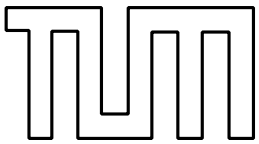


Technische Universität München

Fakultät für Informatik

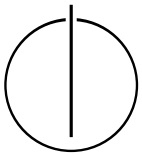
Masterthesis in Informatik

Partikelbasierte Echtzeit-Fluidsimulation

Realtime particle-based fluid simulation

Stefan Auer

Supervisor:	Prof. Dr. Rüdiger Westermann
Advisor:	Dr. Jens Krüger
Submission Date:	TODO



I assure the single handed composition of this master thesis only supported by declared resources

Munich TODO: date

_____

Abstract

Fluid simulation based on smoothed particle hydrodynamics (SPH) is a practical method for the representation of liquids in interactive applications like virtual surgical training or computer games. In recent years various papers introduced ideas for both, the SPH simulation and its visualization. This thesis explains detailed a straightforward CPU-executed implementation of the simulation, as well as an entirely GPU-executed visualization technique using isosurface raycasting, which allows the realtime rendering of multiple refractions. Theoretical foundations and alternative techniques are presented where it seems appropriate.

Fluidsimulation basierend auf Smoothed Particle Hydrodynamics (SPH) stellt eine praktikable Methode zur Repräsentation von Flüssigkeiten in interaktiven Anwendungen wie virtuellem Operationstraining oder Computerspielen dar. In den letzten Jahren wurden vielfältige Ideen sowohl für die SPH Simulation selbst, als auch für deren Visualisierung vorgestellt. Diese Arbeit erklärt detailiert sowohl eine möglichst direkt umgesetzte CPU Implementierung der Simulation, als auch eine rein auf der GPU laufende Visualisierungstechnik basierend auf Raycasting der Iso-Oberfläche, welche die Echtzeitdarstellung mehrfacher Refraktionen erlaubt. Wo es angebracht erscheint, wird auf die theoretischen Grundlagen und auf alternative Techniken eingegangen.

Table of contents

1 Introduction 9

1.1 Motivation 9

1.2 How to simulate fluids 10

1.3 Related work 10

1.4 Used techniques 12

2 Fluid simulation 13

2.1 Chapter overview 13

2.2 Basics of fluid mechanics 13

2.3 Basics of smoothed particle hydrodynamics 19

2.4 Particle based, mathematical model of fluid motion 21

2.5 Smoothing kernels 24

2.6 Basic simulation algorithm 26

2.7 Implementation 27

2.8 Environment and user interaction 30

2.9 Multithreading optimisation 31

2.10 Results 33

2.11 Further work and outlook 34

3 Visualisation 37
3.1 Chapter overview 37
3.2 Target graphics API 37
3.3 Direct particle rendering 39
3.4 Isosurface rendering with marching cubes 41
3.5 GPU-based isosurface ray-tracing 46
3.6 GPU-based volumetric density field construction 66
3.7 Alternatives and further work 69
4 Conclusion 71
Appendix 73
References 73
Glossary 75
List of figures 76
Derivation of the gradient and Laplacian of the smoothing kernels 78

1 Introduction

1.1 Motivation

Fluids like liquids and gases are ubiquitous parts of the environment we live in. For instance we all know how it looks like when milk gets filled into a drinking glass. In realtime computer graphics, where we traditionally try to reproduce parts of our world as visually realistic as possible, it is unfortunately hard to simulate such phenomena. Computational fluid dynamics is a relatively old and well known research topic, but most applications (like i.e. in aerodynamics research) aim at results that are as accurate as possible. Therefore, the simulations are mostly calculated offline and realtime visualisation – if at all - is used only to render precomputed data sets, if at all.

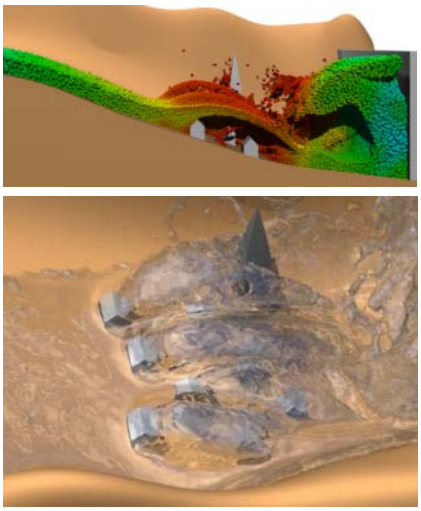


Figure 1: Example for offline simulation
Source:[APK07]

Figure 2: Example for realtime simulation
Source: [MCG03]

Realtime applications that do allow the user to interact with authentically (but not necessarily accurately) simulated and rendered fluids (like i.e. water) are rare today rare. For all types of virtual realities, like surgical training environments or computer games, there is always demand to cover more aspects of our world and so realtime simulation and rendering of fluids is an interesting field of study. In 2003, Müller, Charypar and Gross sparked additional interest in realtime fluid simulation, with a paper that proposed a relatively simple, particle-based fluid-model, which fits well for realtime applications [MCG03]. Since then many different aspects of realtime particle-based fluid simulation where covered in a couple of papers from authors around the world. This thesis gives an overview on the topic, as it discusses my implementation of a particle-based fluid simulation and a suitable water renderer.

1.2 How to simulate fluids

In the 19th century Claude Navier and George Stokes created the fundamentals of modern fluid dynamics as they formulated the well-known Navier-Stokes equations. With these equations, which describe the conservation of momentum, together with two additional equations for mass and energy conservation, it is possible to simulate the fluid flow. As the formulas tend to get very complicated for less common fluids, they are mostly written for Newtonian fluids, which include a variety of common liquids and gases (water, air...).

Simulations apply numerically methods to solve the - in most cases - resulting non-linear partial differential equations. One common way to do this is to treat the fluid as a continuum, discretize the spatial domain into a grid and use finite differences or the finite volume method. In the literature grid-based fluid models are called Eulerian models. For the use within virtual environments grid-based methods, as a matter of principle, have the drawback of a bounded simulation space.

In contrast, particle-based methods (in literature: Lagrangian model, from Lagrangian mechanics) represent the fluid as a discrete set of particles and simulate the fluid flow through solving the particle dynamics. For realtime applications this results in some advantages versus grid-based methods:

- simpler calculation (mass conservation can be omitted, convective term can be omitted, cp. [MCG03])
- no numerical diffusions in the convection terms (diffusion directions are not influenced by the grid layout)
- surface reconstruction is likely to be easier
- fluid can spread freely in space (no boundary through the grid)

For these reasons (especially the last) this thesis focuses on a Lagrangian method based on smoothed particle hydrodynamics (SPH) [Mon05], which became very popular for this kind of applications. The idea behind SPH is that every particle distributes the fluid properties in its neighbourhood using radial kernel functions. To evaluate some fluid property at a given point one must simply sum up the properties of the neighbouring particles, weighted with the appropriate smoothing function.

1.3 Related work

The first studies of smoothed particle hydrodynamics where made in 1977 by Gingold and Monaghan (who coined the term) [GM77] and independently by Lucy [Luc77]. Its first usages took place mainly in the astronomy sector to simulate large scale gas dynamics, but later it also has been applied to incompressible flow problems like beach wave simulation, sloshing tanks and bow waves of ships.

While in realtime computer graphics first the Eulerian approach was favoured, Müller, Charypar and Gross [MCG03] where one of the first who showed, that a SPH based Lagrange method also suits very well to interactive applications. Afterwards many papers adapted the SPH approach for realtime fluid simulation and brought different enhancements and extensions:

- [KC05] proposes to avoid the particle neighbourhood problem by sampling the fluid properties from grids that sum up the weighted properties from all particles
- [KW06] compares the performance of an octree-based (linear time for neighbour search, but large costs for the update of the structure) versus a “staggered grid”-based solution to the neighbour problem
- in [MST04] Müller et al. show how particle-based fluids can interact with deformable solids
- [AIY04] sketches how to use a CPU generated neighbour map so that the property summation for each particle can be handled on the GPU, which reaches twice the performance of their CPU only simulation
- [Hei07] uses the Ageia PhysX engine (one of its developers is Matthias Müller) for a SPH-based simulation of smoke

A complete fluid simulation for interactive applications, however, is not only made up of the movement part. The realtime simulation and rendering of the optical appearance is important just as well. In recent years many papers with relevance for this topic where published, even though some of them do not target liquid visualisation in particular:

- [MCG03] suggests direct point splatting of the particles or marching cubes rendering [LC87] of the isosurface (which implies the creation of a volumetric density field for each frame)
- [KW03] presents a GPU executed raycaster for volumetric scalar fields; in combination with a efficient method for building the density field on the GPU this way the isosurface could be visualized
- [CHJ03] introduces iso-splatting, a point-based isosurface visualisation technique; same as with [KW03] applies here
- [Ura06] demonstrates a GPU version of the marching tetrahedra algorithm (variation of marching cubes); same as with [KW03]
- [KW06] uses a 2.5D “carped visualisation” for the special case of rivers and lakes
- [MSD07] introduces “screen space meshes”, which avoids creation of a discrete density field

1.4 Used techniques

The goal of this thesis is to provide a realtime application that simulates a water-like liquid in a form that is “believable” in terms of movement behaviour and optical appearance. The SPH simulation, therefore, focuses not on physical accuracy. It’s a straightforward implementation of the lightweight SPH model presented in [MCG03], optimized to run on actual multi-core consumer CPUs. To speed up the neighbour search it stores the particles according to their position in a dynamic grid, with a cell size equivalent to the maximal radius of support. The particle interactions are evaluated directly on pairs of particles (simultaneous for both particles). Chapter 0 discusses the theoretical foundations and the implementation details of the simulation.

For visualisation three techniques are provided: The first directly renders the particles as point sprites, which is mainly useful for debug and tuning of the fluid behaviour. The second, which is nearly entirely CPU-based, uses the marching cubes algorithm to construct a triangle mesh representing the isosurface. This technique was implemented to experiment with efficient density field construction methods and to test how well a marching cubes- / triangle-based approach fits for the purpose of liquid visualisation. The last and most sophisticated technique uses the GPU to construct a volumetric density field within a 3D texture and renders the isosurface directly with a ray-tracing shader. The ray-tracing enables the visualisation of effects like multiple refractions and reflections, which are characteristic for the optical appearance of liquids. Chapter 3 explains each visualisation technique in detail.

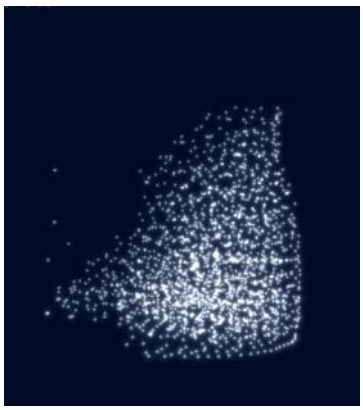


Figure 3: Sprite visualisation

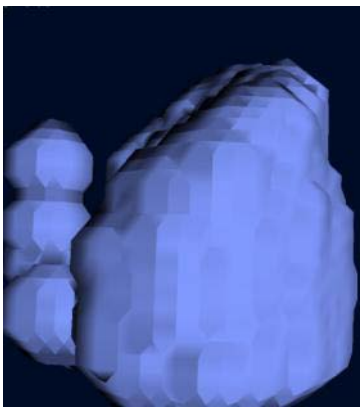


Figure 4: Marching cubes visualisation

Figure 5: GPU ray-tracing visualisation

2 Fluid simulation

2.1 Chapter overview

This chapter discusses realtime particle based simulation of fluid movement, even though other aspects of fluid behaviour (i.e. heat convection) could also be described with the methods presented here. The simulation written for this thesis shall provide a solid basis for the development of realtime visualisation methods for particle based liquid simulations and comprehensible exemplify the implementation of fluid simulations for interactive applications. Therefore, its focus is not on physical accuracy or cover of as much scenarios as possible, but on a clear, simple and highly efficient implementation.

The description of realtime fluid simulation based on SPH starts with an introduction to some basic concepts of fluid mechanics in 2.2. This subchapter clarifies the meaning of the most important simulation quantities and illustrates the differences between the Eulerian and Lagrangian point of view, while it briefly derives the Navier-Stokes equation from Newton's second law of motion. It should not be understood as mathematic derivation, but rather as an attempt to help the reader to comprehend the meaning of the equation.

After the introduction to fluid mechanics in general, 2.3 explains the core concepts of smoothed particle hydrodynamics, which are applied to the Navier-Stokes equation in 2.4 to develop the mathematical model on which the fluid simulation is based. After a short introduction of the used smoothing kernels in 2.5, the conceptual simulation algorithm is presented in 2.6, which is the basis of the implementation that is discussed in 2.7.

In 2.8 the implementation is made capable to interact with the environment and the user, while 2.9 discusses how multithreading could be used to gain more performance on today's CPUs. The results section 2.10 shows how well the final program fulfils its task in terms of performance and simulation quality. 2.11, the last section in the chapter, discusses further improvements that could be made to the existing simulation and gives an outlook on technologies, from which the realtime simulation of fluids could benefit.

2.2 Basics of fluid mechanics

Fluid mechanics normally deals with macroscopic behaviour at length and time scales where intermolecular effects are not observable. In this situation fluids can be treated as continuums where every property has a definite value at each point in space. Mathematically this can be expressed through functions that depend on position and time (i.e. vector or scalar fields). Properties are macroscopic observable quantities that characterize the state of the fluid.

Fluid quantities

The most relevant properties for the movement of fluids are mass, density, pressure and velocity. The mass m specifies "how much matter there is" and is relevant for the inertia of the fluid. The mass density ρ measures the mass per volume and is defined as:

$$\rho \equiv \lim_{\Delta V \to L^3} \frac{\Delta m}{\Delta V} \tag{2.1}$$

L: very small length, but significant greater than the molecule spacing; V: volume

Pressure p is a scalar quantity that's defined as the force F_n acting in normal direction on a surface A (normal stress): $p \equiv \lim_{\Delta A \to 0} \frac{\Delta F_n}{\Delta A}$. Differences in the pressure field of a fluid (= force differences) result in a flow from areas of high to areas of low pressure, while in regions with constant pressure those forces are balanced.

The velocity $\mathbf{v}$ is a measure for how fast and in which direction the fluid passes a fixed point in space. It is perhaps the most important property of the fluid flow. The velocity field effects most other properties either directly (i.e. dynamic pressure) or indirectly (i.e. because of advection). In viscous fluids (all real fluids are viscous to some amount) it is also relevant for the viscosity forces, which are together with pressure forces the most relevant fluid forces.

Viscosity compensates the differences in flow velocity over time (comparable to friction). In case of a fluid with a "constant" viscosity (later more on this topic) it is a measure for how much momentum is transferred between adjacent regions with different flow speeds and is thereby responsible for shear stress (tangential force on a surface). Viscosity as a constant is stated as dynamic viscosity η (when the result is a force) or kinematic viscosity $\nu = \frac{\eta}{\rho}$ (when the result is acceleration).

Surface tension σ is the last cause of forces that we deal with. It is a property of the surface of the fluid (the border to another immiscible fluid, a solid or vacuum) that is relevant for the size of the forces that try to minimize the area and curvature of the surface. A simple explanation for the cause of surface tension is that the cohesive forces (attractive forces between molecules of the same type) between molecules on the surfaces are shared with less neighbour molecules than in the inner of the fluid, which results in a stronger attraction of the molecules on the surface. It is mentioned here for completeness although it is not further discussed in the basics subchapter (we will deal with it later in 2.4).

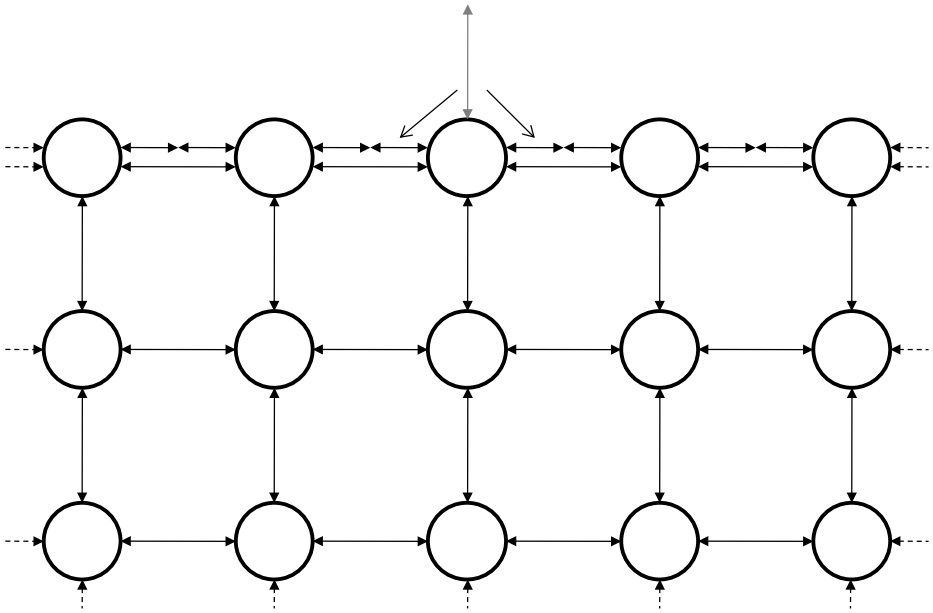


Figure 6: Cause of surface tension

From Newtonian- to fluid mechanics

Now that we know the meaning of most magnitudes, let's see how the motion of a fluid could be described mathematically. Let's start with Newton's second law:

$$\mathbf{F} = m\mathbf{a} \tag{2.2}$$

Note: vectors are written in bold ($\mathbf{a}$), scalars in italics (m).

It states that the acceleration $\mathbf{a}$ of an object depends on its mass m and the force $\mathbf{F}$ that acts on it. This could also be interpreted as conservation of momentum: Without external forces ($\mathbf{F} = 0$) there is no change of velocity ($\mathbf{F} = 0 \Rightarrow \mathbf{a} = \frac{\mathrm{d}\mathbf{v}}{\mathrm{d}t} = 0$) and the momentum $\mathbf{p} = m\mathbf{v}$ stays constant.

In classical (Newtonian) dynamics Newton's second law is usually interpreted from the Lagrangian point of view, meaning that a moving object is observed. With fluids this would mean that the observation area follows the fluid flow, so that always one and the same "amount of fluid" is being watched. Alternatively in the Eulerian point of view the area of observation is locally fixed, so that the fluid passes by and the watched amount of fluid may be a different one at each moment. The Eulerian observer, therefore, not only sees changes due to variances in the currently watched amount of fluid, but also changes due to the fact that the watched amount of fluid may be a different one every moment.

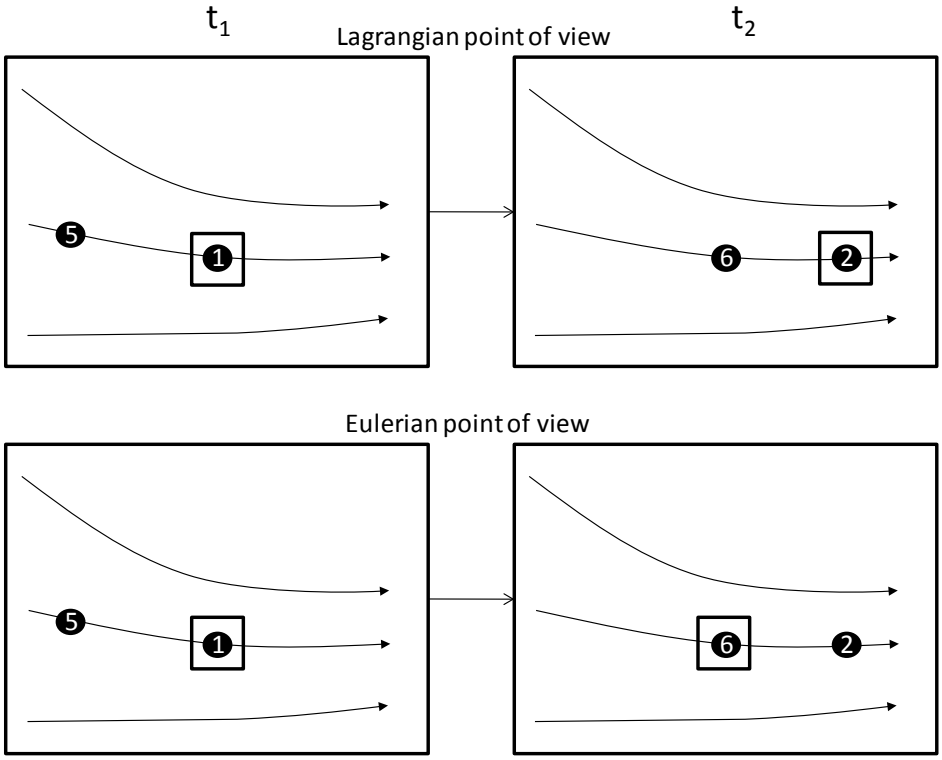


Figure 7: Lagrangian versus Eulerian point of view

In an Eulerian description (which is more common in classical fluid dynamics) the acceleration, therefore, must be a special time derivative of the velocity that takes into account the movement of currents in fluids in both of it forms: Diffusion and advection (together: convection). It is called substantial derivative (synonyms: substantive d., convective d., material d.) and defined as follows:

$$\frac{\mathrm{D}\phi}{\mathrm{D}t} = \frac{\partial\phi}{\partial t} + \mathbf{v}\cdot\nabla\phi = \frac{\partial\phi}{\partial t} + u\frac{\partial\phi}{\partial x} + v\frac{\partial\phi}{\partial y} + w\frac{\partial\phi}{\partial z} \tag{2.3}$$

written in Cartesian coordinates in three dimensions; ∇: del operator; u, v, w: components of velocity; x, y, z: components of position; ϕ: an arbitrary quantity (vector or scalar)

The partial derivative $\frac{\partial\phi}{\partial t}$ expresses the "local" changes in the currently observed amount of fluid (i.e. due to diffusion or external influences) while the term $\mathbf{v}\cdot\nabla\phi$ represents the changes due to advection (transport of properties together with the matter). By replacing the acceleration $\mathbf{a}$ in (2.2) with the substantive derivative of the velocity we get:

$$\mathbf{F} = m\frac{\mathrm{D}\mathbf{v}}{\mathrm{D}t} = m\left(\frac{\partial\mathbf{v}}{\partial t} + \mathbf{v}\cdot\nabla\mathbf{v}\right) \tag{2.4}$$

$\nabla\mathbf{v}$: gradient of the velocity (the Jacobian matrix)

(2.1) states that the mass of the fluid inside the observed control volume depends on its density, therefore we write:

$$\mathbf{F} = \rho\left(\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v} \cdot \nabla \mathbf{v}\right) \tag{2.5}$$

Now we will focus on the forces acting on the fluid. It can be distinguished between internal forces produced by the fluid itself and external forces like gravity or electromagnetic forces:

$$\rho\left(\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v} \cdot \nabla \mathbf{v}\right) = \mathbf{F}_{Fluid} + \mathbf{F}_{External} \tag{2.6}$$

The most important external force is gravity $\mathbf{F}_G = m\mathbf{g}$, which is in fact stated as gravitational acceleration. Synonymously we will describe the external forces as force density field $\mathbf{g}$ that directly specifies acceleration (remember that the mass depends on the density in our case):

$$\rho\left(\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v} \cdot \nabla \mathbf{v}\right) = \mathbf{F}_{Fluid} + \rho\mathbf{g} \tag{2.7}$$

In order to provide a simple expression for the fluid forces, we assume that we deal with a Newtonian fluid that satisfies the incompressible flow condition. A viscid fluid is called Newtonian when the viscous stress is proportional to the velocity gradient (cp. [Pap99]). For Newtonian fluids the equation $\tau = \mu \frac{dv_x}{dy}$ describes the relation between shear stress τ, dynamic viscosity constant μ and the velocity gradient perpendicular to the direction of share $\frac{dv_x}{dy}$ [BE02]. This means in common words that, in contrast to non-Newtonian fluids, the viscosity is a constant and does not change under different shear rates. The fluid flow is called incompressible when the divergence of the velocity field is zero ($\nabla \cdot \mathbf{v} = 0$), meaning that there are no sources or sinks in the velocity field. As a counter example think of air that expands because it is heating up. Note that also flows of compressible fluids (all real fluids are compressible to some extent) can satisfy the incompressible flow condition (i.e. regular air flow till ~ mach 0.3). If the fluid fulfils all this conditions, we can simply spilt fluid forces into forces due to pressure differences (normal stresses) and in viscosity forces due to velocity differences (shear stresses):

$$\rho\left(\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v} \cdot \nabla \mathbf{v}\right) = \mathbf{F}_{Pressure} + \mathbf{F}_{Viscosit\ y} + \rho\mathbf{g} \tag{2.8}$$

The pressure forces depend only on the *differences* in pressure and let the fluid flow from areas of high to areas of low pressure. We model them with the negative gradient of the pressure field $-\nabla p$, which points from high to low pressure areas and has a magnitude proportional to the pressure difference:

$$\rho\left(\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v} \cdot \nabla \mathbf{v}\right) = -\nabla p + \mathbf{F}_{Viscosity} + \rho \mathbf{g} \tag{2.9}$$

Because of our assumption of an incompressible flow, the viscosity force becomes a relatively simple term:

$$\mathbf{F}_{Viscosity} = \eta \nabla \cdot \nabla \mathbf{v} \tag{2.10}$$

η: dynamic viscosity; $\nabla \cdot \nabla$: the Laplacian operator, sometimes also written ∇^2

For a mathematical derivation of the term above see i.e. chapter 5 in [Pap99] or [WND]. Here it should only be remarked, that the Laplacian is an operator that measures how far a quantity is from the average around it and therefore the force expressed by (2.10) smoothes the velocity differences over time. This is what viscosity is supposed to do. By combining the last two formulas we end up with the Navier-Stokes momentum equation for incompressible, Newtonian fluids often simply referred to as *the* Navier-Stokes equation:

$$\rho\left(\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v} \cdot \nabla \mathbf{v}\right) = -\nabla p + \eta \nabla \cdot \nabla \mathbf{v} + \rho \mathbf{g} \tag{2.11}$$

Navier-Stokes equation

This equation is the basis of a bunch of fluid simulation models. In 2.4 we will combine it with the basic principles of smoothed particle hydrodynamics (2.3) to form the mathematical model of the fluid simulation presented in this thesis. The sense of its rather descriptive derivation in this subchapter was to make the equation plausible in each of its parts and as a whole. The derivation, therefore, was intentional not mathematically strict and left out some concepts that are relevant for other forms of the equation (like the stress tensor $\boldsymbol{\tau}$). In the literature (i.e. [Pap99]) numerous mathematical strict derivations can be found if needed. This subchapter made clear that the Navier-Stokes equation is simply a formulation of Newton's second law and a statement of momentum conservation for fluids.

2.3 Basics of smoothed particle hydrodynamics

Smoothed particle hydrodynamics is a technique developed by Gingold and Monaghan [GM77] and independently by Lucy [Luc77] for the simulation of astrophysical gas-dynamics problems. As in other numerical solutions to fluid dynamic problems, the value of a physical quantity at a given position $A(\mathbf{r})$ must be interpolated from a discrete set of points. SPH derives from the integral interpolation:

$$A_I(\mathbf{r}) = \int A(\mathbf{r}')W(\mathbf{r}-\mathbf{r}',h)d\mathbf{r}' \tag{2.12}$$

$W(\mathbf{r},h)$ is a radial symmetric smoothing function (also called kernel) with smoothing length h (also called core radius). One could say that the interpolation uses the smoothing kernel to spread a quantity from a given position in its surroundings. In practice the kernel is even ($W(\mathbf{r},h) = W(-\mathbf{r},h)$) and normalised ($\int W(\mathbf{r})d\mathbf{r} = 1$) and tends to become the delta function for h tending to zero (if W would be the delta function, $A_I(\mathbf{r})$ would reproduce $A(\mathbf{r})$ exactly). This thesis follows the example of [MCG03] to treat h as the radius of support, so all used smoothing functions will evaluate to zero for $r \geq h$.

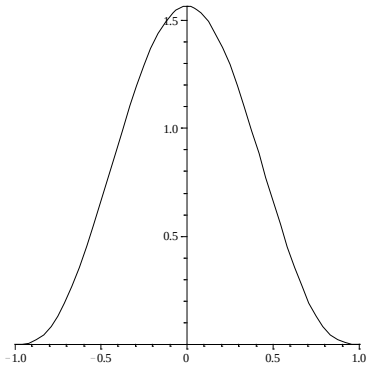


Figure 8: 1D example for a smoothing kernel

With SPH, a Lagrangian method, the interpolation points are small mass elements that are not fixed in space (like the grid points in the Euler method) but move with the fluid. For each such fluid particle j a position $\mathbf{r}_j$, velocity $\mathbf{v}_j$, mass m_j and density ρ_j is tracked. The value of a quantity at a given position can be interpolated from the particle values A_j using the summation interpolant (derived from the integral form):

$$A_s(\mathbf{r}) = \sum_j A_j \frac{m_j}{\rho_j} W(\mathbf{r}-\mathbf{r}_j,h) \tag{2.13}$$

The mass-density-coefficient appears because each particle represents a volume of $V_i = \frac{m_j}{\rho_j}$. As an interesting example (2.13) applied to the density ρ gives:

$$\rho(\mathbf{r}) = \sum_j m_j W(\mathbf{r} - \mathbf{r}_j, h) \tag{2.14}$$

Which shows that with SPH the mass density is estimated by smoothing the mass of the particles.

In practice not all particles must participate in the summation. As the smoothing kernel only has a finite radius of support, all particles with a greater distance to the evaluated point can be omitted.

An advantage of SPH is that spatial derivatives (which appear in many fluid equations) can be estimated easily. When the smoothing kernel is differentiable the partial differentiation of (2.13) gives:

$$\frac{\partial A_s(\mathbf{r})}{\partial x} = \sum_j A_j \frac{m_j}{\rho_j} \frac{\partial W(\mathbf{r} - \mathbf{r}_j, h)}{\partial x} \tag{2.15}$$

The gradient, therefore, becomes:

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

According to [MCG03] this could also be applied to the Laplacian:

$$\nabla \cdot \nabla A_s(\mathbf{r}) = \sum_j A_j \frac{m_j}{\rho_j} \nabla \cdot \nabla W(\mathbf{r} - \mathbf{r}_j, h) \tag{2.17}$$

There also exist some different SPH formulations for the gradient and Laplacian that will not be further discussed here. Chapter 2.2 in [CEL06] gives a good overview of other useful formulations. Monaghan also suggests alternatives to (2.17) in chapter 2.3 of [Mon05].

This rules cause some problems when they are used to derive fluid equations for particles. The derivate does not vanish when A is constant and a number of physical laws like symmetry of forces and conservation of momentum are not guaranteed. When the time has come, we will therefore have to adjust the particle fluid equations slightly to ensure physical plausibility.

2.4 Particle based, mathematical model of fluid motion

Now the core concepts of SPH from 2.3 will be applied to the Navier-Stokes equation introduced in 2.2 in a straightforward way, to form a mathematical model for particle based fluid simulation that is simple enough to be suitable for realtime usage. This subchapter is entirely based on the [MCG03] paper, which introduced the lightweight simulation model used in this thesis.

In the model presented here each particle represents a small portion of the fluid. The particles carry the properties mass (which is constant and in this case the same for all particles), position and velocity. All other relevant quantities will be derived from that using SPH rules and some basic physical equations.

Grid-based Eulerian fluid models need an equation for the conservation of momentum like the Navier-Stokes equation (2.11) and at least one additional equation for conservation of mass (sometimes one for energy conservation too) like the continuity equation:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0 \tag{2.18}$$

Continuity equation

The mass of each particle and the count of particles are constant, so mass conservation is guaranteed automatically. Hence, the momentum equation is all that is needed to describe the movement of the fluid particles in our model. Furthermore, a Lagrangian model doesn't have to take advection of currents into account (see the comparison in 2.2) and thus the substantial derivative of the velocity field in the Navier-Stokes equation can be replaced with an ordinary time derivative of the particle velocity. What we get is a momentum equation for a single fluid particle:

$$\rho(\mathbf{r}_i)\frac{d\mathbf{v}_i}{dt} = \mathbf{F}_i = -\nabla p(\mathbf{r}_i) + \eta \nabla \cdot \nabla \mathbf{v}(\mathbf{r}_i) + \rho(\mathbf{r}_i)\mathbf{g}(\mathbf{r}_i) \tag{2.19}$$

$\mathbf{F}_i$: force acting on particle i

$\rho(\mathbf{r}_i)$: density at position of particle i

$\nabla p(\mathbf{r}_i)$: pressure gradient at position of particle i

$\nabla \cdot \nabla \mathbf{v}(\mathbf{r}_i)$: velocity Laplacian at position of particle i

For the acceleration of a particle we get therefore:

$$\mathbf{a}_i = \frac{d\mathbf{v}_i}{dt} = \frac{\mathbf{F}_i}{\rho(\mathbf{r}_i)} \tag{2.20}$$

In (2.14) we have already seen how we could calculate the density at the particles position using the SPH rule (2.13):

$$\rho(\mathbf{r}_i) = \sum_j m_j W(\mathbf{r}_i - \mathbf{r}_j, h) \tag{2.21}$$

The external force density field rightmost in (2.19) directly specifies acceleration when the density factor vanishes after the division in (2.20). All what is left for a complete description of the particle movement based on the Navier-Stokes equation are the terms for pressure and viscosity forces.

Pressure

According to the SPH rules the pressure term would look like as follows:

$$\mathbf{F}_i^{pressure} = -\nabla p(\mathbf{r}_i) = -\sum_j p_j \frac{m_j}{\rho_j} \nabla W(\mathbf{r}_i - \mathbf{r}_j, h) \tag{2.22}$$

Unfortunately the resulting force is not symmetric. This could be easily seen when only two particles interact. Because the gradient of a radial smoothing kernel is zero at its centre, particle i only uses the pressure of particle j and vice versa. The pressure varies at different positions and thus the pressure forces would be different for the two particles. [MCG03] suggests balancing of the forces by using the arithmetic mean pressure of the two interacting particles:

$$\mathbf{F}_i^{pressure} = -\sum_j m_j \frac{p_i + p_j}{2\rho_j} \nabla W(\mathbf{r}_i - \mathbf{r}_j, h) \tag{2.23}$$

Till now the pressure at the particle positions was an unknown. Müller et al. propose to use the ideal gas state equation to derive the pressure directly from the density:

$$p = k(\rho - \rho_0) \tag{2.24}$$

k: gas constant depending on temperature; ρ_0: rest density

Viscosity

Applying the SPH rule to the viscosity term yields the following equation:

$$\mathbf{F}_i^{viscosity} = \eta \nabla \cdot \nabla \mathbf{v}(\mathbf{r}_i) = \eta \sum_j \mathbf{v}_j \frac{m_j}{\rho_j} \nabla \cdot \nabla W(\mathbf{r}_i - \mathbf{r}_j, h) \tag{2.25}$$

This again results in asymmetric forces for two particles with different velocities. The viscosity forces depend only on velocity differences, not on absolute velocities; therefore the use of velocity differences is a legitimate way of balancing the force equation:

$$\mathbf{F}_i^{viscosity} = \eta \sum_j m_j \frac{\mathbf{v}_j - \mathbf{v}_i}{\rho_j} \nabla \cdot \nabla W(\mathbf{r}_i - \mathbf{r}_j, h) \tag{2.26}$$

This means the viscosity force in our model accelerates a particle to meet the relative speed of its environment.

Surface tension

Now we have a simple model for the forces acting on the particles that contains everything what is expressed by the Navier-Stokes equation. But there is an additional fluid force relevant for the scenario we would like to describe, which is not covered by the momentum equation. Fluids interacting with solid environments often produce small splashes and puddles with much free surface, where the surface tension force plays a noticeable role. As described in 2.2 the surface tension forces try to minimize the surface of the fluid body, to achieve an energetically favourable form. The bigger the curvature of the surface is, the bigger should be the surface tension forces that push the border particles towards the fluid body. In order to find the particles at the surface and calculate the surface tension forces, the colour field method is used in [MCG03]. A colour field is 1 at particle positions and 0 everywhere else. The smoothed colour field has the form:

$$c_S(\mathbf{r}) = \sum_j 1 \frac{m_j}{\rho_j} W(\mathbf{r} - \mathbf{r}_j, h) \tag{2.27}$$

The gradient ∇c_S of the color field gives us two kinds of information: Its length becomes huge only near the surface, which helps us identifying surface particles and its direction points towards the centre of the fluid body, which is a good choice for the direction of the surface force. The surface curvature, which is a magnitude for the size of the force, can be expressed by the Laplacian of the colour field:

$$\kappa = \frac{-\nabla \cdot \nabla c_S}{|\nabla c_S|} \tag{2.28}$$

Using the colour field gradient as force direction and "marker" for surface particles and the curvature as magnitude for the force size leads to the following equation for the surface tension force:

$$\mathbf{F}^{surfacetension} = \sigma \kappa \nabla c_S = -\sigma \nabla \cdot \nabla c_s \frac{\nabla c_s}{|\nabla c_S|} \tag{2.29}$$

σ: surface tension coefficient; depends on the materials that form the surface

$|\nabla c_S|$ is near to zero for inner particles, so the surface tension is only getting evaluated when it exceeds a certain threshold to avoid numeric problems. It should be mentioned that this surface tension model can be error-prone under some circumstances, so also other models proposed in the literature (i.e. in [BT07]) may be worth an evaluation.

2.5 Smoothing kernels

The smoothing kernels used in the interpolations have great influence on speed, stability and physical plausibility of the simulation and should be chosen wisely. As every kernel $W(\mathbf{r}, h)$ is radial symmetric, it is normally specified only as function of the length of $\mathbf{r}$: $W(r, h)$. It should be even ($W(\boldsymbol{r}, h) = W(-\boldsymbol{r}, h)$), normalised ($\int W(\mathbf{r}, h)\, d\mathbf{r} = 1$) and differentiable as often as needed. Despite of these requirements one is free to specify the kernel in every form that is suitable for its task. In the literature there exist many different ways to specify them, from M_n splines, over exponential functions up to Fourier transformation generated kernels. [Mon05] contains a good overview of the most common techniques.

In this thesis the kernels proposed in [MCG03] are used. The first is the Poly6 kernel:

$$W_{poly\,6}(\mathbf{r}, h) = \begin{cases} \frac{315}{64\pi h^9}(h^2 - r^2)^3, & 0 \le r \le h \\ 0, & otherwise \end{cases} \tag{2.30}$$

with: $r = |\mathbf{r}|$

it has the gradient:

$$\nabla W_{poly\,6}(\mathbf{r}, h) = -\mathbf{r}\frac{945}{32\pi h^9}(h^2 - r^2)^2 \tag{2.31}$$

and the Laplacian:

$$\nabla \cdot \nabla W_{poly\,6}(\mathbf{r}, h) = \frac{945}{8\pi h^9}(h^2 - r^2)\left(r^2 - \frac{3}{4}(h^2 - r^2)\right) \tag{2.32}$$

Note that in the appendix there is the section ”Derivation of the gradient and Laplacian of the smoothing kernels”

Its advantage is that r appears only squared, so the computation-intense calculation of square roots can be avoided. The Poly6 kernel is used for everything except the calculation of pressure and viscosity forces. With pressure forces the problem is that the gradient goes to zero near the centre. Therefore, the repulsive pressure force between particles vanishes when they get too close to each other. This problem is avoided by the use of the Spiky kernel, which has a gradient that does not vanish near the centre:

$$W_{spiky}(\mathbf{r}, h) = \begin{cases} \frac{15}{\pi h^6}(h - r)^3, & 0 \le r \le h \\ 0, & otherwise \end{cases} \tag{2.33}$$

Gradient:

$$\nabla W_{spiky}(\mathbf{r}, h) = -\mathbf{r}\frac{45}{\pi h^6 r}(h - r)^2 \tag{2.34}$$

With viscosity the problem of the Poly6 kernel is that its Laplacian becomes negative really fast. A particle that is faster than its environment may therefore be accelerated by the resulting viscosity forces, while it should actually get slowed down. In the viscosity calculation thus the “Viscosity” kernel is used, which’s Laplacian stays positive everywhere:

$$W_{viscosity}(\mathbf{r},h)=\begin{cases}\frac{15}{2\pi h^3}\left(-\frac{r^3}{2h^3}+\frac{r^2}{h^2}+\frac{h}{2r}-1\right), & 0\le r\le h\\ 0, & otherwise\end{cases} \tag{2.35}$$

Gradient:

$$\nabla W_{viscosity}(\mathbf{r},h)=\mathbf{r}\frac{15}{2\pi h^3}\left(-\frac{3r}{2h^3}+\frac{2}{h^2}-\frac{h}{2r^3}\right) \tag{2.36}$$

Laplacian:

$$\nabla\cdot\nabla W_{viscosity}(\mathbf{r},h)=\frac{45}{\pi h^5}\left(1-\frac{r}{h}\right) \tag{2.37}$$

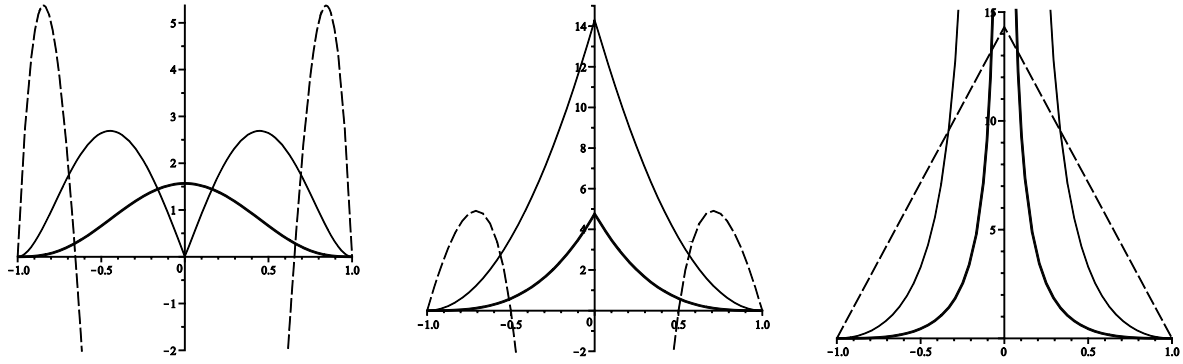


Figure 9: Used smoothing kernels

$W_{poly\,6}, W_{spiky}, W_{viscosity}$ (from left to right) along the x-axis for $y=0, z=0, h=1$
thick lines: kernel, thin l.: absolute value of gradient, dashed l. Laplacian

2.6 Basic simulation algorithm

Subchapter 2.4 described how the fluid forces acting on the particles could be derived directly from the particle positions and velocities. This enables us to specify the basic algorithm for the fluid simulation:

Listing 1: Basic simulation algorithm

```
while simulation is running
   h ← smoothing-length

   init density of all particles
   clear pressure-force of all particles
   clear viscosity-force of all particles
   clear colour-field-gradient of all particles
   clear colour-field-laplacian of all particles

   // calculate densities
   foreach particle in fluid-particles
      foreach neighbour in fluid-particles
         r ← position of particle - position of neighbour
         if length of r ≤ h
            add mass * W_poly6(r, h) to density of particle // compare (2.14)
         end-if
      end-foreach
   end-foreach

   // calculate forces and colour-field
   foreach particle in fluid-particles
      foreach neighbour in fluid-particles
         r ← position of particle - position of neighbour
         if length of r ≤ h
            density-p ← density of particle
            density-n ← density of neighbour

            pressure-p ← k * (density-p - rest-density) // (2.24)
            pressure-n ← k * (density-n - rest-density)

            add mass * (pressure-p + pressure-n) / (2 * density-n) // (2.23)
                  * gradient-W-spiky(r, h) to pressure-force of particle

            add eta * mass * (velocity of neighbour - velocity of particle)
                  / density-n * laplacian-W-viscosity(r, h)
                  to viscosity-force of particle //(2.26)

            add mass / density-n * gradient_W_poly6(r, h)
                  to colour-field-gradient of particle

            add mass / density-n * laplacian_W_poly6(r, h)
                  to colour-field-laplacian of particle
         end-if
      end-foreach
```

```
   end-foreach

   // move particles
   foreach particle in fluid-particles
      gradient-length ← length of colour-field-gradient of particle
      if gradient-length ≥ threshold // (2.29)
         surface-tension-force ← -sigma * colour-field-laplacian of particle
               * colour-field-gradient of particle / gradient-length
      else
         surface-tension-force ← 0
      end-if

      total-force ← surface-tension-force + pressure-force of particle
            + viscosity-force of particle

      // (2.20)
      acceleration ← total-force / density of particle * elapsed-time + gravity

      add velocity of particle + acceleration * elapsed-time
            to velocity of particle

      add velocity * elapsed-time to position of particle
   end-foreach
end-while
```

The dependencies on the density and the forces lead to a tripartite evaluation scheme. First the density of each particle is evaluated by summation over the contributions of all particles in the neighbourhood. In the second step every neighbour exerts forces on the particle and the colour field is being built. At last the accumulated forces are used to approximate the movement of the particles in the current time step.

2.7 Implementation

The fluid simulation as well as whole other CPU code of the program was implemented with C++, because today it is the de facto standard in professional, realtime computer graphics on PCs. The pseudo code in the last chapter describes the real implementation of the simulation component relatively well. The update method that is called once for every simulation step, indeed linearly executes the following four tasks:

1. calculate density at every particle position
2. calculate pressure forces, viscosity forces and colour field values for each particle
3. move the particles and clear the particle related fields
4. update the acceleration structures

As stated before the particles carry only the properties position and velocity (the mass is constant and the same for all particles). This is the only information that is transferred from one simulation step to the next. All other per-particle data, like density and forces is stored in separate arrays. The particle data structure, therefore, consists of one three-component vector for position, one for velocity and an integer index that locates derived particle properties in the respective arrays.

The first task, the density calculation, has to implement the summation interpolation. The summation is the most crucial point for the overall performance of the simulation. The naive summation over all particles in the simulation would result in a computation complexity that is quadratic in the number of particles, which is impracticable for the amounts of particles we aim at. Therefore, it is necessary to implement the summation as a neighbour search that finds all neighbour particles that are near enough to influence a certain particle. Those are all particles with a distance lower than their radius of support (= smoothing length in our case). In this simulation the smoothing length is treated constant and equal for all particles.

Neighbour search

This allows the use of a location grid as efficient acceleration structure for the neighbour search. The grid consists of cubic cells with a side length equal to the smoothing length. Each cell contains a reference to a list of all particles that map to the space partition associated with the cell or a null pointer, if no such particle exists. The particle positions change with every simulation step. Thus after each step the grid location and cell count dimensions must be updated to fit the space occupied by the particles and the particles must be sorted into the grid again.

The neighbour search finds the neighbours of all particles in a particular grid cell. Because the side length equals the smoothing length, all neighbouring particles must be contained in the current or one of the maximal 26 adjacent cells. This reduces the time complexity of the summation from $\mathrm{O}(n^2)$ to $\mathrm{O}(nm)$ (m being the average number of particles per grid cell) at the cost of the time needed to rebuild the grid ($\mathrm{O}(n)$).

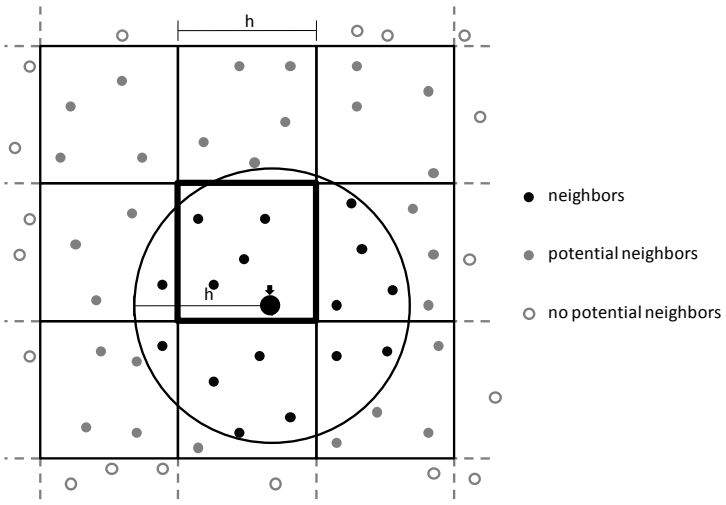


Figure 10: Grid based neighbour search

A further performance gain is accomplished through storing copies of the particles in the grid cells instead of references. This dramatically lowers the cache miss rate of the CPU, because all particles that are accessed during the neighbour search for particles within one cell lie close to each other in system memory.

Exploitation of symmetry

The neighbour relation between the particles is symmetric ($a\ neighbor\ b \Rightarrow b\ neighbor\ a$) and also the interactions between the neighbors (density accumulation, force exertion) are mostly symmetric. This allows another optimisation: Whenever a particle pair contained in the neighbour relation is found, all necessary calculations are performed in both directions, so that every pair must be evaluated only once. The algorithm visits cell after cell. First it checks each particle against all that follow in the same cell. Then it checks all the pairs between the current cell and one half of the neighbour cells. If all neighbouring cells would be considered, the whole algorithm would evaluate each cell-neighbourhood twice. Thus all cells that are located on the opposite site of already checked cells are skipped (see Figure 11). In this manner the algorithm halves the computation complexity and ensures that every pair is found exactly once. The optimisation also has the consequence that no particle gets evaluated against itself, which is all right when the density initialisation takes care of the self induced density (for the forces and the colour field gradient/Laplacian this doesn't matter at all).

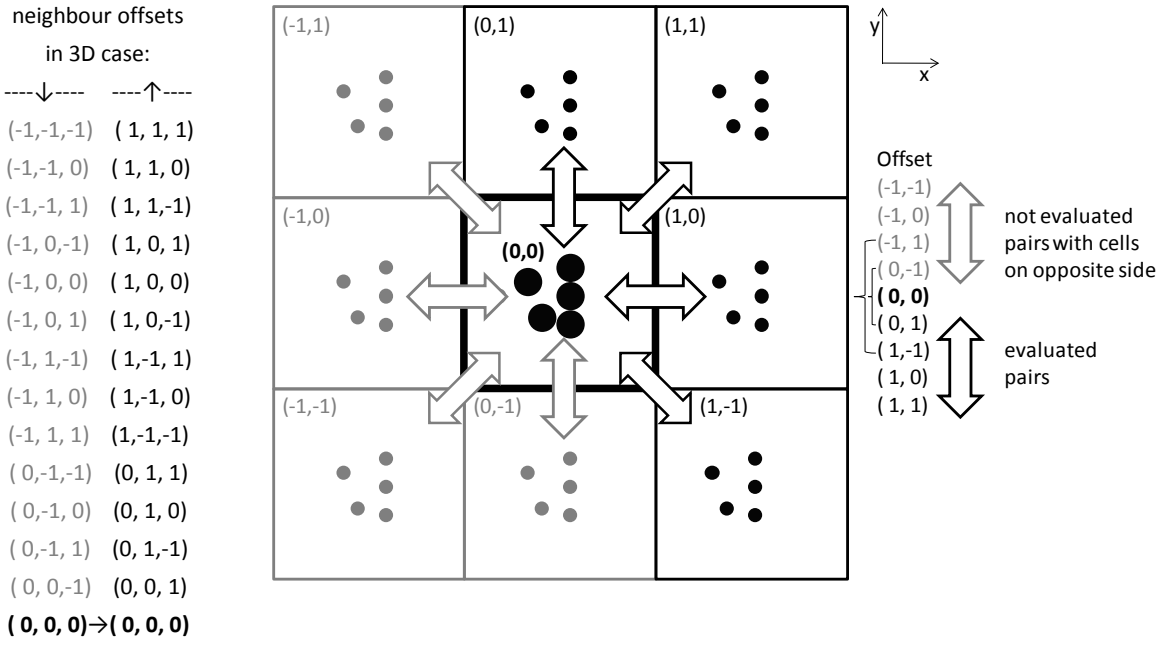


Figure 11: Skip neighbour cells on the opposite side

The density calculation is not the only task where the summation interpolation and therefore the neighbour search must be performed. In the separate force and colour field calculation the same neighbourhood relations are needed. Therefore, the particle pairs that are found by the neighbour search during the density computation phase are stored and reused within the following force and colour field stage.

The neighbour search delivers us all particle pairs with a distance below the smoothing length. The C++-method for the density computation calculates the additional density that the two particles impose on each other ($\rho_{additional} = m \cdot W(distance, h)$) and ads it to the total densities of both particles.

Similar the pressure force, viscosity force, colour field gradient and colour field Laplacian calculations in the second task first compute a common term for both particles according to the appropriate SPH equation. The term gets weighted with the density inverse of the neighbour particle (which is part of all four related SPH equations) and provided with the right direction (in case of a vector) before it is added to the particles overall values.

Calculation of movement

After the first two tasks have pair wise evaluated the density, pressure force, viscosity force and the colour field values of every particle, the third task processes the particles linearly. The colour field gradient and Laplacian is used to calculate the surface tension force, which ads up with pressure and viscosity forces to the total per-particle force in the current time step. Total force divided by mass density results in an acceleration (Newton's law), which is added to the constant earth acceleration g to get the total acceleration. Combined with current velocity, position and step duration it finally leads to the new velocity and position of our particle at the end of the current simulation step, respectively the beginning of the next. Because position and velocity are the only information that is kept for the next step, all the other property fields get cleared/initialized at the end of the calculation.

The fourth and final step clears the neighbour search grid and rebuilds it from the new particle positions. For that purpose first the new spatial dimensions of the particle cloud are calculated and a properly placed and scaled empty grid is created. Then particle after particle gets sorted into the grid according to its position, whereby new cells are created on demand if a particle falls to a position where no cell exists yet.

2.8 Environment and user interaction

A fluid floating around in empty space is rather untypically in our everyday environment. Thus we want to simulate interaction of the fluid with solid obstacles or containers. Moreover, the name suggests that an interactive realtime simulation should provide some sort of user interaction with the simulated object. Therefore, a liquid fluid has been placed in a virtual water glass that the user can move around with the mouse. This scenario is comparatively easy to simulate and because the fluid cannot flow away, the user gets a steady simulation that he can interact with over a long time. Additionally surely everyone once watched his drink when it is shaken around in the glass and thus we know very well how the fluid would behave in reality.

The environment interaction in this simulation works only in one direction, meaning that the movement of the simulated glass is entirely controlled by the user with the mouse and the fluid does not exert forces on the glass that would cause it to move. Conceptually the glass is modelled as an infinite long, vertical aligned cylinder as side walls and a horizontal aligned plane as ground of the glass. The collision detection, therefore, becomes a simple check of the particles distance from the cylinders centre line, respectively from the bottom plane. A first implementation of the glass interaction only checked if a particle was outside the glass and repositioned it back into the glass along the border normal. However, this does not lead to any physical plausible results, because thus the glass does not influence the fluid density near the border, nor does it participate in the pressure and viscosity computation. The actual implementation simulates the interaction of the glass with the fluid particles with the same SPH methods that are responsible for the particle-particle interactions. Therefore, synonym to the density and forces calculation phases for the fluid itself, extra density and forces calculation phases for the glass have been added to the simulations update method. Thus, a simulation update is now performed in six steps:

1. update densities (particle <-> particle)
2. update densities (glass -> particle)
3. update pressure forces, viscosity forces and colour field (particle <-> particle)
4. update pressure and viscosity forces (glass -> particle)
5. move particles, enforce glass boundary, clear fields
6. update the neighbour search grid

Because in some extreme situations the glass emitted pressure forces are not sufficient to keep the particles inside the glass, the fluids move-method (step 5) was equipped with a modified version of the old collision response code. It ensures that the particles do not leave the glass too far and prevents them from permanently moving away under some extreme rotation conditions.

2.9 Multithreading optimisation

Today's higher end consumer PC's are all equipped with dual or quad-core CPUs. The performance of a single-process application can only profit from more than one CPU core when it distributes its computation load among multiple threads. In that way the different cores can execute multiple parts of the computation in parallel, whereas a single-threaded application would only utilize one of the cores.

As a consequence of the simulation's step based execution scheme, the threads do not work on long running tasks, but instead on short recurring ones. Therefore, it must be possible to quickly allocate threads (creation would be too expensive), assign them a task, start their execution and wait until they are all finished with as minimal overhead as possible. For that purpose a worker-thread manager was created that holds a pool of worker threads (per default as much as physical cores are available to the process) and offers functions for comfortable parallel execution of jobs.

Two major ways to parallelize the program execution exist: Make use of task parallelism or make use of data parallelism. At the beginning of the multi-core era on consumer PC's, mostly task parallelism was exploited, because it is comparatively easy to execute distinct parts of a program in parallel. However, task parallelism requires the existence of enough independent heavy-worker tasks to make use of all cores. Furthermore, in a realtime application it is unlikely that each task requires comparable execution times, so some cores will run at full capacity while others are often idle. In the case of this fluid simulation, all performance critical tasks depend on their precursor, so there cannot be made any reasonable use of task parallelism at all. The fluid simulation, therefore, utilizes data parallelism where ever it seems possible and lucrative. This means concretely that the first 5 of the 6 update tasks where parallelized:

The density calculation step begins with the grid based neighbour search. The distinct grid cells thereby provide a natural data separation criterion. Each thread only searches neighbours for particles in grid cells with an index dividable by its own id. This fine grained distribution causes an almost equal utilisation of all threads. However, it doesn't prevent the threads to find pairs with particles in neighbour cells that are handled by a different thread. This principally becomes a problem when the thread adds the additional density to the values for both particles. Because the add-operation (C++: +=) is not atomic at the instruction level, a simultaneous add attempt from two threads could lead to a swallow of one of the summands. To overcome this problem, one could use atomic operations at x86-instruction-set level (inline assembler; CMPXCHG-instruction) or provided by the operating system (Win32-API; InterlockedIncrement-function). However, the summation is very performance critical, so the memory barriers needed for those commands would cause an immense performance hit and with the vector values in the later phases things would get complicated. The good news is that with many particles the probability for such a collision is very low and its consequences (losing the contribution of one particle) are not dramatically for the overall simulation. Thus the density array is only marked as "volatile" to prevent the worst multithread-errors because of caching and further possible collisions are treated as an acceptable risk. Every thread stores its own particle pair list for the later forces step, so that the same data distribution among the threads is used there. The pressure and viscosity force arrays as well as the colour field arrays are also simply marked as volatile, but not further synchronized.

The tasks for glass-related density and force calculation as well as the movement task simply let every thread linearly compute on the same count of particles. Those three tasks do not need any synchronisation at all, because they always operate on distinct data.

The last task, the sort of the particles into the neighbour search grid is performed single-threaded. Because a failure with the insertion of the particles into the lists in the cells would cause major trouble to the simulation, a strong synchronisation associated with a performance hit would be necessary. Performance improvements here would not make a great difference anyhow, because the insertion into the grid does need only ~5% of the total computing time of the simulation.

The work on the multithread ability of the simulation did pay off. In the tests the program version with the multithreaded fluid simulation engine achieved an 83% better overall performance on a quad-core CPU (Intel Core 2 Quad Q6600 @ 3.24 GHz) than the pure single threaded version (the frames per second of the entire application inclusive sprite rendering were measured).

2.10 Results

The fluid simulation produces satisfying results in terms of performance and believability of the liquid's behaviour.

The performance can be expressed in numbers: With 1728 particles (12^3) and simulation of all possible forces (pressure, viscosity and surface tension) the application (x64 binary) runs with ~330 frames per second (FPS) on a PC with 3.2 GHz quad-core CPU and 2 GB RAM (measured inclusive sprite rendering which ads no measureable overhead). With 10648 particles (22^3) still 53 FPS are achieved. At 27000 particles (30^3) the frame-rate drops down to 14. All experiments where run with simulation time steps depended on the real elapsed time to provide constant time behaviour for the viewer.

A measure for the plausibility of the behaviour is harder to find. First it should be mentioned that the application is capable to simulate the major effects that could be observed when a real liquid is shaken around in a glass: vortex formation, wave breaking, wave reflection, drop formation and drops that slowly drain down along the side of the glass to name a view

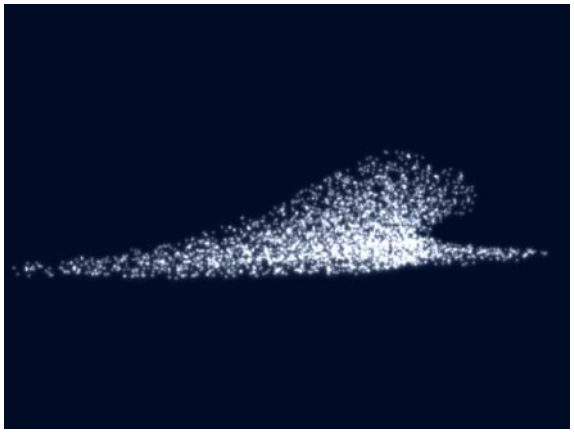


Figure 12: Liquid behaviour

More important is however, and sadly this couldn't be expressed with text or pictures, that the liquid's behaviour "feels" realistic. This implementation, therefore, is a solid basis for experiments with visualisation methods for interactive, particle-based liquid simulations. But there is still much room for improvements.

2.11 Further work and outlook

Incompressibility

As a consequence of the relatively simple simulation model, which uses density fluctuations as a basic concept (pressure derived from ideal gas state equation), the simulated fluids have a high compressibility. While all real fluids are compressible to some amount, water and many other liquids are so hard to compress, that they are commonly assumed incompressible. In the literature different solutions for the incompressibility in SPH simulations where proposed. In [CEL06] as example an algorithm is presented that makes a velocity field divergence fee (remember: $\nabla \cdot \mathbf{v} = 0$ is a statement of volume conservation or incompressibility in fluid dynamics). Hence, a "compressible simulation algorithm" could be used to generate velocities, which are modified for incompressibility in an extra step. Becker and Teschner mention that this approach is too time-consuming and prefer a solution that is comparable to the one of Monaghan. In [BT07] they use Tait's equation to specify the pressure term, which leads to a simulation that guarantees a maximal compressibility which "spreads" with the speed of sound (therefore small time steps are required). However, both approaches were used with offline simulations and to the author's knowledge there is still no paper with a satisfying solution to the compressibility problem suitable for realtime applications.

Surface tension

The surface tension algorithm is another point that could be improved. As mentioned in [BT07] the second order derivative of the colour field, which is used to model the surface tension forces, is sensitive to particle disorder and therefore not adequate for turbulent settings. Because of that, a model based on cohesive forces between the particles (see Figure 6: Cause of surface tension) is

proposed. In the current program a comparable model is already implemented, as in the simulation a higher “rest density” can be specified, which causes the particles to group together in energetically favourable shapes. In that way the effect of surface tension can be approximated with negative pressure forces, which makes the whole colour field computations obsolete.

Environment interaction

In the existing simulation an imaginary glass is the only object the fluid can interact with. The “collision detection” only measures the distance to the centre line and to the ground plane. A more general form of collision detection and collision handling would be necessary for the interaction with a richer environment. A common way to simulate obstacles in SPH simulations is to model them as particles that participate in the force and density calculations. This would kill two birds with one stone, as it delivers for free the forces that the fluid exerts on the obstacles, which would be necessary for two-way interaction with rigid body simulations (or other physics simulations). Because the mapping from common 3D geometry to a particle representation is not trivial and may introduce high additional computation costs, also other alternatives (i.e. interaction with simplified geometry) would be welcome.

Neighbour search

One major advantage of particle-based simulations among the Euler-grid-based ones is the absence of spatial limitations in the simulation domain. This advantage is relativized to some amount, because the current implementation still needs a kind of grid (a fairly coarse and dynamic one however) to find the neighbourhood relations. The neighbour search could be made more spatial flexible with the use of hashing algorithms that map unlimited amounts of space partitions to only few linear list slots (comp. [THM03]). Also other flexible space partitioning techniques like special forms of octrees or kd-trees may deliver feasible results for the neighbour search, if techniques would be developed that minimize the costs of the every-frame structure updates.

Target hardware

At last, the performance of the simulation still may not be sufficient to be used in real world applications, like i.e. commercial video games. Highly interactive frame rates for only a few thousand particles is not sufficient for the big, expressive effects one may probably want to see in such applications. This problem should be solvable in the next time. There are certainly still some further performance tricks and simplifications that can be applied to the code to get some more performance out of it. Furthermore, in the future more potent hardware will be used to execute such kind of programs. Today’s GPUs may be a good choice for such heavily parallelizable, floating point and vector related tasks (leads to a [GPGPU] simulation), if someone finds some suitable GPU acceleration structures for the neighbour search. But also the CPU manufacturers seem to work on products that provide better support for the SPMD (single program multiple data) like execution, that’s required for such programs.

Intel works on “Larrabee”, which best could be described as an “x86 GPU” that executes “real” general purpose programs on many, many hardware threads. AMDs technology is called “Fusion” and is about placing a CPU and GPU on the same processor die. AMD says that while it first will be used for cheap and energy-efficient solutions, later one wants to take advantage of the combined processing power that benefits from the direct connection and share of memory. So while the firms develop in slightly different directions, it is clear that both picked up the idea of massively parallel general purpose processing units, which is good news for physics simulation in general and realtime SPH in particular.

3 Visualisation

3.1 Chapter overview

This chapter discusses visualisation methods for realtime, particle-based fluid simulations. The goal was to visualise the simulated fluid as a water-like liquid. The optical behaviour of water should be simulated as realistic as possible in a realtime application. For this purpose a ray-tracing based water renderer was implemented, which allows simulation of multiple refractions and reflections. Before the work on the ray-tracer, two other visualisation techniques have been implemented, which could be seen as prerequisites for the real visualisation.

Subchapter 3.3 shows the implementation of sprite-based direct particle rendering with Direxť3D 9 fixed function- and Direct3D 10 geometry shader functionality. This simple technique was the precondition for the successful work on the SPH simulation. It is the best way to study the movement of fluid particles, as every single particle is observable.

3.4 demonstrates a surface rendering technique based on marching cubes. The algorithm, which is entirely CPU executed, constructs a triangle-mesh of the isosurface from a discrete volumetric density field. It represents the first experiments with volume rendering techniques and methods to build such volumes from a set of particles.

The main effort went into the GPU based isosurface ray-tracing, presented in 3.5. The subchapter first introduces a ray-casting based volume visualisation technique from 2006, which was the inspiration for the core of the ray-tracer. Then it presents the basic optical principles that are simulated by the renderer, before the actual shader model 4.0- based implementation is shown.

Subchapter 3.6 shows how the GPU is used to transform the particle fluid-representation into the volumetric density texture, which is needed by the ray-tracer and many other visualisation techniques. 3.7 closes the chapter with a short introduction of another promising visualisation technique and an outlook on possible developments in the future.

3.2 Target graphics API

The manufactures of PC graphics cards support two major 3D-graphics API's: OpenGL and Direct3D. Even though OpenGL has the advantage to be available on various different operating systems, many interactive applications use Direct3D, which is available as part of the DirectX multimedia API for most Microsoft platforms. The visualisation, presented in this thesis, was implemented with Direct3D too. It was chosen because of the author's personal experience with the API and not because of any actual advantages.

Direct3D 10 is the latest version and was released together with the Windows Vista operating system. It is one of the bigger updates that have been made to the API in its previous history. The major changes in this version to some extent also reflect the recent developments with graphics cards. Since modern GPUs use the same processing units for pixel and vertex programs, also the shader core was unified. This means all sorts of shaders now have (mostly) the same instruction set and access to resources. The new shader model 4.0 brings also better flow control and branching support. ASM-shader support was dropped, so HLSL is now the only shader authoring language. Another noticeable innovation is the drop of some parts of the fixed function pipeline. Transformation, lighting, shading and fogging are some of the operations that now must be implemented via shaders by the API user himself.

The new geometry shader stage allows per-primitive computation and primitive amplification (primitive = point/line/triangle). Generally it takes all the per-vertex data available to one primitive (inclusive the vertices from edge-adjacent primitives) and generates an arbitrary number of topologically connected vertices as output (supported topologies: triangle-strip, lines-strip, point-list). Its functionality is later used in the sprite-visualisation as well as in construction of the volumetric density texture. The Geometry shader output can be either feed the rest of the pipeline or can be outputted to a vertex-buffer in the also new stream-out stage. The following diagram shows the pipeline stages for Direct3D 10:

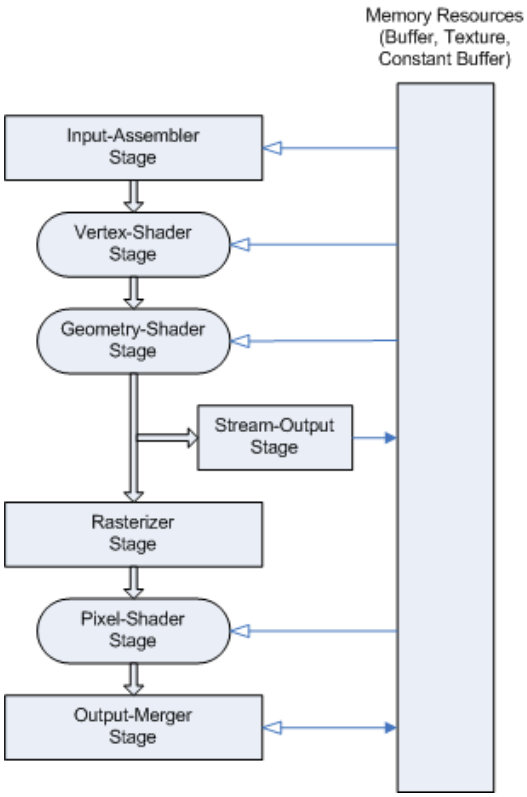


Figure 13: Direct3D pipeline stages
Source: MSDN library

Also the resource system was improved. Direct3D 10 distinguishes between buffers, which represent the actual GPU resources, and resource views, which control their interpretation and binding to the pipeline stages. In context of this more flexible resource system, it is now also possible to choose the render-target individually for each primitive in the geometry shader. This is especially helpful for the creation of the density volume (3.6), which also utilizes the new instanced draw call. The instanced draw allows drawing the same geometry multiple times with a single call. A unique id allows distinction of the instances within the shaders.

3.3 Direct particle rendering

The simplest method to visualize the results of the SPH simulation is to render the fluid particles directly. Even if this does not lead to a visual representation that looks like the fluid that is imitated, it is nevertheless a very useful visualisation technique. Being able to see the movement of every single particle is an immense help for fine-tuning and debugging the simulation. It could easily be seen when single particles accelerate unnaturally, start to vibrate or move in another unwanted way. But also global effects, like particles that form up in striking patterns, can be perceived by watching direct particle visualisations. Per particle quantities of the simulation (i.e. density) can be made visible by the use of size-, transparency- or colour-coding.

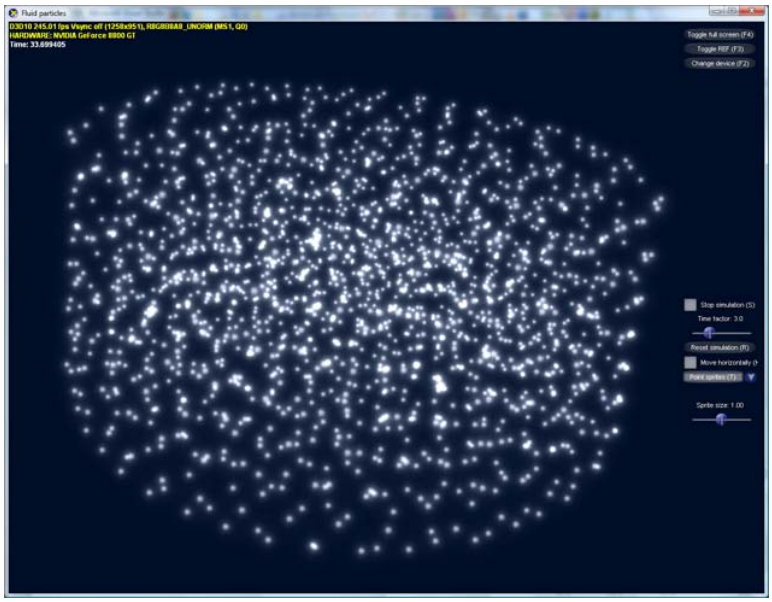


Figure 14: Sprite rendering

Different particle visualisation techniques exist. The most common of them in realtime computer-graphics is billboard rendering of sprites. At every particle position it places a textured rectangle that is always aligned towards the viewer (hence the name billboard). In order to provide a depth effect the size of the rectangles should be proportional to the distance from the viewer and alpha blending may be necessary to achieve visual appealing results.

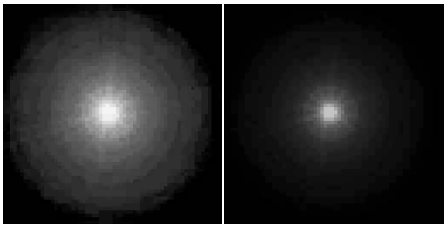


Figure 15: RGB and alpha channel of the particle texture

Two implementations of the sprite renderer where provided: One for Direct3D 9 and one for Direct3D 10. The D3D9 renderer uses the existing sprite render functionality provided by the fixed function pipeline. Therefore, it only must transfer the particle positions into a vertex buffer, set the sprite texture, enable point sprites and specify some additional render state variables, like alpha-enable and point-scale, before it starts rendering with a draw call on a point list.

Sprites with D3D10

Because in D3D10 the default billboard renderer is gone together with most of the fixed function pipeline, sprite rendering is slightly more complex to implement with it. Like its D3D9 pendant, the D3D10 sprite renderer transfers the particle positions in a vertex buffer and sets the sprite texture. Additionally it calculates position offsets of the four sprite corners relatively to the sprite centre in world space. For this purpose the inverse view matrix is needed to position the sprite corners in a way that aligns the billboards with their front facing towards the camera. During the rendering a geometry shader is invoked that generates two triangles for every input vertex, which represent the billboard rectangle. The triangle vertices are generated by adding the sprite corners to the particle positions in world space and transform the result to clip space afterwards. The associated pixel shader performs a lookup in the sprite texture for every fragment that is generated by the fixed function rasterizer and depth- and blend-states control the composition to the final image.

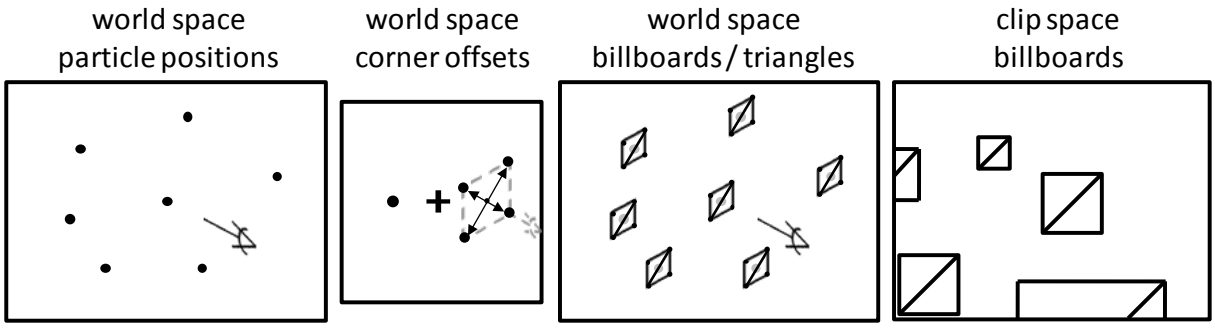


Figure 16: Billboard rendering with a D3D10 geometry shader

3.4 Isosurface rendering with marching cubes

The goal of the thesis is to simulate water-like fluids and therefore also the visualisation should produce images that look like water. However, good-looking efficient realtime water rendering for particle based simulations is still an open research topic. Since clear water is nearly as transparent as air, most visualisations display only the water surface. Thus, first of all the challenge is to find this free surface. Most current visualisations adopted a concept that is used with grid based fluid simulations: Isosurface rendering.

The isosurface

The basis for isosurface rendering is a discrete representation of the volumetric density scalar field of the fluid. How such a volume grid of the density can be constructed from a set of fluid particles will be discussed later in this subchapter. The idea is to look at regions with an equal density value (Greek word for equal: ἴσος - isos; hence the names isovalue/isolevel and isosurface). It is assumed that these regions have the form of a 2d surface, an orientable 2-manifold without boundary. Note that this would not be true, if the density function is discontinuous (-> boundaries) or has an equal value in a whole volumetric area (-> 3-manifold with boundary). The isosurface for a small density isovalue is a good approximation of the water surface. In the computer representation of the density field (basically some sort of float array) both the domain and the range are discrete. To avoid problems with volumetric zones of the same value, it is reasonable to define the isosurface as the area between regions where the value is less than the isovalue and regions where it is equal or higher. Thereby even volumetric areas of the same value produce a closed 2d isosurface.

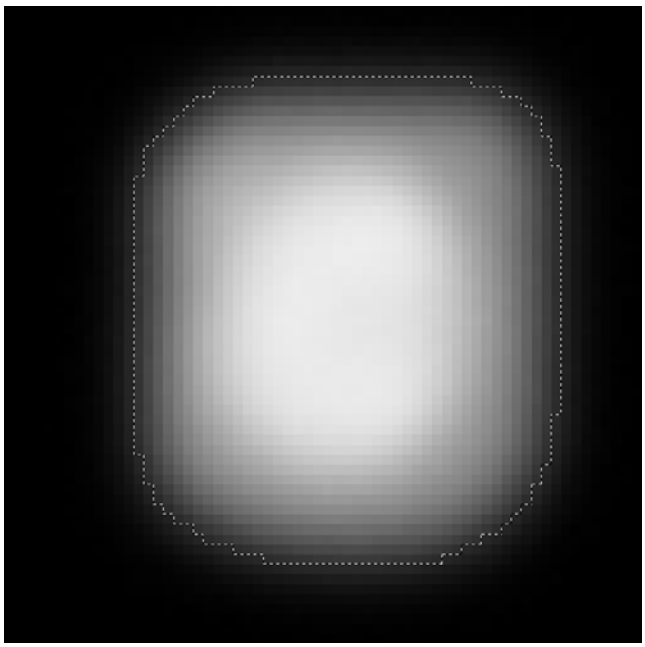


Figure 17: 2d illustration of a discrete isosurface

Marching cubes

A famous technique to construct a triangle mesh as renderable representation of the isosurface was published in 1987 by Lorensen and Cline[LC87]. The marching cubes algorithm piecewise processes eight density values at a time, which form an imaginary graph in form of a cube. First it checks for each of the 12 edges if it intersects the isosurface. This is the case if one of the connected nodes is less and the other is greater or equal than the isovalue. For every intersection it generates a vertex through linear interpolation of the two node positions, which estimates the position where the field value would exactly equal the isovalue. Then it generates an 8-bit code for the corners of the cube (1 if corner value is below isovalue, 0 otherwise). This code exactly identifies the 256 possible triangle configurations of a cube, which derive from 15 unique combinations trough rotation and reflection. The mesh that is formed by the triangles from all cubes, by design of the algorithm, represents the isosurface without any gaps.

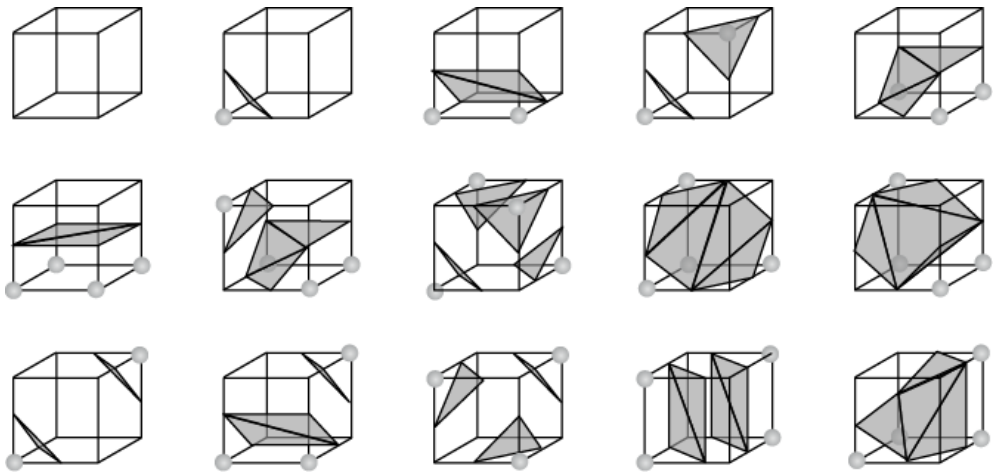


Figure 18: The 15 distinct cube triangle configurations
Source: Wikimedia commons

Implementation

The implementation for this thesis was tuned for high performance and thus became slightly more complex than this simple explanation. First of all, additionally to the scalar density field a vector-valued gradient field is used to provide an efficient way to generate normals. Furthermore, a clever implementation of the algorithm should take advantage of the facts that the cube edges share the same corner nodes (inside one cube as well as between adjacent cubes) and the triangles share the same vertices. The current implementation of the marching cubes algorithm, therefore, processes the volume in rows and uses the already calculated information of each cube's direct precursor, for the 4 shared corners and 4 shared edges (respectively up to 4 shared vertices). Hence, in every step only 4 instead of 8 corners and only 8 instead of 12 edges must be evaluated. Better use of shared information and hence generation of even fewer vertices would only be possible with complex management of already visited corners and edges. This would only be reasonable if the generated mesh is used for much more than one frame (when a speedup in visualisation would overweight the slower mesh construction).

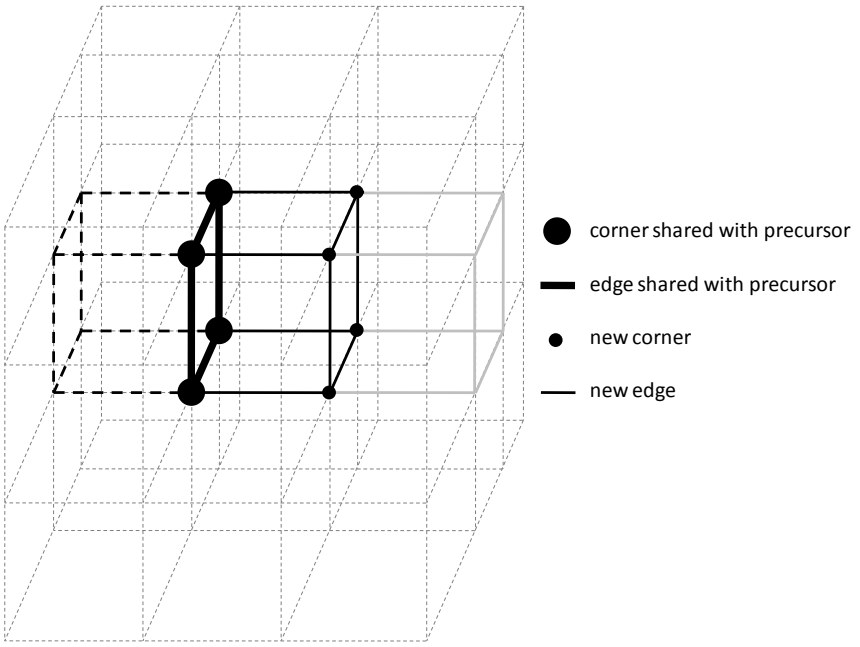


Figure 19: Marching cubes shared corners and edges

The current algorithm first evaluates the 4 new corner nodes and checks if their values are below the isolevel. From the results of the 4 old and 4 new corners it creates the 8-bit key that identifies the topology of the local mesh. First this key is used as index in an edge-table to obtain another key that specifies for each edge if it intersects the isosurface. For all new edges that intersect the isosurface, a vertex, inclusive interpolated position and normal, is generated and stored in a vertex-buffer. Then the same 8-bit key is used as index in a triangle-table to obtain the local vertex indices of the triangles, which are converted to global vertex indices and stored in an index-buffer. The vertex and index buffers are later used in a usual indexed draw call to visualize the generated isosurface mesh.

Efficient volumetric density field construction

As stated before, marching cubes is intended to work on discrete volumetric scalar fields. In order to use it as surface visualisation technique for particle based simulations, thus, it is necessary to create such a density grid first. For this task the application manages a grid that adapts its size and position to the volume occupied by the fluid particles. Each particle “renders” a footprint of its density into the grid. In the simple version of the algorithm a cubic set of voxels is determined that contains all voxels that may be affected by the density of the particle. For each voxel the density contribution of the particle is approximated by evaluation of the SPH density equation at the voxel centre. This density contribution is added to the related overall density value of the voxel.

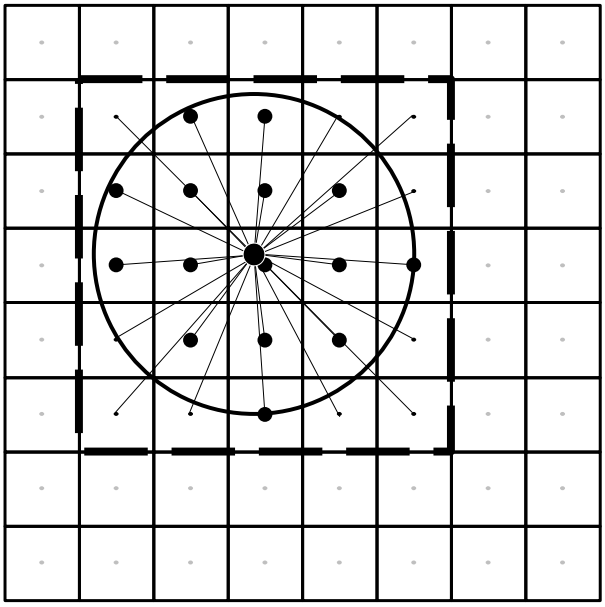


Figure 20: Simple contribution of particle density

Density stamps

A second, more advanced density distribution method uses so called density stamps to avoid the repeated evaluation of the density equation for every particle-voxel interaction. It assumes that the voxels are so small, that the movement of a particle inside a single voxel does not result in a noticeable change in the density distribution among the voxels. The algorithm precomputes a stamp of the density distribution. The stamp is a set of density values for an imaginary set of voxels that is really influenced by a particle. The stamp must only be updated if the smoothing length or the voxel size changes. For density distribution with a stamp, it is only relevant in which voxel the particle is located. The stamp values then are added directly to the corresponding voxels. A somehow sophisticated stamp data structure (which will not be further discussed here) is necessary to access only those voxels that are really influenced by the particle.

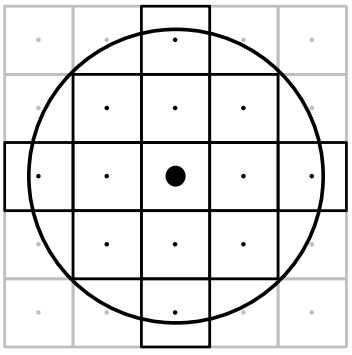


Figure 21: Stamp creation

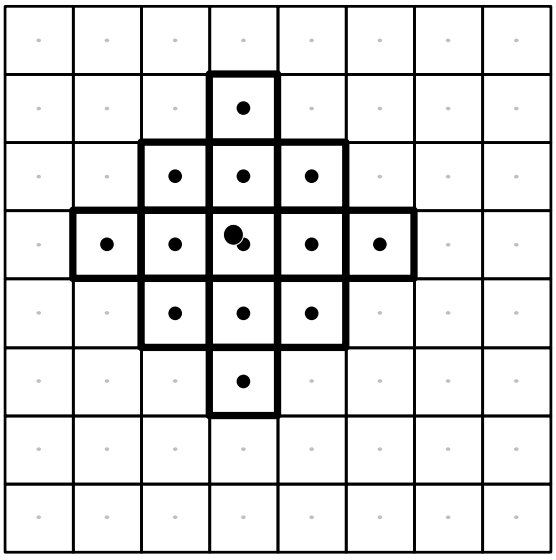


Figure 22: Contribution of particle density with stamp

Results

The program component based on the presented algorithm is capable to perform all necessary steps, from filling of the grid to construction and rendering of the mesh, at interactive frame rates. With deactivated simulation, the whole chain can be executed 150 times per second on a single 3.2 GHz CPU (no multithread support) for 1728 particles and a grid voxel-size of 0.9^3 (compared to a volume size of ~22^3). However, this is a high amount of the total computation time, as the frame rate only drops to 113 frames-per-second (fps) when the simulation is activated too. Especially the voxel-size has a great impact on the computation time: A voxel-size of 1.8^3 results in 499 FPS, while only 17 FPS are achieved with a voxel-size of 0.4^3. An interesting fact is that the mesh construction via marching cubes does need less time than the fill of the grid: 295 FPS with mesh construction only versus 220 with stamp-based grid-fill only. The actual rendering of the mesh needs nearly no time in comparison, as there is no measureable difference between the FPS of the rendering only and the program doing “nothing” (both ~800 FPS).

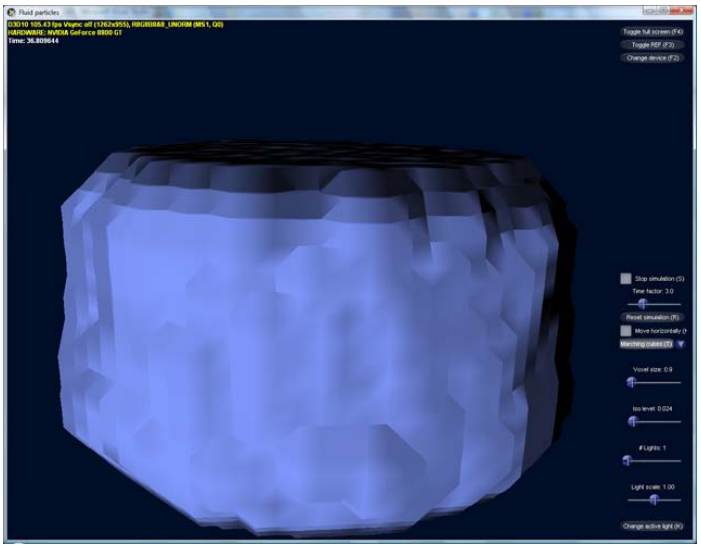


Figure 23: Resulting marching cubes surface visualisation (voxel-size 0.9^3)

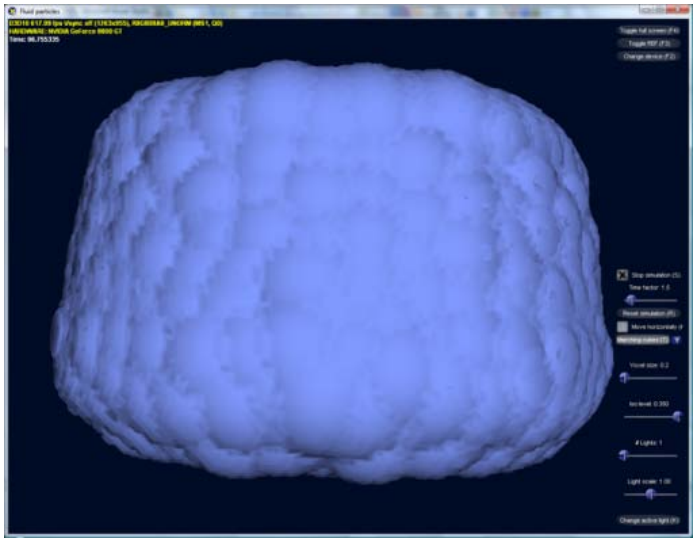


Figure 24: Marching cubes visualisation (voxel-size 0.1^3)

Pros and cons

This shows that the marching cubes algorithm may be an adequate technique for visualisation of static volume datasets, where new isosurface meshes must be constructed only sometimes. As visualisation for the output of a realtime liquid simulation, however, the time needed for the generation of the density grid and the isosurface mesh quickly becomes a bottleneck. Also the visualisation quality remains a problem. Despite the interpolation of intersection positions and normals, the generated meshes still look square-cut. Even with fine-grained voxelisation, which soon results in heavily over-tessellated meshes, the grid-origin or the visualisation remains visible. To name also advantages of marching cubes, a triangle mesh is exactly the sort of geometric model for which realtime 3D-APIs and graphics cards are optimized for. Hence, there is a bunch of techniques that could be used to let the mesh look like a water surface. Sadly, one important optical effect of water, namely refraction, is hard implement with triangle raster graphics.

Further work

To improve the visual quality of marching cubes based rendering, some mesh-smoothing technique could be used to give the images the smooth and round look that is characteristic for liquids. A great jump in performance could be made if the CPU executed algorithm would be replaced with a GPU pendant. A geometry shader based marching tetrahedra algorithm was presented by Uralsky on the GDC 2006 [Ura06]. An implementation inclusive source code can be found in the actual Nvidia SDK 10. It would not be a problem to combine it with the GPU based density field construction presented in 3.6 to an entirely GPU executed marching cubes visualisation for a particle based liquid simulation.

3.5 GPU-based isosurface ray-tracing

Explicit representations of isosurfaces (like i.e. the triangle meshes of marching cubes) are not the only way to produce visualisations of volumetric data sets. Direct volume rendering (DVR) techniques i.e. project all voxels with different opacities directly on the view plane. In [KW03] Krüger and Westermann demonstrated direct volume rendering that utilizes the high texture sampling performance of modern graphics cards. In contrast to comparable existing solutions that where based on several view-plane aligned slices of 3D-textures, they proposed an algorithm that samples the volume texture along the view rays (volume raycasting). This not only resulted in a DVR implementation, which is more efficient because it skips occluded voxels, it also opened the door for GPU based isosurface ray-casting.

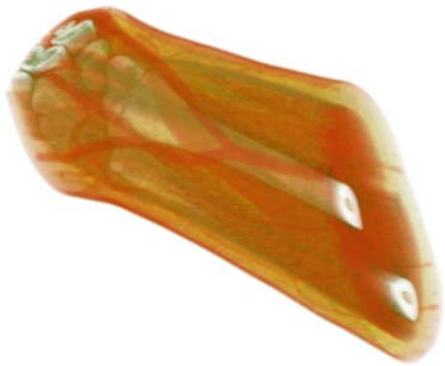


Figure 25: Direct volume rendering of a CT scan with color coding of different densities Source: wikipedia.org

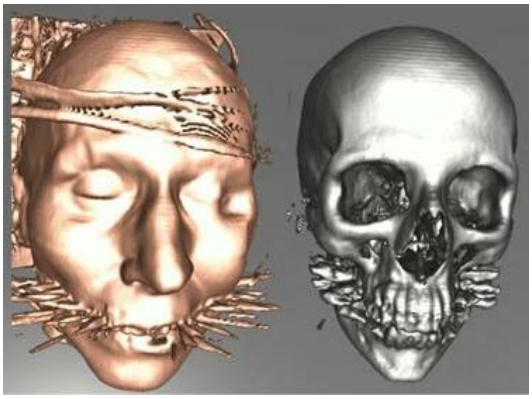


Figure 26: Isosurface-raycast renderings Source: [KW03]

Isosurface raycasting, basic principle

In principle, the algorithm works as follows: For every pixel of the final image that would show the border of the volume a view ray is calculated (backward ray-tracing). Starting from the entry intersection with the volume border, the data set is being sampled repeatedly at incremental positions along the view ray. The procedure stops when a sample value is found that is greater than the isolevel (or enough opacity has been accumulated, in case of DVR). In this case a refinement operation is invoked that tries to find the best position for the isosurface between the current and the last sampling point, in a binary-search like manner. In the case that no sample greater than the isovalue is found, it is assumed that the ray does not hit the isosurface.

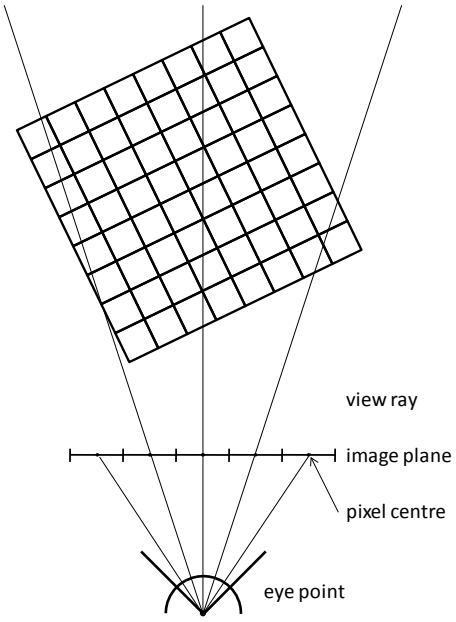


Figure 27: Ray construction

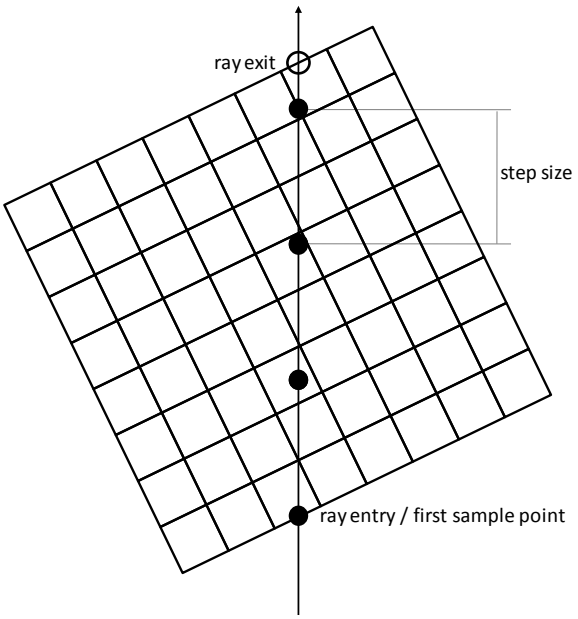


Figure 28: Sampling along the ray

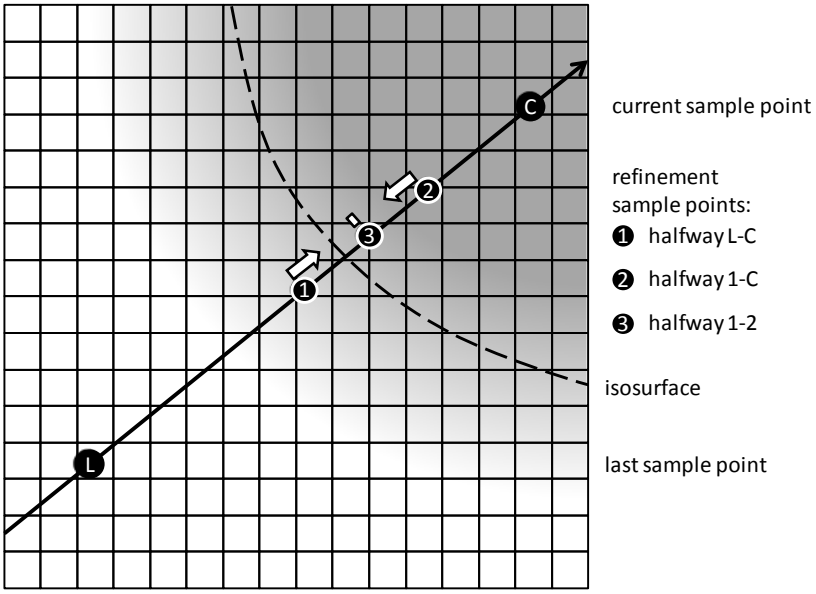


Figure 29: Hit refinement

Volume raycasting with pixel shader 2.0

The application discussed in [KW03] implemented all essential parts of the raycast algorithm as shader model 2.0 pixel shader. The restrictions of this earlier API version lead to a multi-pass rendering technique, which relied heavily on the render-to-texture functionality to transfer intermediate results between subsequent rendering passes. The first pass was used to render the back-faces of a bounding box that represents the volume border. Note that the rasterisation of the bounding box creates only fragments for pixels that would result in a ray which really hits the volume (compare Figure 27). The rasterizer interpolates the (3D-) texture space coordinates of the box vertices and the pixel-shader puts them out to a render-target with the dimensions of the final image.

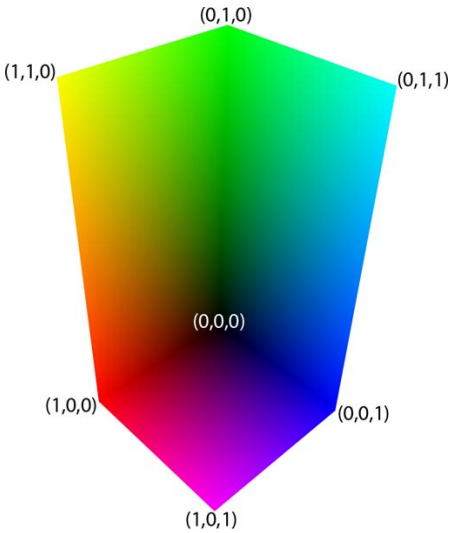


Figure 30: Ray exit texture

In the second pass the front faces of the bounding box are rendered and the texture coordinates are interpolated once again. The invoked pixel-shader uses the normalised device coordinates of the fragment (x, y) to sample the texture that was the render-target of the first pass. The exit- (first pass) and entry (second pass) positions of the ray, presented by the 3D-texture coordinates, are used to calculate (texture space-) ray direction (xyz) and -length (w), which are outputted to the next render-target. This information is sampled and used in pass 3 and all odd following passes, which perform the actual raycasting. Again, they render only the front-faces and use the ray-entry $\mathbf{r}_0$ (still: interpolated per-vertex texture coordinates) and ray-direction $\mathbf{d}$ to calculate volume sampling points with the parameterized ray equation: $r(t) = \mathbf{r}_0 + \mathbf{d}t$. The step parameter t is calculated from the step-size and the number of already performed steps and is stored in a constant, to be available in further raycast passes. When the volume texture's size is not equal on every dimension, the step size must be calculated individually for each pixel. This is because the transformation of the step-size from world to texture space (which ranges only from [0, 0, 0] to [1, 1, 1]) leads to different lengths depending on the ray direction, in this case.

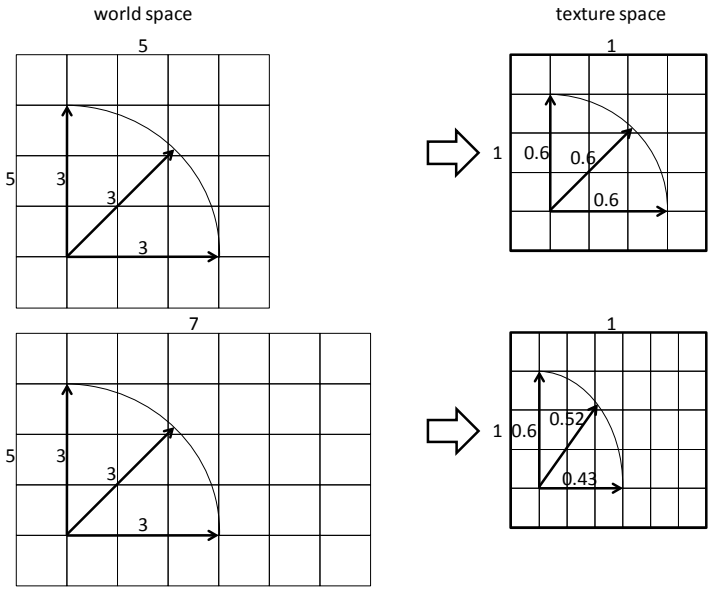


Figure 31: Reason of per pixel step-size

Each raycast pass collects M samples from the volume texture. In case of direct volume rendering the samples are accumulated and the opacity is raised. In case of isosurface rendering, the sampling is done back-to-front in every pass (while execution order of the passes remains front-to-back) and only the last position where the isovalue was reached is kept. Thereby the first sample (in direction of the ray) that matches the isovalue is found. After every raycasting pass, an intermediate pass checks if the stop criterion (enough density reached / isovalue reached / exit passed) is met. If so, it sets the z-buffer to its maximum value, so that the early z-test prevents all further passes for this pixel.

Luckily the capabilities of the pixel-shader API have been greatly improved since 2003. The shader model 4.0, which was used for the isosurface ray-tracing component of this project, offers some pixel-shader features that make the implementation of the raycast algorithm far more intuitive and improve its performance. First of all, the discard command, which allows the pixel-shader to “kill” the current fragment at any time, makes the intermediate stop pass completely obsolete. Second the (theoretically) unlimited length of shader-code and the branching features (introduced with SM 3.0, but greatly improved in 4.0) allow the execution of the whole ray-tracing (this time we will cast secondary rays too) in a single pass. This eliminates the extra texture sampling that was needed before to transfer information between subsequent passes. The only extra pass that remains, is the initial determination of the exit point via rendering of the back-faces, because it is still the most promising way to that.

With this greater flexibility in the pixel-shader it is possible to extend the ray-casting algorithm to simple ray-tracing. Ray-tracing is the basis for the simulation of some important water related optical phenomena in the presented visualisation technique. Therefore, we will first discuss the optical characteristics of water and how ray-tracing could be used to simulate them, before we get into the details of the shader program.

Optical characteristics of water surfaces

Figure 32: Real water in a non-imaginary glass

As stated some chapters earlier, clear water is quite transparent on short distances and so it makes sense to do not render the inner of the water volume at all. The surface of the water, however, has a strong influence on its overall look. Two optical phenomena occur when light hits the border between water and air: Reflection and refraction. Dense materials, such as water, have a higher index of refraction like less dense materials. The refractive index is a measure for how much the medium reduces the speed of waves (i.e. light) that travel through it. In a medium with refraction index $n = 2$ for example, light travels only with half of its speed in vacuum (vacuum: $n = 1.0$). Note that the velocity of propagation in a medium, and therefore also the refractive index, depends on the type of radiance and on the particular wavelength. When a ray of light arrives at a border where the refractive index changes, a part of the light is reflected back into the current medium and another part passes the barrier, but not without a change of direction. To describe the interaction of light

with such a surface, three values must be assessed: The direction of the reflection ray, the direction of the refraction ray and the ratio of reflected versus refracted amount of light.

Law of reflection

The first is the simplest one. The law of reflection states that incoming ray, reflected ray and surface normal lie in the same plane and that the angle between incoming ray and normal is the same as between reflection and normal.

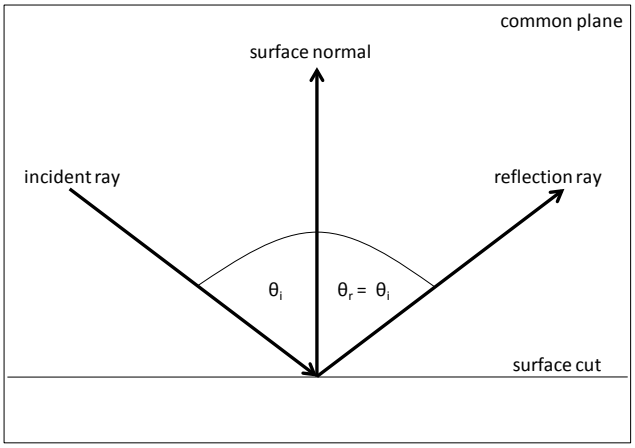


Figure 33: Law of reflection

In 3D computer graphics we treat directions mostly as normalised three-component vectors. Hence, the law of reflection has to be formulated in vector notation. A common technique is to define the reflection vector as sum of the scaled light- and normal vectors:

$$\mathbf{r} = \mathbf{a} + \mathbf{b} = a\mathbf{l} + b\mathbf{n} \tag{3.1}$$

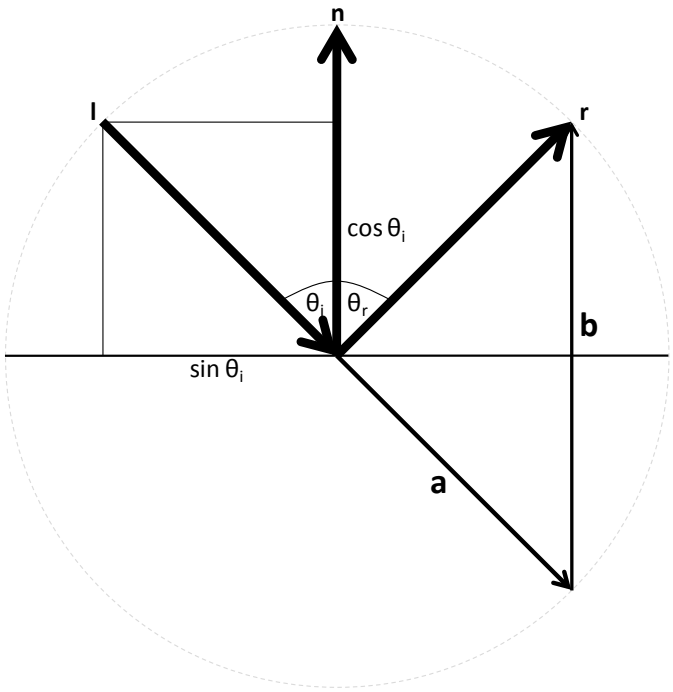


Figure 34: Vector form of reflection

Taking into account that the dot-product of two normalised vectors equals the cosine of the angle between them, Figure 34 results in the following equation for the reflection direction:

$$\mathbf{r} = \mathbf{l} - 2(\mathbf{n} \cdot \mathbf{l})\mathbf{n} \tag{3.2}$$

Snell's law

Also the refraction ray lies in the same plane as incoming ray and normal. In contrast to reflection, however, the direction of refraction depends on the refraction indices (therefore the name) of the participating materials. It is described by Snell's law (named after its discoverer, Willebrord Snellius), which is also called law of sines or law of refraction:

$$\frac{\sin\theta_i}{\sin\theta_t} = \frac{v_i}{v_t} = \frac{n_t}{n_i} \quad \Leftrightarrow \quad \theta_t = \arcsin\left(\frac{n_i}{n_t}\sin\theta_i\right) \tag{3.3}$$

θ_i: angle between incoming ray and surface normal

θ_t: angle between refraction ray and negative surface normal

v_i: light velocity in first medium

n_i: refraction index of first medium

Note: refraction direction is named **t**, from transmission, to not be confused with **r**

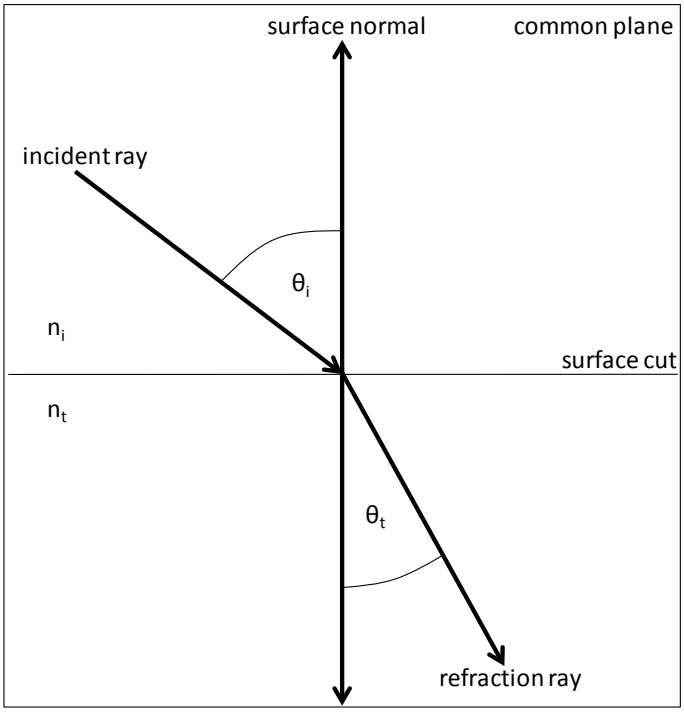


Figure 35: Snell's law

The index of refraction depends on the wavelength (except for vacuum), which generally results in different refraction directions for different light colors. This phenomenon is called dispersion and can be observed at optical prisms or rainbows. Because the effect is barely noticeable in our scenario, we will ignore it and treat the refraction indices as constant.

The careful reader may have noticed that there is not always a solution to Snell's equation. When light hits a material with lower index of refraction than the current, the term $\frac{n_i}{n_t}\sin\theta_i$ can become greater than one, which is not in the domain of the arcsine function. In this case the ray undergoes total internal reflection, meaning all light is reflected and nothing is refracted at all. This coincides with the later equations, which describe how much light is reflected and refracted. In the special case when the term becomes exactly one, θ_i is called the critical angle and the refracted light travels directly along the surface.

As before, Snell's law must be formulated in vector notation, before it can be used in our shader program. We can use the technique from equation (3.1) here too and write the refraction direction as sum of the scaled incident- ($\mathbf{a}$) and normal- ($\mathbf{b}$) vectors. The following drawing illustrates this:

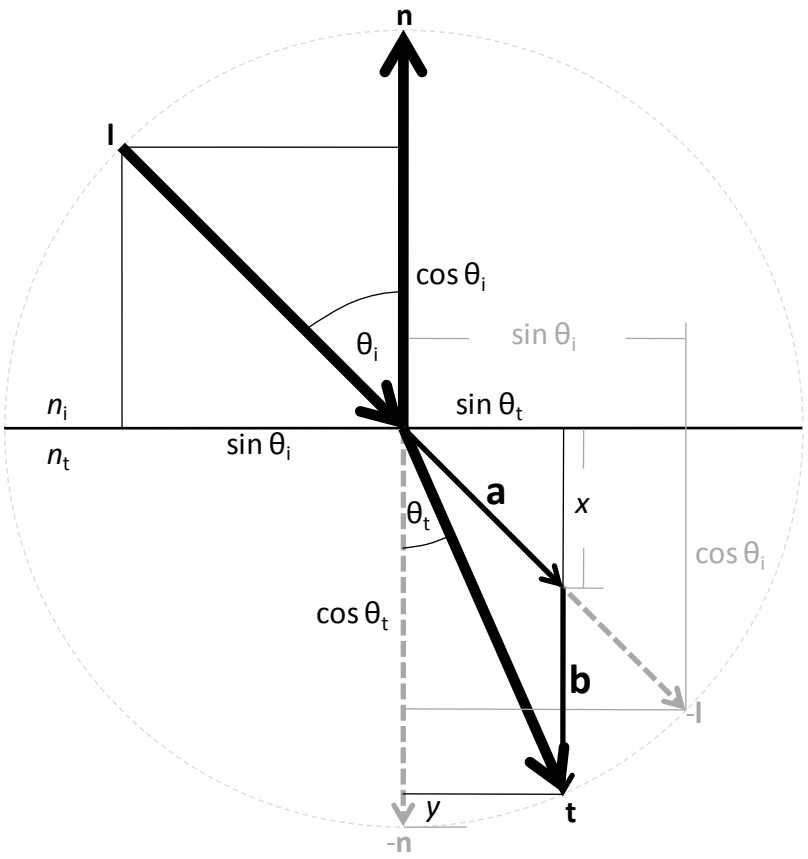


Figure 36: Vector form of refraction

First we search for $a = |\mathbf{a}|$, which is the factor $\mathbf{l}$ must be scaled with. From geometry we know $\frac{|\mathbf{a}|}{|-\mathbf{l}|}$ must equal $\frac{\sin\theta_t}{\sin\theta_i}$. $\mathbf{v}$ is normalised therefore $a = \frac{\sin\theta_t}{\sin\theta_i}$. Snell's law says the quotient of the sines equals the inverse quotient of the refraction indices, so we get:

$$a = \frac{n_i}{n_t} \tag{3.4}$$

To determine $b = -|\mathbf{b}|$ we subtract x and y from $|-\mathbf{n}| = 1$:

$$b = -|\mathbf{b}| = -(1 - x - y) = x + y - 1 \tag{3.5}$$

x can be obtained through scaling of $\cos\theta_i$:

$$x = \cos(\theta_i)\frac{\sin\theta_t}{\sin\theta_i} = -(\mathbf{n}\cdot\mathbf{l})\frac{n_i}{n_t} \tag{3.6}$$

y can be acquired from $\left|-\mathbf{n}\right| - \cos\theta_t$. Application of Snell's law and consequent use of the rule $\cos^2 x + \sin^2 x = 1$ lead to the result:

$$\begin{aligned} y = |-\mathbf{n}| - \sqrt{1-\sin^2\theta_t} &= 1 - \sqrt{1 - \left(\frac{n_i}{n_t}\right)^2 \sin^2\theta_i} \\ &= 1 - \sqrt{1 - \left(\frac{n_i}{n_t}\right)^2 (1-\cos^2\theta_i)} \\ &= 1 - \sqrt{1 - \left(\frac{n_i}{n_t}\right)^2 (1-(\mathbf{n}\cdot\mathbf{l})^2)} \end{aligned} \tag{3.7}$$

Combining equations (3.1), (3.4), (3.5), (3.6) and (3.7) finally leads to the refraction direction in vector form:

$$\mathbf{t} = \frac{n_i}{n_t}\mathbf{l} - \left(\frac{n_i}{n_t}(\mathbf{n}\cdot\mathbf{l}) + \sqrt{1 - \left(\frac{n_i}{n_t}\right)^2 (1-(\mathbf{n}\cdot\mathbf{l})^2)}\right)\mathbf{n} \tag{3.8}$$

Fresnel term

We now know both rays on which the light from the incoming ray can be split, but we still have to decide how it is distributed among the two directions. This distribution is given by the reflection coefficient R and the transmission coefficient T. It is sufficient to define only R, because zero absorption is assumed ($R + T = 1$). Without going to deep into optics, it should been remarked that the reflectance (intensity reflection coefficient) is the square of the amplitude reflection coefficient: $R = r^2$. The intensity of the reflected light depends on its polarisation relative to the plane in which the three rays lie. The reflection coefficient for light polarised parallel to the plane (p-polarised) is called R_p. The one for light polarised perpendicular to the plane (s-polarised) is called R_s. They are given by the Fresnel equations (deduced by Augustin-Jean Fresnel):

$$R_s(\theta_i) = R_\perp = r_\perp^2 = \left(\frac{n_i\cos\theta_i - n_t\cos\theta_t}{n_i\cos\theta_i + n_t\cos\theta_t}\right)^2 = \left(-\frac{\sin(\theta_i-\theta_t)}{\sin(\theta_i+\theta_t)}\right)^2 \tag{3.9}$$

$$R_p(\theta_i) = R_\parallel = r_\parallel^2 = \left(\frac{n_i\cos\theta_t - n_t\cos\theta_i}{n_i\cos\theta_t + n_t\cos\theta_i}\right)^2 = \left(\frac{\tan(\theta_i-\theta_t)}{\tan(\theta_i+\theta_t)}\right)^2 \tag{3.10}$$

θ_t is still acquired from θ_i with Snell's law. Because these equations are undefined for $\theta_i = 0$ this limit applies:

$$R(0) = R_s(0) = R_p(0) = \frac{(n_i - n_t)^2}{(n_i + n_t)^2} \tag{3.11}$$

The following plot illustrates both coefficients once for $\frac{n_i}{n_t} < 1$ and once for $\frac{n_i}{n_t} > 1$:

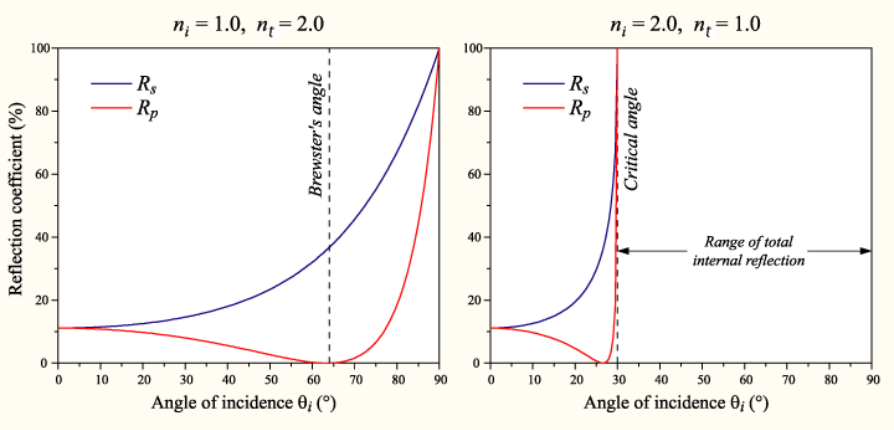


Figure 37: Fresnel reflection coefficients
Source: Wikimedia commons

For unpolarised light (containing an equal mix of s- and p-polarisations) the reflection coefficient is:

$$R = \frac{1}{2}(R_s + R_p) \tag{3.12}$$

According to [NVF02], substitution of (3.3), (3.9) and (3.10) into (3.12) yields:

$$R = \frac{1}{2}\frac{\sin^2(\theta_i - \theta_t)}{\sin^2(\theta_i + \theta_t)}\left(1 + \frac{\cos^2(\theta_i + \theta_t)}{\cos^2(\theta_i - \theta_t)}\right) \tag{3.13}$$

It can be simplified by defining c and g as:

$$c = \cos(\theta_i)\frac{n_i}{n_t} = -(\mathbf{l} \cdot \mathbf{n})\frac{n_i}{n_t}, \qquad g = \sqrt{1 + c^2 - \left(\frac{n_i}{n_t}\right)^2} \tag{3.14}$$

This results in an equation, which is known as "Fresnel term" in computer graphics:

$$F = R(\theta_i) = \frac{1}{2}\left(\frac{g - c}{g + c}\right)^2\left(1 + \left(\frac{c(g + c) - \left(\frac{n_i}{n_t}\right)^2}{c(g - c) + \left(\frac{n_i}{n_t}\right)^2}\right)^2\right) \tag{3.15}$$

Note: Different formulations of the Fresnel term exist.
Each defines c and g slightly different.

[NVF02] suggests to use computational less expensive polynomial approximation of (3.15) in shader programs:

$$R(\theta_i) \approx R_a(\theta_i) = R(0) + \big(1 - R(0)\big)\,(1 - \cos\theta_i)^5 \tag{3.16}$$

Figure 38 shows how far this approximation is from equation (3.15):

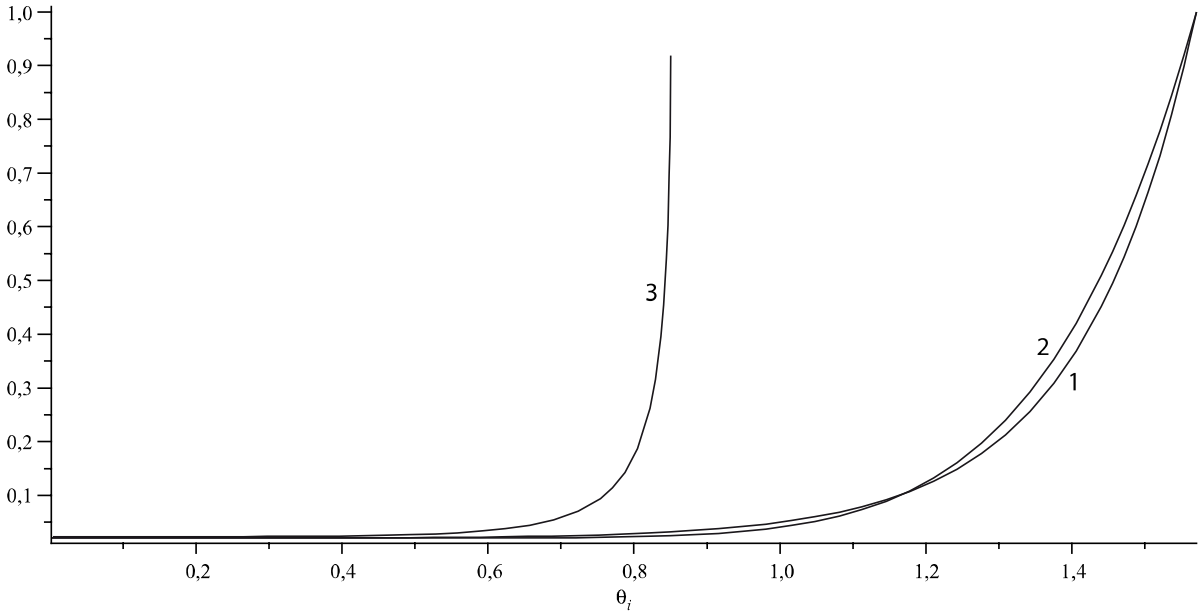


Figure 38: Fresnel term approximation quality

1: $R(\theta_i)$ for $\frac{n_i}{n_t} = 0.75$

2: $R_a(\theta_i)$ for $\frac{n_i}{n_t} = 0.75$ and $\frac{n_i}{n_t} = \frac{1}{0.75} = 1.3333$

3: $R(\theta_i)$ for $\frac{n_i}{n_t} = 1.3333$

Apparently R_a is a good approximation when $\frac{n_i}{n_t} < 1$. But the plot also shows why it is not usable for $\frac{n_i}{n_t} > 1$: R_a delivers the same result when n_i and n_t are swapped. For this reason still the complete Fresnel term is used in the second case.

Description of the overall effect

Law of reflection, law of refraction and Fresnel term altogether describe how the optical behaviour of the water surface is simulated at the first intersection with the view ray. The visualisation thereby utilises the general interchangeability of light ray and view ray in lighting calculations. Water has a higher index of refraction than air and so Snell's law bends the refraction of the view ray towards the negative surface normal. Trough the mostly round shape of the surface, the water body, therefore, appears like a magnifying glass. Due to the Fresnel term, reflections are mostly visible at the silhouette, where the view ray is almost orthogonal to the normal. At the inside the refractions prevail. Figure 39 illustrates directions and intensities of the involved rays, when light hits a water surface from air direction. Figure 40 does the same for light from water direction. For backward

raytracing the light direction simply can be exchanged with the view direction. In this case the refraction coefficient describes the secondary ray's percentage of the cumulated overall intensity.

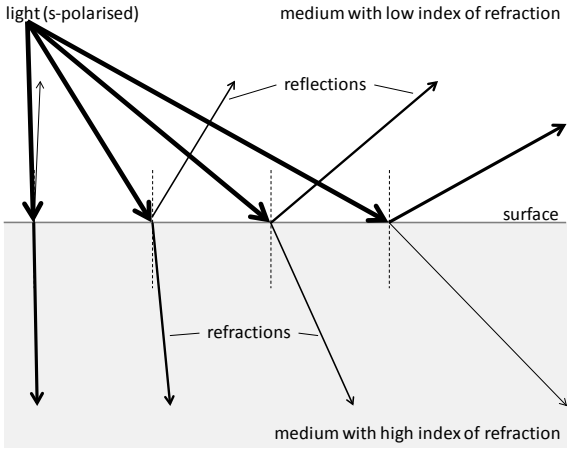


Figure 39: Light hits surface from air direction

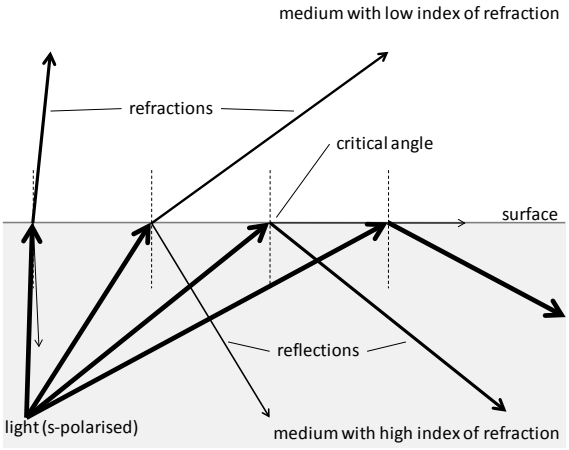


Figure 40: Light hits surface from water direction

The following photographs show the discussed effects at real surfaces.

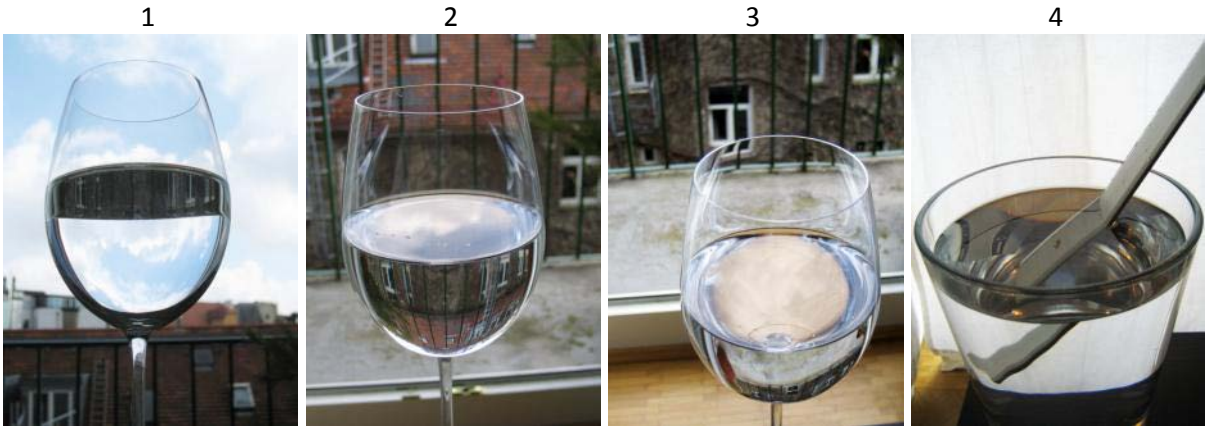


Figure 41: Water surface - optical effects

1: water to air + low view angle -> total internal reflection
2: low view angle -> reflection dominates; strong refraction on round surface -> mirror-inverted
3: higher view angle -> refraction dominates
4: view rays bend down

Ray-tracing depth

Let us now slowly come back to the actual shader program. What happens with the calculated secondary rays? The reflection ray (or better: its direction) is used to perform a texture lookup in an environment map. If the same would be done with the refraction ray, we would only visualise the "front-face" of the water surface. This would not be physical plausible, would not result in an authentic look and besides, it would be possible with conventional raster graphics just as well. However, the visualisation component was called ray-tracer and not ray-caster. This is, because actually the refraction ray is getting traced too. In a second ray-cast step, the exit-intersection of the

view ray with the isosurface is being identified. When the water surface is closed and no other object is contained inside, it is clear that such a second intersection must exist. This time, the first position is searched where the volume exhibits a density value *below* a certain, slightly higher isovalue.

When the second intersection is found, in principle the same steps as for the first must be performed. The subsequent internal reflections, however, would lead to a great number of additional ray-casting steps. Hence, a simplification is made and the internal reflection ray is assumed to never hit the isosurface. The colour values for the internal reflection ray and the exit-ray are sampled directly from the environment map. Although this is not exactly physical plausible, it leads to optically appealing results. The "back-face" of the water surface is visible and the magnifying effect functions as desired. Without the outgoing refraction, the transmitted view ray would be redirected way too much. The slightly incorrect internal reflections do not annoy, because our brain could not process them anyway. Smoothing (i.e. trough use of a low detail MIP level) can additionally improve their appearance. The following figure shows all calculated rays:

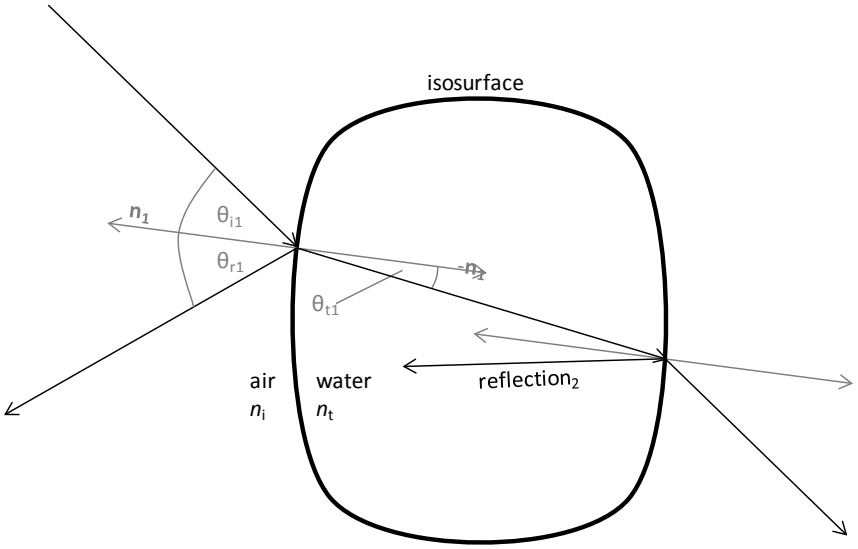


Figure 42: Calculated rays

Also the special case of total internal reflection must be taken into account. When there is no outgoing refraction possible only the internal reflection is considered. The three (or two) environment lookups mark the end of the ray-tracing activity. No additional intersections are calculated, even if in principle the isosurface could be hit much more often than two times.

Implementation

After the basic introduction to isosurface ray-casting, we have discussed so far *what* the program simulates with it and *which equations* it uses. Now we will focus on *how* it does this all. This is the actual render method of the ray-trace visualization component:

Listing 2: Ray-trace FrameRender method

```
void fp_RenderRaytrace::OnD3D10FrameRender(
        ID3D10Device* D3DDevice,
        const D3DXMATRIX*  View,
        const D3DXMATRIX*  Projection,
        const D3DXMATRIX*  ViewProjection,
        const D3DXMATRIX*  InvView,
        bool UpdateVis) {
    RenderEnvironment(D3DDevice, View, Projection);
    if(UpdateVis)
        FillVolumeTexture(D3DDevice);
    RenderVolume(D3DDevice, View, ViewProjection, InvView);
}
```

The “RenderEnvironment” method in our case renders a simple skybox with the same environment map that is used for the reflected and refracted rays. However, just as well it could be a method that generates even this environment map from the actual surrounding 3D environment. The “FillVolumeTexture” method, which apparently is only executed when the fluid simulation has been updated, is responsible for the generation of the volume texture that contains the density field. It will be presented separately in the next subchapter. For now we will concentrate on the “RenderVolume” method, which does everything what has been discussed in the current subchapter.

It was said earlier that only two rendering passes are used. Hence, the C++ method simply renders the bounding box of the volume two times, each one with different rendering-passes (pass in the sense of the DirectX effect framework). The actual work is done entirely in the shader programs. The first time only the 3D texture coordinates of the back-faces (aka: the ray exit points) are rendered to a separate texture. The second pass, which renders the front faces to the final image, contains the actual ray-tracing. Just like during the first pass, the vertex-shader calculates the vertex’s positions in 3D-texture- and world-space. The rasterizer interpolates both for each generated fragment. Additionally it transforms the world-space position to screen space. Thus, it can be used in the pixel shader to directly load the ray-exit point from the texture that was generated in the first pass. The starting point for the ray-tracing of the fragment, therefore, is the entry and the exit intersection of the view ray with the texture volume.

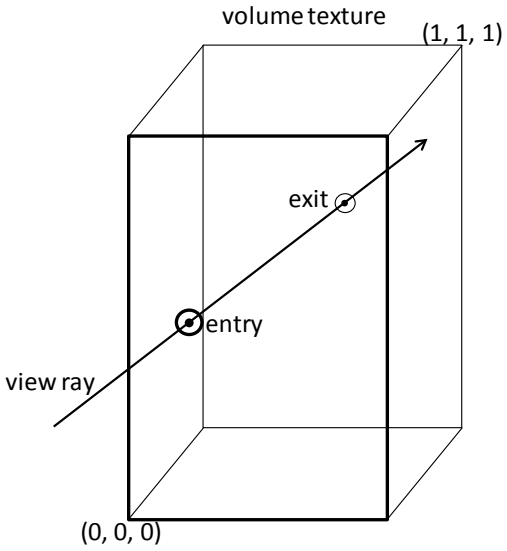


Figure 43: Ray-tracing starting point

When the texture dimensions are equal on each axis, the step size is taken from a constant. Otherwise it is calculated individually for the current ray segment from entry to exit. An offset vector with direction of the ray and a length equal to the step size is calculated. The entry is the first sample point. In a loop the texture is being sampled (with tri-linear interpolation) at the current sample point and the gathered value is compared to the iso-value. If the sample-value is equal or higher, the loop stops. Otherwise the offset is added to the sample point and the loop continues. If no intersection is found at all, the loop stops before the sample point would leave the volume. In this case the exit point is used as last sample point. If also the last sample is below the iso-level, the current fragment is being discarded (discard command).

If a sample value higher than the iso-level is found, the binary-search-based refinement function is invoked (for principle see Figure 29 on side 48). It executes a predefined number of intersection refinement steps. Each step checks a sample point halfway between two previous ones. For this purpose the offset vector is halved after each step and either added to (sample value below iso-level) or subtracted from (sample value above) the sample position. Thanks to the refinement, the step-size can be considerably higher than the voxel size without producing major artefacts.

After the intersection has been found, the normal of the isosurface at this point must be evaluated. This is done by taking the normalised gradient of the density field. The gradient is calculated from 6 additional samples, each a voxel size away.

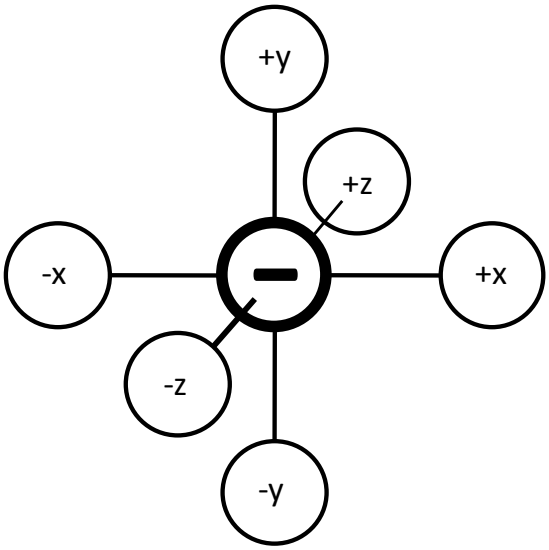


Figure 44: Gradient calculation

Because calculation of the gradient is a memory intensive task, all non-dependant calculations are done before, to take advantage of memory latency hiding effects. (GPUs execute memory access operations as soon as possible, so that other calculations can be done, while waiting for the results). In this program it is only the calculation of the screen-space depth of the first intersection. Nonetheless, this leads to a noticeable performance gain.

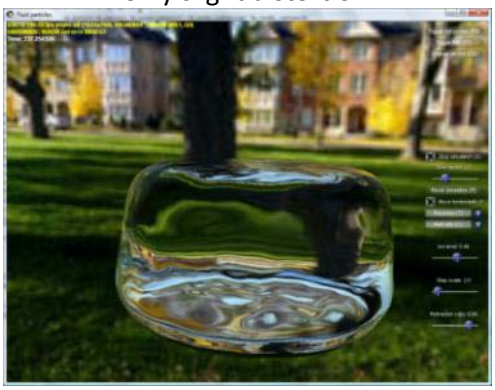
After the determination of the intersection and its normal, the reflection direction is being calculated. This can be done either with the vector form of the law of reflection (3.2), or with the HLSL-intrinsic reflect-function. The direction is used later (memory latency hiding) as coordinates for a cube-map texture lookup in the environment texture, to gain the colour value for reflection 1. The refraction direction is computed according to Snell's law. Again, this can be done with the according vector equation (3.8) or with the intrinsic refract-function. A simple box intersection is then used to determine the volume exit point of the refraction ray. Using the first intersection as start- and the exit as end point, another isosurface raycast is performed, to find the intersection of the refracted ray with the isosurface. Also for the second intersection, the reflection direction is calculated and used to lookup the environment-map. If a refraction ray exists, the refraction direction is calculated too and also used in an environment-lookup. In this case the complete Fresnel term (3.15) is being calculated, to define how the colour values from second reflection and refraction are blended together to the colour of the first refraction. Otherwise, only the reflection colour is used (total internal reflection). At last, the approximated Fresnel term (3.16) for the first intersection is calculated and used as blend factor for the colour values from first reflection and refraction. The earlier calculated depth forms together with the final colour the result that is written to the render-target.

Optical results

The following screenshots show the discussed effects in the final images:

only slight distortion

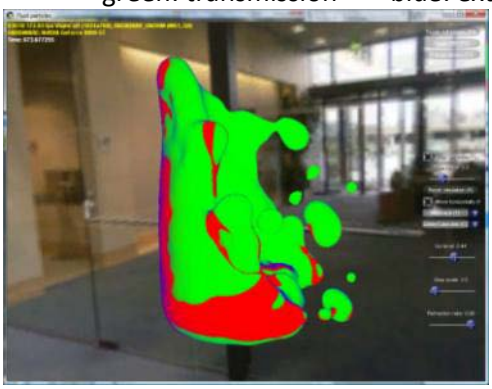
water refraction index



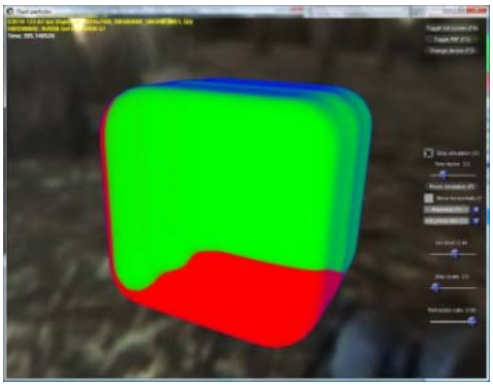
refraction becomes mirror-inverted

diamond refraction index

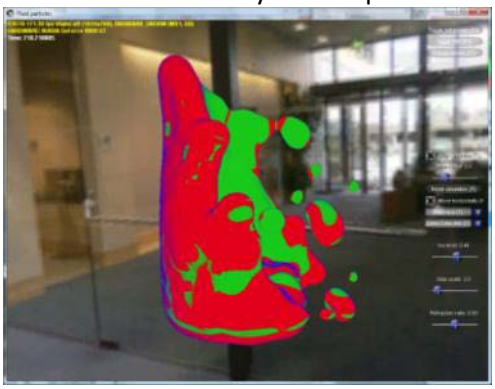
green: transmission blue: external reflection red: internal reflection



internal reflection only on oblique surface



external reflection mainly on top side



heavy internal reflection

slight external reflection everywhere

Figure 45: Isosurface ray-tracing screenshots

Each scene is shown once for a material with low refraction index (upper images) and once for one with higher index. As one could see, the ray-tracer produces much smoother images than the marching cubes visualisation. The visualization also produces significant less grid-related motion artefacts. Only in nearly rest position tiny jumps of the surface remind of the grid background. To say the images look “realistic” would probably be exaggerated. Nonetheless, the renderer achieves an optical quality that, before, was known rather from offline ray-tracers. At all pictures the Fresnel effect is clearly noticeable. Reflections, both external and internal, are mainly visible on parts of the surface which are seen under a plane viewing angle. At the last four images this are the regions in blue for external- and in red for internal reflections. Also it is obvious that the reflectivity rises with increasing refraction index. With the colour-coded screenshots the minor effect of the Fresnel term for the outgoing intersection is visible. Because of the sharp curve of the reflectance function in Figure 38, the transition from total transmission to total internal reflection is very short. If this second, more complex Fresnel calculation is skipped, the differences in the images are hart to notice. However, because its performance cost is hidden by the following texture sampling, the calculation is practically for free.

Performance results

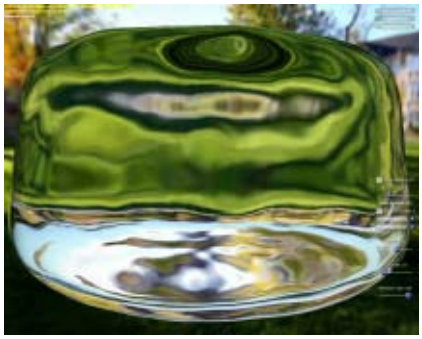


Figure 46: Screen filling at 1280 resolution

Figure 47: 1/9 screen-coverage

The application (x64 binary) achieves interactive frame-rates with running simulation (12^3 particles) and full-screen ray-tracing visualisation with all possible effects. The test machine was a PC with Intel Core 2 Quad Q6600 (3.24 GHz), 2 GB RAM and GeForce 8800 GT graphics card with 512 MB video memory (clock-speeds: core 640 / shader 1600 / memory 950 MHz) running Windows Vista x64. In 1280x1024 full-screen mode the water surface is rendered screen-filling with 65 frames-per-second. The rendering performance depends heavily on the volume bounding box size on the screen. When the visible water surface covers approximately 1/9 of the screen (the actual bounding box is considerably larger) the frame-rate is 161. When the water surface resides outside the visible screen, it raises to 288, which is only 6 % under the 307 FPS of the sprite visualisation. This numbers show that the generation of the density grid on the graphics card (see next subchapter) is no major performance bottleneck this time. An interesting observation was made during the performance testing: The overall performance of the ray-tracing visualisation depends more on core- and shader-clock-speeds of the GPU than on its memory clocks. This was seen as the clock speeds where test-wise lowered individually. But nevertheless the activation or deactivation of longer computations in

shader code, have no influence on the frames-per-second, when they are located before texture lookups. This is a very good example how the performance can profit when one takes care about the effect of memory-latency-hiding.

Pros and cons

As the last section showed, the performance of the visualisation algorithm is sufficient for interactive applications. In this respect ray-tracing is ideal for realtime simulations, as the surface is “generated” only for the current view. Because the number of traced rays depends on size on the final image, the algorithm offers implicit level-of-detail, which also comes in handy with interactive applications. Another quality characteristic is the smooth look of the surfaces on the generated images. Adding the fact that ray-tracing is capable to simulate important optical effects, which are impossible for raster-graphics, allows saying the presented technique stands out for its image quality.

But the technique also has its disadvantages. Grid regions containing only low density (empty space) still need lots of computation time, because no intersection stops the sampling along the traversing rays. The original volume ray-casting paper suggests an octree data structure to mark such empty regions. While this is generally a good approach when the acceleration structure can be precomputed, the cost for its on-the-fly generation may perhaps quickly eat up its benefit during rendering. However, other limitations are more profound than performance drawbacks. The need to contain all visualised fluid in a grid of limited dimensions, for example, clearly constrains the otherwise good spatial flexibility of a particle based simulation. Another issue is the optical interaction with objects of the 3D-environment. The major difficulty thereby is to integrate the ray-tracing / environment-mapping based visualisation into a rasterization based renderer. Because interest in ray-tracing is growing in the realtime graphics community, perhaps we may see more work on this general problem in the future. The next section already introduces some thoughts how these challenges could be faced.

Further work and outlook

It should be evaluated if empty space skipping with traditional octree generation (based on the final density grid) can improve the overall performance. Maybe a coarse-grained empty space structure, generated directly from particle positions, would be more suitable as a realtime updated acceleration structure.

To break the spatial limitations, it would perhaps be a good idea to generate only a single density grid for the whole current view. This could be done, for example, by treating the normalised device space as domain of the volume grid. Grid generation and ray-tracing would have to be adapted to operate within this different coordinate system. The advantage would be that the simulation do not has to bother with encapsulation of each “fluid-cluster” into its own volumetric grid.

For optical interaction with objects drawn by conventional rasterization, two cases are of importance. The first question is, how the colour contribution of outgoing reflected- and refracted

rays could be determined, taking into account the surrounding environment. The conventional solution in realtime computer graphics is to generate dynamic environment maps in this case. However, this reduces once again spatial flexibility, as such an environment map must be computed for a certain space position. More general solutions will probably have to involve some kind of ray-tracing. The second question is, how objects are handled that intersect the water surface or are contained entirely within. The challenge thereby is that these objects in principle must be taken into account during ray-tracing. An idea to do this in a rasterization like way comes from parallax occlusion mapping [Tat06].

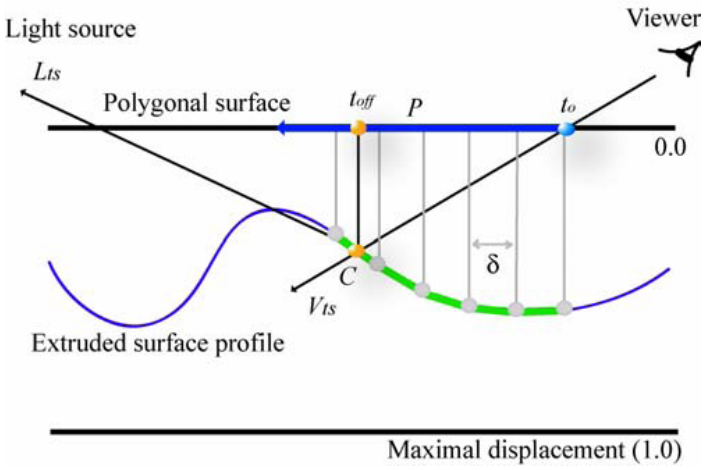


Figure 48: Parallax occlusion mapping principle
Source: [Tat06]

As Figure 48 shows, parallax occlusion mapping in principle is ray-casting of a 2D height field. The same technique could possibly be combined with the isosurface ray-tracer to handle intersections with the 3D environment. The idea is to first render the environment and then use the depth-buffer as height field during the ray-tracing. For internal rays (after first isosurface intersection) intersections would be searched parallel once for the isosurface and once for the height-field. If the height field is hit first, the colour-value could be read from the former frame-buffer. Again, solutions closer to real ray-tracing may provide a "cleaner" way to do this.

This section shows that sophisticated combination of isosurface ray-tracing with conventional raster graphics is quite a challenge. Things would be much easier, when the technique had to be integrated into a ray-tracer. Since ray-tracing seems to grow in its importance for realtime computer graphics, maybe in the future rendering techniques may evolve that harmonise better with such ray-based visualisations.

3.6 GPU-based volumetric density field construction

The description of the ray-tracers implementation began with the actual C++ render method (Listing 2: Ray-trace FrameRender method). It was stated there, that whenever the fluid simulation has been updated, the density grid inside the volumetric texture must be re-generated. When the simulation is running, this is normally the case in every frame. Because of this, the overall performance depends heavily on an as fast as possible “FillVolumeTexture”-method. As the subchapter on marching cubes (3.4) showed, filling of the density grid on the CPU quickly can become a threat to overall performance. However, the algorithm by nature is highly parallelisable and extremely dependant on memory performance, which also could be thought as “fill-rate” in this context. Those are the very best preconditions for a GPU based implementation.

Direct3D 10 offers everything what is needed for this purpose. The new resource system makes it comparatively easy to bind volumetric textures as render targets. However, this does not mean it suddenly offers some mysterious capability to output “volumetric images”. That would contradict the whole concept. Instead it still outputs 2D images, but allows the programmer to delay the decision for a final render-target (2D slice of a 3D texture in this case) until the very last station before the rasterizer. For D3D 10 this is the geometry shader.

Basic principle

This architecture leads to the following slice-based algorithm for the spreading of a single particle’s density in the volumetric texture:

1. find out how much slices of the volumetric density texture are influenced by the particle
2. for every influenced slice construct a square that covers the affected area
3. rasterise the square onto the right slice
4. for every fragment check the distance to the particle centre
   and compute the additional density from the SPH-density-equation

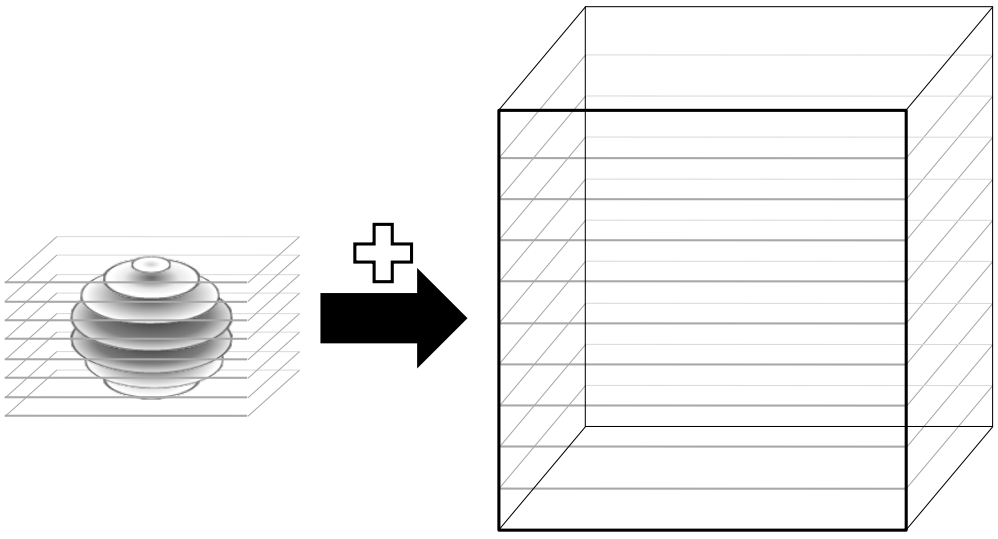


Figure 49: Particle slicing

Implementation

The Direct3D 10 implementation of the algorithm achieves this in the following way: First a vertex-buffer is filled with the world-space particle positions and the densities at these positions (one vector with four components). The volumetric texture is set as render target and the view port is adjusted accordingly. A one-dimensional texture, containing the results of the SPH density equation, is set as shader resource. The texture allows the calculation of the additional density for a fragment by a single texture lookup (with distance to the particle as texture coordinate), followed by a multiplication with the density at the particle's position. The calculation is started with a single instanced draw call, which draws all vertices (particles) as point-list. The count of instances is set to the number of slices that is influenced by one particle (basic principle point 1). This means that all vertices are rendered as often as slices are affected by a single particle.

The rest of the algorithm is implemented in shader code. The vertex shader fulfils three tasks. First it calculates the slice index of the render-target inside the 3D texture. It does this by combining the z-component of the particle position with the instance id.

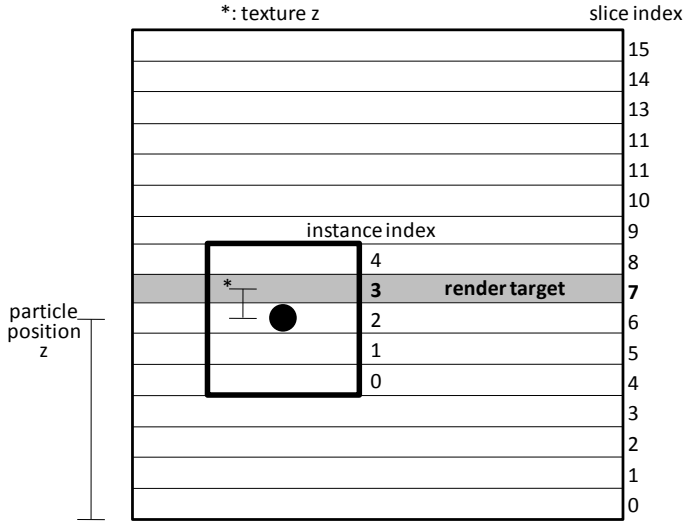


Figure 50: Slice index construction

Additionally it converts the x and y position components from world- to normalised device space (equivalent to volumetric texture space). At last it calculates the offset from particle position to render-target slice ("texture z" in the image), which will be needed at the very end of the algorithm to determine the distance from the fragment to the particle.

A geometry shader follows the vertex shader. It first validates the calculated render-target slice index. (This is done in the geometry shader, because in contrast to the vertex-shader it can also produce no output at all). If the slice index is in valid range, it generates four output vertices, connected in a strip to two triangles, which form a square (basic principle point 2). Each output vertex carries the following attributes: volume slice index (= render target array index; per primitive data -> only interpreted from the leading vertex of each triangle), particle density (equal for all four vertices), vertex position (in normalised device space with z=0) and 3D texture coordinates. The position results from the 2D normalised device space particle position plus an offset vector for each

corner. X and Y texture coordinates are from the array $[(-1,1),(1,1),(-1,-1),(1,-1)]$, while texture z is the offset from particle position to render-target plane that was calculated in the vertex shader.

The two triangles then are rastered on the chosen render-target (3D texture slice), which includes interpolation of the attributes (basic principle point 3). The final pixel shader calculates the length of the texture coordinates vector. This scalar represents $\frac{|\mathbf{r}|}{h}$ in terms of SPH. It is used as coordinate into the texture that holds the results of the density equation. The sampled value multiplied with the particle density finally gives the additional density, which is added to the existing value in the render-target through additive blending (basic principle point 4).

Results

The result of the whole algorithm is a volumetric scalar-valued texture that contains the cumulated densities from all particles. Figure 51 and Figure 52 exemplarily show its content during the running program.

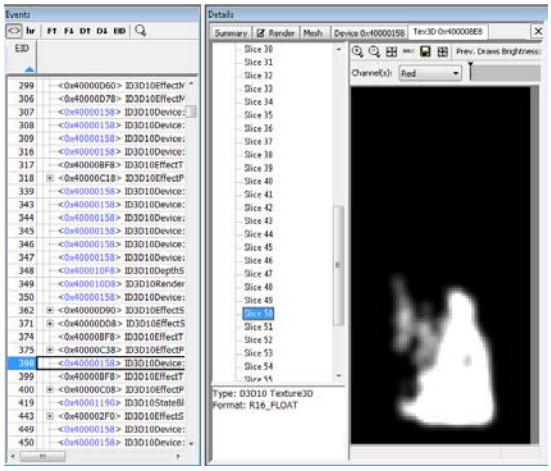


Figure 51: Generated density texture viewed in PIX (a performance analysis tool for Direct3D)

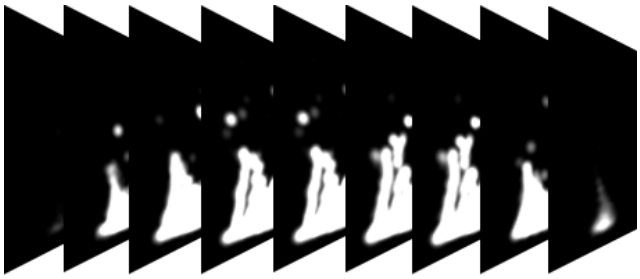


Figure 52: Slices of the volumetric density texture

As expected, the GPU implementation brought an immense performance boost compared to the CPU version. During the test 2016 particles filled a volume texture with a dimension of 64x128x64 (which is completely sufficient for good optical results). The smoothing length of 3.0 resulted in a particle voxel diameter of 13. This means every particle affects 13^3 voxels of the density texture and consequently each particle must be rendered to 13 slices. The test machine runs simulation and density texture fill together with 273 FPS. Omitting the texture fill it runs with 285 FPS. For the texture fill this would lead to a calculated computation time of 0.15 ms and a calculated fill rate of 29.5 billion per second. Even if this is a very high number, it seems possible compared to the theoretical fill rate of 33.6 billion that Nvidia specifies for this graphics card type. To give also a more conservative assumption, the whole program is executed with 614 FPS with texture fill only and 850 FPS without doing anything. This would result in a texture fill time of 0.45 ms and a fill rate of 9.8 billion per second, which is still a huge factor faster than the CPU version.

3.7 Alternatives and further work

The presented techniques for rendering of liquid surfaces, marching cubes and isosurface ray-tracing, are only two out of many different solutions. The existing techniques differ in the geometric primitives they are based on, in their rendering technique and in the question if they are generally view-dependent or do create view-independent explicit surface representations. Marching cubes in example uses triangle primitives and constructs a view independent mesh representation, which can be rasterised or ray-traced just as well. Another common primitive to construct surfaces explicitly are surfels (surface elements; surface-aligned sprites or other point-based primitives). The related technique is mostly called surface splatting (see exemplarily [ZPB01] for an introduction) and can be used as well view-dependant or not.

[Kno07] gives an overview on various surface rendering techniques. To evaluate each ones potential for particle-based realtime fluid simulation would go beyond the scope of this thesis. Instead at last we will shortly focus on a technique that was developed for exact this purpose. It is called “screen space meshes” and was introduced by Müller et al. in [MSD07]. It generates a triangle mesh, but this times a view-dependent one. What makes it interesting is the fact that no explicit density grid is needed. In short the algorithm works as follows:

1. generate an approximated depth map of the surface for the current view directly from the particles; i.e. by drawing every particle as (imaginary) sphere to the z-buffer
2. find the “silhouette” of the surface in the depth buffer
3. smooth the depth values
4. generate a 2D triangle mesh for the covered screen area
5. smooth the silhouette of the mesh
6. transform the mesh back to world space
7. render the mesh

Also this technique is not the ultimate solution. But it demonstrates two properties that are important in this very special scenario. First, the entire algorithm does only what is necessary to generate an image of the surface *for the current view*. It generates neither an explicit surface for the whole fluid, nor an explicit density field. As the fluid anyhow changes till the next frame, it is waste to generate structures that allow creation of different views or different surfaces. Second, the generated triangle mesh fits in the overall rendering technique. Visual interaction with the rest of the 3D environment, therefore, can be achieved with common raster-graphics techniques.

Perhaps further visualisation approaches should build upon these two principles and try two overcome potential drawbacks of the current solution. The smoothing of depth values and mesh silhouettes for example is already a method to conceal the fact that no real isosurface is generated. Without smoothing, the forms would not fade into each other, like they do because of the summation of densities in a grid. Questionable is, if it is really necessary to generate a triangle mesh. Good surface rendering techniques may be possible on basis of height-fields just as well. The

restricted capabilities for reflections and refractions could be extended with existing technologies. [KBW06] for example presents interactive screen space photon tracing on GPUs, which is capable to render water reflections and refractions, as well as the related caustics and god rays. This technique would be an ideal combination, because it already works on meshes and does not need any pre-computation, so that it can be used on such dynamically created objects.

4 Conclusion

This thesis demonstrated one way to simulate and render fluids in an interactive application. It showed that pleasant results regarding fluid-behaviour and optical appearance are possible on today's hardware. Realtime fluid simulation benefits from ongoing research in this area and from a hardware trend towards massive parallel general purpose computing. It can be assumed that we will see interactive fluids in commercial applications in the near future. When particle-based fluid simulations will be integrated into game-physics-middleware, they may quickly become standard in video-games. Nvidia recently bought Ageia, one major developer of such game-physics systems. Mathias Müller, who contributed a lot to the topic, was head of research at Ageia. It is therefore possible that particle based fluid simulation will be included in the GPU-accelerated version of the PhysX SDK, which Nvidia wants to develop with its CUDA programming framework. Demand for sophisticated visualisation techniques may rise soon, if that really happens.

Appendix

References

[APK07] Adams, et al. 2007. Adaptively Sampled Particle Fluids. *Proceedings of the 2007 SIGGRAPH conference.* 2007.

[AIY04] Amada, et al. 2004. Particle-Based Fluid Simulation on GPU. *ACM Workshop on General-Purpose Computing on Graphics Processors.* 2004.

[BT07] Becker and Teschner. 2007. Weakly compressible SPH for free surface flows. *Proceedings of the ACM SIGGRAPH Symposium on Computer Animation.* 2007, pp. 63-72.

[BM07] Bridson and Müller-Fischer. 2007. Fluid simulation. *SIGGRAPH 2007 course notes.* 2007.

[BE02] Burgess and Elst, van. 2002. MAS209: Fluid Dynamics. *Course Material.* 2002.

[CHJ03] Co, Hamann and Joy. 2003. Iso-splatting: A Point-based Alternative to Isosurface Visualization. *Computer Graphics and Applications, 2003. Proceedings.* 2003, pp. 325-334.

[CEL06] Colin, Egli and Lin. 2006. Computing a null divergence velocity field using smoothed particle hydrodynamics. *Journal of computational physics.* 2006, 217, pp. 680-692.

[GM77] Gingold and Monaghan. 1977. Smoothed particle hydrodynamics: theory and application to non-spherical stars. *Royal Astronomical Society, Monthly Notices.* 181, 1977, pp. 375-389.

[GPGPU] GPGPU.org. [Online] http://www.gpgpu.org.

[Hei07] Heinecke. 2007. Physikalische Rauchsimulation auf Partikelbasis in Echtzeit mit der PhysX-Engine. [ed.] Technische Universität Dresden. [Minor thesis]. 2007.

[KW06] Kipfer and Westermann. 2006. Realistic and interactive simulation of rivers. *ACM International Conference Proceeding Series.* 2006, 137.

[Kno07] Knoll. 2007. A Survey of Implicit Surface Rendering Methods, and a Proposal for a Common Sampling Framework . *Proceedings of the 2nd IRTG Workshop.* 2007.

[KC05] Kolb and Cuntz. 2005. Dynamic Particle Coupling for GPU-Based Fluid Simulation. *Proc. 18th Symposium on Simulation Technique.* 2005, pp. 722-727.

[KW03] Krüger and Westermann. 2003. Acceleration Techniques for GPU-based Volume Rendering. *Proceedings of the 14th IEEE Visualization.* 2003, p. 38.

[KBW06] Krüger, Bürger and Westermann. 2006. Interactive Screen-Space Accurate Photon Tracing on GPUs. *Eurographics Symposium on Rendering.* 2006.

[LC87] Lorensen and Cline. 1987. Marching cubes: A high resolution 3D surface construction algorithm. *Proceedings of the 14th annual conference on Computer graphics and interactive techniques.* 1987, pp. 163-169.

[Luc77] Lucy. 1977. A numerical approach to the testing of the fission hypothesis. *Astronomical Journal.* 82, 1977, pp. 1013-1024.

[Mon05] Monaghan. 2005. Smoothed particle hydrodynamics. *Reports on Progress in Physics.* 2005, 8.

[Mon92] —. 1992. Smoothed particle hydrodynamics. *Annual review of astronomy and astrophysics.* 1992, 30, pp. 543-574.

[MCG03] Müller, Charypar and Gross. 2003. Particle-Based Fluid Simulation for Interactive Applications. *Proceedings of 2003 ACM SIGGRAPH Symposium on Computer Animation.* 2003, pp. 154-159.

[MST04] Müller, et al. 2004. Interaction of Fluids with Deformable Solids. *Computer Animation and Virtual Worlds.* 2004, 15, pp. 159 - 171.

[MSD07] Müller, Schirm and Duthaler. 2007. Screen Space Meshes. *Eurographics/ ACM SIGGRAPH Symposium on Computer Animation.* 2007.

[NVF02] NVIDIA Corporation. 2002. Fresnel Reflection Technical Report. [Online] 2002. http://developer.nvidia.com/object/fresnel_wp.html.

[Pap99] Papanastasiou, Georgiou and Alexandrou. 1999. *Viscous Fluid Flow.* s.l. : CRC Press, 1999. ISBN13: 9780849316067.

[Tat06] Tatarchuk. 2006. Dynamic Parallax Occlusion Mapping with Approximate Soft Shadows. *Proceedings of the 2006 symposium on Interactive 3D graphics and games.* 2006, SESSION: Shader-based rendering, pp. 63 - 69 .

[THM03] Teschner, et al. 2003. Optimized Spatial Hashing for Collision Detection of Deformable Objects. *Proceedings of VMV'03.* 2003, pp. 47-54.

[Ura06] Uralsky. 2006. Practical Metaballs and Implicit Surfaces. *Game Developers Conference 2006 Presentation.* 2006. http://developer.nvidia.com/object/dx10-practical-metaballs.html.

[WND] wikipedia.org. *Navier-Stokes equations/Derivation.* [Online] [Cited: 03 10, 2008.] http://en.wikipedia.org/w/index.php?title=Navier-Stokes_equations/Derivation&oldid=177609104.

[ZPB01] Zwicker, et al. 2001. Surface Splatting. *Proceedings of the 28th annual conference on Computer graphics and interactive techniques.* 2001, pp. 37-378.

Glossary

atomic operations 31
Continuity equation....20
CUDA 69
density volume 38
direct volume rendering....45
Direct3D....35
Eulerian model 15
exploitation of symmetry 28
fluid particle....19
Fresnel term 53
geometry shader 36
gradient 20
incompressibility 33
incompressible flow condition 17
integral interpolation 18
isosurface 38
Lagrangian model 15
Laplacian....20
law of reflection 49
marching cubes 39
mass density....14
multithreading....30
Navier-Stokes equation 18
neighbour search....27
Newton's second law 15
Newtonian fluid 17
parallax occlusion mapping 64
polarisation 53
pressure 14
pressure forces 17
refraction index 50
resource view.... 36
smoothed particle hydrodynamics 18
smoothing function 18
smoothing kernels 23
Snell's law 50
spatial derivatives 19
SPH pressure 21
SPH surface tension 22
SPH viscosity 22
sprites 37
substantial derivative
substantive d., convective d., material d.. 16
surface splatting 67
surface tension 14
viscosity 14
viscosity force 18
volume raycasting.... 46
volumetric density field construction.... 64

List of figures

Figure 1: Example for offline simulation 9
Figure 2: Example for realtime simulation 9
Figure 3: Sprite visualisation 12
Figure 4: Marching cubes visualisation 12
Figure 5: GPU ray-tracing visualisation 12
Figure 6: Cause of surface tension 15
Figure 7: Lagrangian versus Eulerian point of view 16
Figure 8: 1D example for a smoothing kernel 19
Figure 9: Used smoothing kernels 25
Figure 10: Grid based neighbour search 28
Figure 11: Skip neighbour cells on the opposite side 29
Figure 12: Liquid behaviour 34
Figure 13: Direct3D pipeline stages 38
Figure 14: Sprite rendering 39
Figure 15: RGB and alpha channel of the particle texture 40
Figure 16: Billboard rendering with a D3D10 geometry shader 40
Figure 17: 2d illustration of a discrete isosurface 41
Figure 18: The 15 distinct cube triangle configurations 42
Figure 19: Marching cubes shared corners and edges 43
Figure 20: Simple contribution of particle density 44
Figure 21: Stamp creation 44
Figure 22: Contribution of particle density with stamp 44
Figure 23: Resulting marching cubes surface visualisation (voxel-size 0.9^3) 45
Figure 24: Marching cubes visualisation (voxel-size 0.1^3) 45
Figure 25: Direct volume rendering of a CT scan with color coding of different densities 47
Figure 26: Isosurface-raycast renderings 47
Figure 27: Ray construction 47
Figure 28: Sampling along the ray 47
Figure 29: Hit refinement 48
Figure 30: Ray exit texture 48
Figure 31: Reason of per pixel step-size 49
Figure 32: Real water in a non-imaginary glass 50
Figure 33: Law of reflection 51
Figure 34: Vector form of reflection 51
Figure 35: Snell's law 52
Figure 36: Vector form of refraction 53
Figure 37: Fresnel reflection coefficients 55
Figure 38: Fresnel term approximation quality 56

Figure 39: Light hits surface from air direction 57
Figure 40: Light hits surface from water direction 57
Figure 41: Water surface - optical effects 57
Figure 42: Calculated rays 58
Figure 43: Ray-tracing starting point 60
Figure 44: Gradient calculation 61
Figure 45: Isosurface ray-tracing screenshots 62
Figure 46: Screen filling at 1280 resolution 63
Figure 47: 1/9 screen-coverage 63
Figure 48: Parallax occlusion mapping principle 65
Figure 49: Particle slicing 66
Figure 50: Slice index construction 67
Figure 51: Generated density texture viewed in PIX 68
Figure 52: Slices of the volumetric density texture 68

Derivation of the gradient and Laplacian of the smoothing kernels

Used calculation rules:

Gradient (of a scalar valued function):

$$grad\ \varphi(\mathbf{r}) = \nabla\varphi(\mathbf{r}) = \begin{pmatrix} \partial\varphi/\partial x \\ \partial\varphi/\partial y \\ \partial\varphi/\partial z \end{pmatrix}$$

Laplacian (of a scalar valued function; sometimes also written ∇^2 or Δ):

$$\nabla \cdot \nabla\varphi = div(grad\ \varphi) = \nabla \cdot (\nabla\varphi)$$

Chain rule:

$$\big(f(x) \circ g(x)\big)' = f'\big(g(x)\big) \cdot g'(x)$$

Product rule:

$$\big(f(x) \cdot g(x)\big)' = f'(x) \cdot g(x) + f(x) \cdot g'(x)$$

Gradient and Laplacian of $W_{poly\,6}$:

$$W_{poly\,6}(\mathbf{r}, h) = \frac{315}{64\pi h^9}(h^2 - r^2)^3 = \frac{315}{64\pi h^9}\big(h^2 - {r_x}^2 - {r_y}^2 - {r_z}^2\big)^3$$

with: $r = |\mathbf{r}| = \sqrt{r_x^2 + r_y^2 + r_z^2}$, $h = const$

$$\nabla W_{poly\,6}(\mathbf{r}, h) = \left(\frac{-3 \cdot 315 \cdot 2r_x}{64\pi h^9}(h^2 - r^2)^2, \dots, \dots\right)^T = -\mathbf{r}\frac{945}{32\pi h^9}(h^2 - r^2)^2$$

$$\begin{aligned} \nabla \cdot \nabla W_{poly\,6}(\mathbf{r}, h) = \nabla \cdot \left(\nabla W_{poly\,6}(\vec{r}, h)\right) &= \frac{\partial \frac{-945}{32\pi h^9}(h^2 - r^2)^2 r_x}{\partial x} + \frac{\dots}{\partial y} + \frac{\dots}{\partial z} \\ &= \frac{-945}{32\pi h^9}(h^2 - r^2)^2 + \frac{945}{32\pi h^9}(h^2 - r^2)2r_x 2r_x + \frac{\dots}{\partial y} + \frac{\dots}{\partial z} \\ &= r^2\frac{945}{8\pi h^9}(h^2 - r^2) - \frac{2835}{32\pi h^9}(h^2 - r^2)^2 = \frac{945}{8\pi h^9}(h^2 - r^2)\left(r^2 - \frac{3}{4}(h^2 - r^2)\right) \end{aligned}$$

Gradient of W_{spiky} :

$$W_{spiky}(\mathbf{r},h) = \frac{15}{\pi h^6}(h-r)^3 = \frac{15}{\pi h^6}\left(h - \sqrt{{r_x}^2 + r_y^2 + r_z^2}\right)^3$$

$$\nabla W_{spiky}(\mathbf{r},h) = \left(-\frac{15\cdot 3\cdot 2r_x}{2\pi h^6 r}(h-r)^2, \dots, \dots\right)^T = -\mathbf{r}\frac{45}{\pi h^6 r}(h-r)^2$$

Gradient and Laplacian of $W_{viscosity}$:

$$W_{viscosity}(\mathbf{r},h) = \frac{15}{2\pi h^3}\left(-\frac{r^3}{2h^3} + \frac{r^2}{h^2} + \frac{h}{2r} - 1\right)$$

$$\begin{aligned}\nabla W_{viscosity}(\mathbf{r},h) &= \left(\frac{15}{2\pi h^3}\left(-\frac{3r^2 2r_x}{2\cdot 2h^3 r} + \frac{2r2r_x}{2h^2 r} - \frac{h2r_x}{2\cdot 2r^2 r}\right), \dots, \dots\right)^T \\ &= \mathbf{r}\frac{15}{2\pi h^3}\left(-\frac{3r}{2h^3} + \frac{2}{h^2} - \frac{h}{2r^3}\right)\end{aligned}$$

$$\begin{aligned}\nabla\cdot\nabla \mathrm{W}_{\text{viscosity}}(\mathbf{r},\mathrm{h}) &= \nabla\cdot\left(\nabla \mathrm{W}_{\text{viscosity}}(\mathbf{r},\mathrm{h})\right) \\ &= \frac{\partial}{\partial x} r_x \frac{15}{2\pi h^3}\left(-\frac{3r}{2h^3} + \frac{2}{h^2} - \frac{h}{2r^3}\right) + \frac{\dots}{\partial y} + \frac{\dots}{\partial z} \\ &= \frac{15}{2\pi h^3}\left(-\frac{3r}{2h^3} + \frac{2}{h^2} - \frac{h}{2r^3}\right) + r_x\frac{15}{2\pi h^3}\left(-\frac{3r_x}{2h^3 r} + \frac{3hr_x}{2r^5}\right) + \cdots + \cdots \\ &= \frac{45}{2\pi h^3}\left(-\frac{3r}{2h^3} + \frac{2}{h^2} - \frac{h}{2r^3}\right) + \frac{15}{2\pi h^3}\left(-\frac{3r}{2h^3} + \frac{3h}{2r^3}\right) \\ &= \frac{15}{2\pi h^3}\left(-\frac{9r}{2h^3} + \frac{6}{h^2} - \frac{3h}{2r^3} - \frac{3r}{2h^3} + \frac{3h}{2r^3}\right) = \frac{15}{2\pi h^3}\left(-\frac{12r}{2h^3} + \frac{6}{h^2}\right) \\ &= \frac{90}{2\pi h^5}\left(-\frac{2r}{2h} + \frac{1}{1}\right) = \frac{45}{\pi h^5}\left(1 - \frac{r}{h}\right)\end{aligned}$$