://www.gamasutra.com/gdce/2001/jensen/jensen_01.htm`, September 2001.

[14] M. Jonsson, A. Hast, *Animation of Water Droplet Flow on Structured Surfaces*, SIGRAD2002, The Annual SIGRAD Conference, 17-22, November 2002.

[15] K. Kaneda, S. Ikeda, H. Yamashita, *Animation of Water Droplets Moving Down a Surface*, The Journal of Visualization and Computer Animation, 10 (1), 15-26, 1999.

[16] M. Kass, G. Miller, *Rapid, Stable Fluid Dynamics for Computer Graphics*, SIGGRAPH 1990 Conference Precedings, pp. 49-57, 1990.

[17] J. Křivánek, *Representing and Rendering Surfaces With Points*, Postgraduate Study Report no. DC-PSR-2003-03. Dept. of Computer Science and Engineering, CTU Prague, 2003.

[18] Y. Liu, X. Liu, E. Wu, *Real-Time 3D Fluid Simulation on GPU with Complex Obstacles*, Computer Graphics and Applications, 12th Pacific Conference on (PG'04) October 06-08, 2004

[19] W. Lorensen, H. Cline, *Marching cubes: A high resolution 3D surface construction algorithm*, Proceedings of the 14th annual conference on Computer graphics and interactive techniques, 163-169, 1987.

[20] L. B Lucy, *A Numerical Approach to the Testing of the Fission Hypothesis*, Astronomical Journal, 82, 1013-1024, 1977.

[21] J. J Monaghan, *Smoothed Particle Hydrodynamics*, Annual Review of Astronomy and Astrophysics, 30, 543-574, 1992.

[22] J. J Monaghan, *Simulating Free Surface Flows with SPH*, Journal of Computational Physics, 110, 399, (1994)

[23] J. J Monaghan, *SPH without a Tensile Instability*, Journal of Computational Physics, 159(2), 290-311, 2000.

[24] J. P Morris, *Modeling Low Reynolds Number Incompressible Flows Using SPH*, Journal of Computational Physics, 136, 214-226, 1997

[25] J. P Morris, *Simulating Surface Tension with Smoothed Particle Hydrodynamcs*, International Journal for Numerical Methods in Fluids, 33(3), 333-353, 2000.

[26] M. Müller, D. Charypar, M. Gross, *Particle-Based Fluid Simulation for Interactive Applications*, Proceeding of 2003 ACM SIGGRAPH Symposium on Computer Animation, 154-159, 2003.

[27] M. Müller, S. Schirm, M. Teschner, *Interactive Blood Simulation for Virtual Surgery Based on Smoothed Particle Hydrodynamics*, Journal of Technology and Health Care, 12(1), 25-31, February 2004.

[28] A. Murta, J. Miller, *Modelling and Rendering Liquids in Motion*, Proceedings of WSCG'99, 194-201, February 1999.

[29] F. Neyret, *Advected Textures*, Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, 147-153, 2003.

[30] J. F O'Brien, J. K Hodgins, *Dynamic Simulation of Splashing Fluids*, Proceedings of Computer Animation '95, 198-205, April 1995.

[31] V. Pascucci, *Isosurface Computation Made Simple: Hardware Acceleration, Adaptive Refinement and Tetrahedral Stripping*, Joint Eurographics - IEEE TVCG Symposium on Visualization (VisSym), 293-300, 2004.

[32] K. Perlin, *An Image Synthesizer*, ACM Computer Graphics (SIGGRAPH '85), 19(3), 287-296, July 1985.

[33] L. Raghupathi, *Simulation of Bleeding and other Visual Effects for Virtual Laparoscopic Surgery*, Master thesis, University of Texas at Arlington, 2002

[34] W. T. Reeves, *Particle Systems - A Technique for Modeling a Class of Fuzzy Objects*, Computer Graphics, 17 (3), 359-376, 1983.

[35] T. M Roy, *Physically Based Fluid Modeling Using Smoothed Particle Hydrodynamics*, Master thesis, Willamette University, 1988

[36] B. Schlatter, *A Pedagogical Tool Using Smoothed Particle Hydrodynamics to Model Fluid Flow Past a System of Cylinders*, Dual MS Project, Oregon State University, June 11th, 1999.

[37] J. Stam and E. Fiume. *Turbulent Wind Fields for Gaseous Phenomena*, In SIGGRAPH 93 Conference Proceedings, 369-376, August 1993.

[38] J. Stam and E. Fiume. *Depiction of Fire and Other Gaseous Phenomena Using Diffusion Processes*, In SIGGRAPH 95 Conference Proceedings, 129-136, August 1995.

[39] J. Stam, *Stable Fluids*, SIGGRAPH 99 Conference Proceedings, Annual Conference Series, 121-128, August 1999.

[40] J. Stam, *Interacting with Smoke and Fire in Real Time*, Communications of the ACM, 43 (7), 76-83, 2000

[41] J. Stam, *Real-time Fluid Dynamics for Games*, Precedings of the Game Developer Conference, March 2003.

[42] X. Wei, W. Li, K. Mueller, A.Kaufmann, *Simulating fire with texture splats*, Proceedings of the conference on Visualization '02, 227 - 235, 2002.

[43] X. Wei, W. Li, K. Mueller, A.Kaufmann, *The Lattice-Boltzmann Method for Simulating Gaseous Phenomena*, IEEE Transactions on Visualization and Computer Graphics, 10(2), 164-176, March 2004.

[44] E. Wu, Y. Liu and X. Liu. *An Improved Study of Real-Time Fluid Simulation on GPU.* Computer Animation & Virtual World, 15(3-4), 139-146, July 2004.

[45] J. Zátonyi, R. Paget and G. Székely and M. Bajka, *Real-time Synthesis of Bleeding for Virtual Hysteroscopy*, Procs. of the 6th International Conference on Medical Image Computing and Computer-Assisted Intervention, Vol 1, 67-74, 2003

[46] M. Zwicker, H. Pfister, J. van Baar, M. Gross, *Surface Splatting*, Proceedings of SIGGRAPH 2001, 371-378, 2001.