Realtime particle based fluid simulation

Stefan Auer

Contents

1) Motivation

- Fluids are ubiquitous parts of our environment

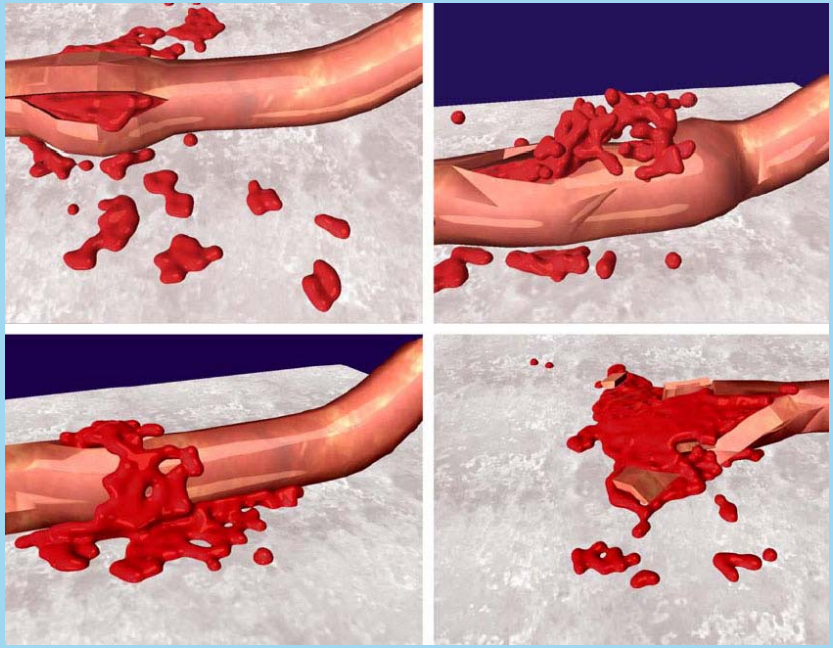
(fluids include liquids and gases, but our focus is on liquids)

1) Motivation

- realtime computer graphics tries to simulate the world believable
- dynamic fluids are rare in interactive applications
- CFD is a big topic, but mainly offline

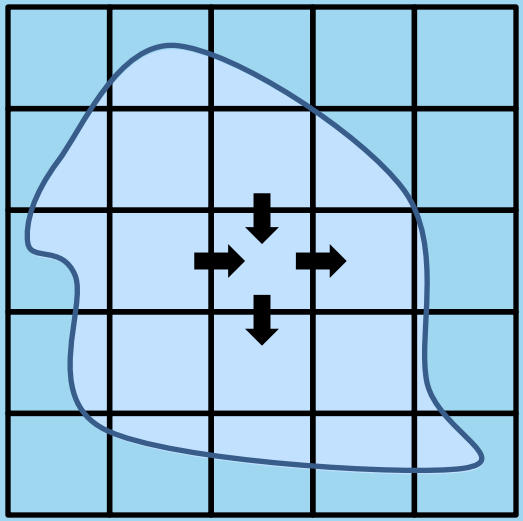
1) Motivation

- demand for fluids in interactive applications
- “Particle-Based Fluid Simulation for Interactive Applications” (2003)



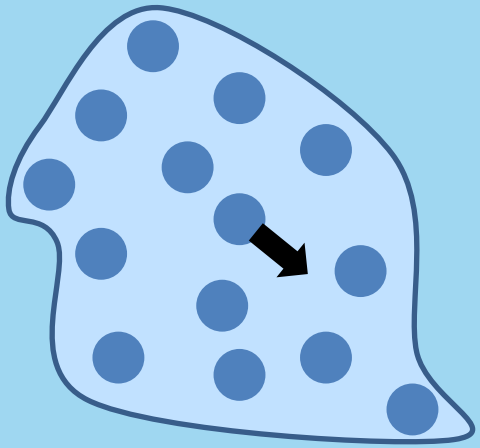
2) How to simulate fluids

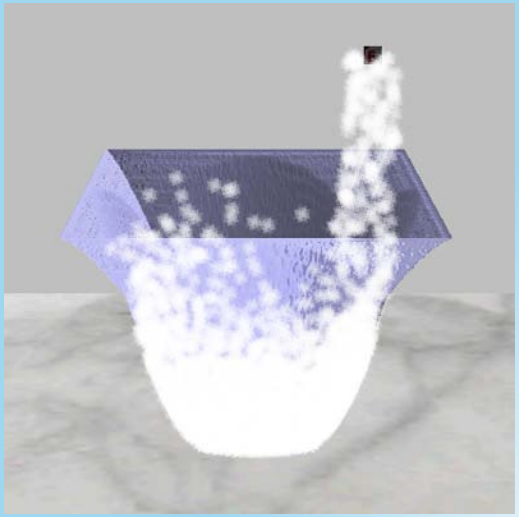
- basis: Navier Stokes equations
- mostly result in non-linear PDEs
- -> numerical methods
- common: discretize (grid) + finite differences (or FEM)



2) How to simulate fluids

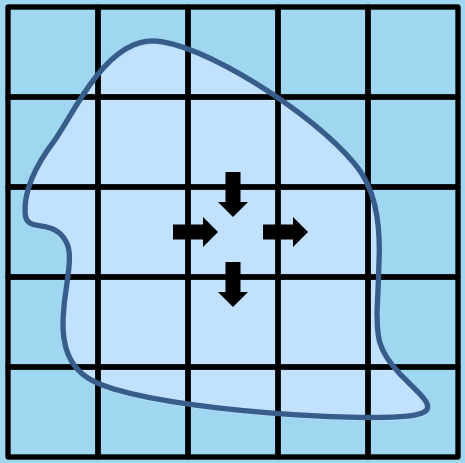
- alternative to (Eulerian) grid based approach: particle based methods (Lagrangian)
- represent fluid as set of particles
- particle dynamics simulate fluid flow

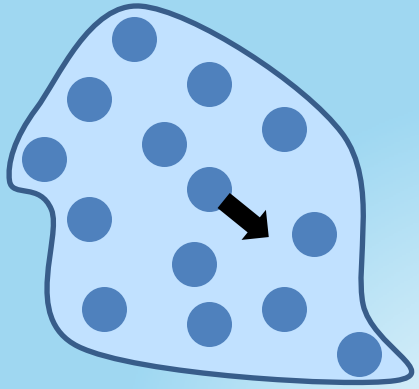




2) How to simulate fluids

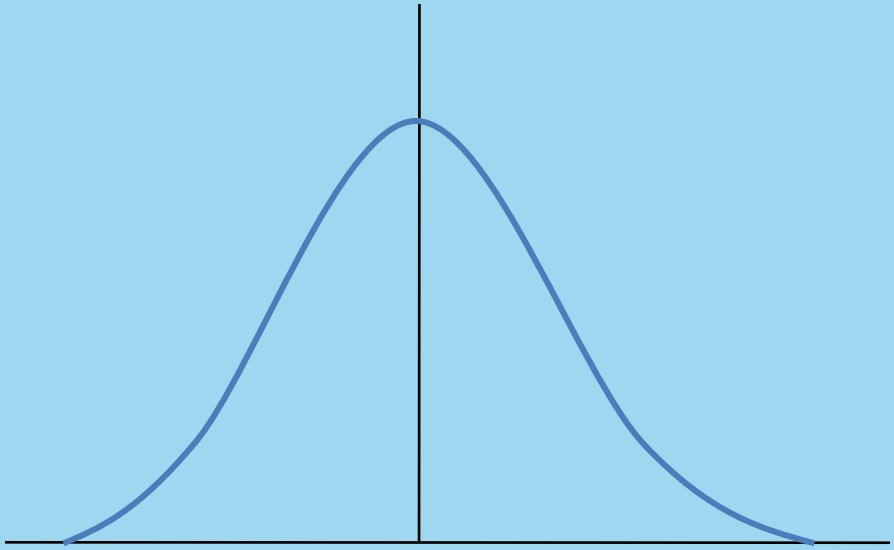
- advantages:
  - equations and computations become simpler
  - no numerical diffusions
  - partly easier surface construction
  - fluid can spread freely in space

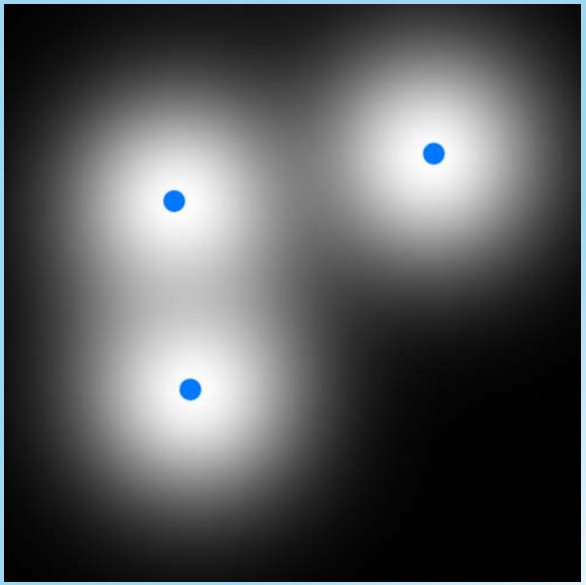




2) How to simulate fluids

- chosen technique: smoothed particle hydrodynamics
- idea: particles distribute fluid properties in their neighbourhood
- done via smoothing functions





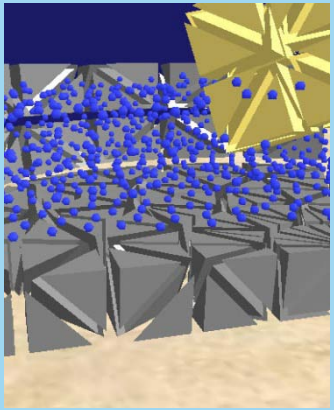
3) Related work

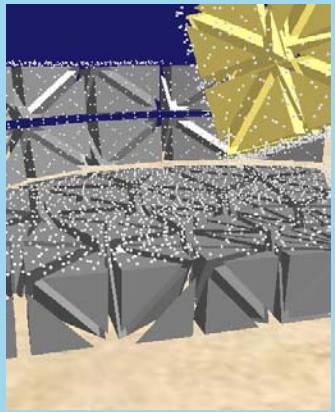
- SPH in general:
  - Developed by Gingold and Monaghan 1977
  - and independently by Lucy (1977 too)
  - first usage: astronomy (LARGE scale gas-dynamics)
  - later used for liquids too (beach waves, sloshing tanks, ...)

- “SPH light” for interactive applications:
  - made public 2003 by Müller, Charypar, Gross:
    “Particle based fluid simulation for interactive appliactions”

3) Related work

- following papers on realtime SPH:
  - Müller et al. 2004: “Interaction of fluids with deformable solids”





  - Kolb, Cuntz 2005: “Dynamic Particle Coupling for GPU-based Fluid Simulation”
  - Heinecke 2007: “Physikalische Rauchsimulation auf Partikelbasis in Echtzeit mit der PhysX-Engine“

3) Related work

- good visualisations are important (and complex) too:
  - Müller suggests 2003 direct point splatting of the particles or marching cubes
  - “Acceleration Techniques for GPU-based Volume Rendering” (2003 by Krüger, Westermann) among other things: isosurface ray-casting
  - Co et al. 2003: iso-splatting
  - Kipfer, Westermann 2003: “carpet based” river visualisation
  - Müller et al. 2007; “screen space meshes”

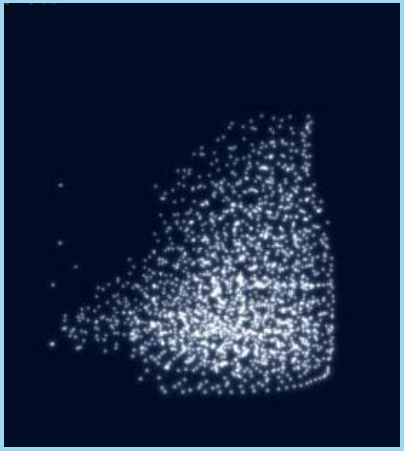
4) Used techniques

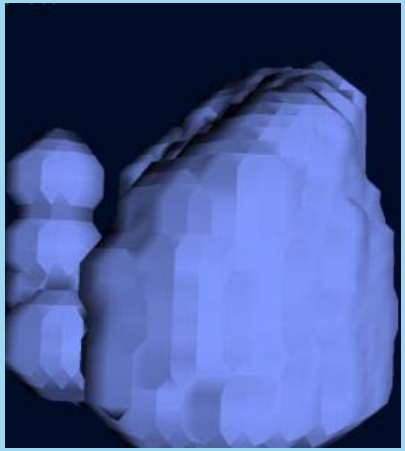
- goal: simulation of water like liquids
  - believable movement
  - believable in optical appearance

- implemented fluid simulation in short:
  - same “lightweight” SPH model as Müller 2003
  - neighbour search via dynamic grid
  - particles interact pair-wise
  - multi-core optimised

4) Used techniques

- implemented visualisations in short:
  - sprite-based direct particle rendering
  - marching cubes
  - isosurface ray-tracing
    - based on isosurface ray-casting
    - simulates multiple reflections and refractions





Contents

1) Basics of fluid mechanics

- treat fluid as continuum
- every property has a defined value at each point in space
- mathematically: vector- or scalar fields
- fluid properties:
  - mass
    - "how much matter there is"
  - (mass-) density
    - "mass per volume"

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

1) Basics of fluid mechanics

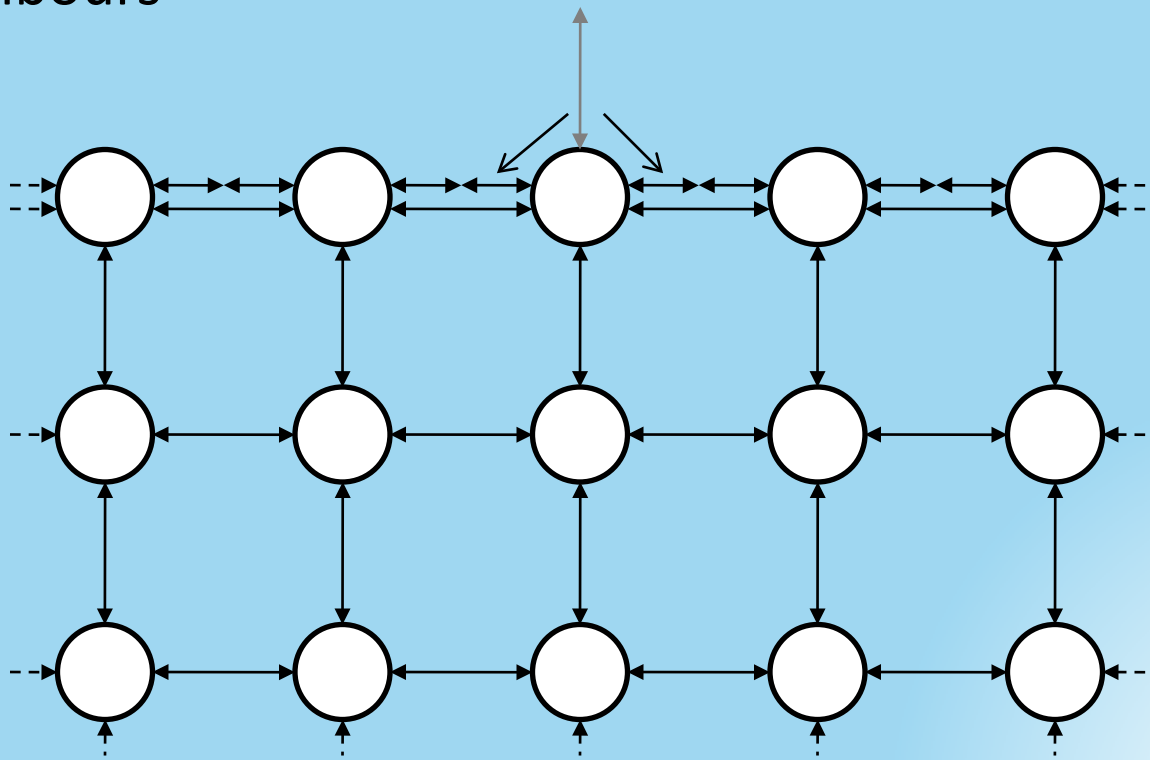
- fluid properties continued:
  - pressure
    - scalar quantity
    - force in normal direction on a surface $p \equiv \lim_{\Delta A \to 0} \frac{\Delta F_n}{\Delta A}$
  - velocity
    - vector quantity
    - measures how fast fluid passes a fixed point
    - influences most other quantities (advection)
    - relevant for viscosity forces

1) Basics of fluid mechanics

- fluid properties continued:
  - viscosity
    - compensates differences in flow velocity
    - comparable to friction
    - viscosity constant describes momentum transfer between adjacent regions of different flow speed
    - viscosity force: tangential force on a surface
  - surface tension
    - property of the surface
    - describes size of forces that try to minimize the surface

1) Basics of fluid mechanics

- cause of surface tension (simple explanation)
  - surface molecules share cohesive forces with less neighbours



1) Basics of fluid mechanics

- now following: from Newtonian- to fluid mechanics
- “derivation” of the Navier Stokes equation
  - no mathematical derivation
  - only one particular equation variant discussed
  - goal: understand basic concepts, meaning and components of the equation
- basic statement:
  - “The Navier Stokes equation is the formulation of Newton’s second law of motion for fluids.”

1) Basics of fluid mechanics

- Newton's second law of motion:

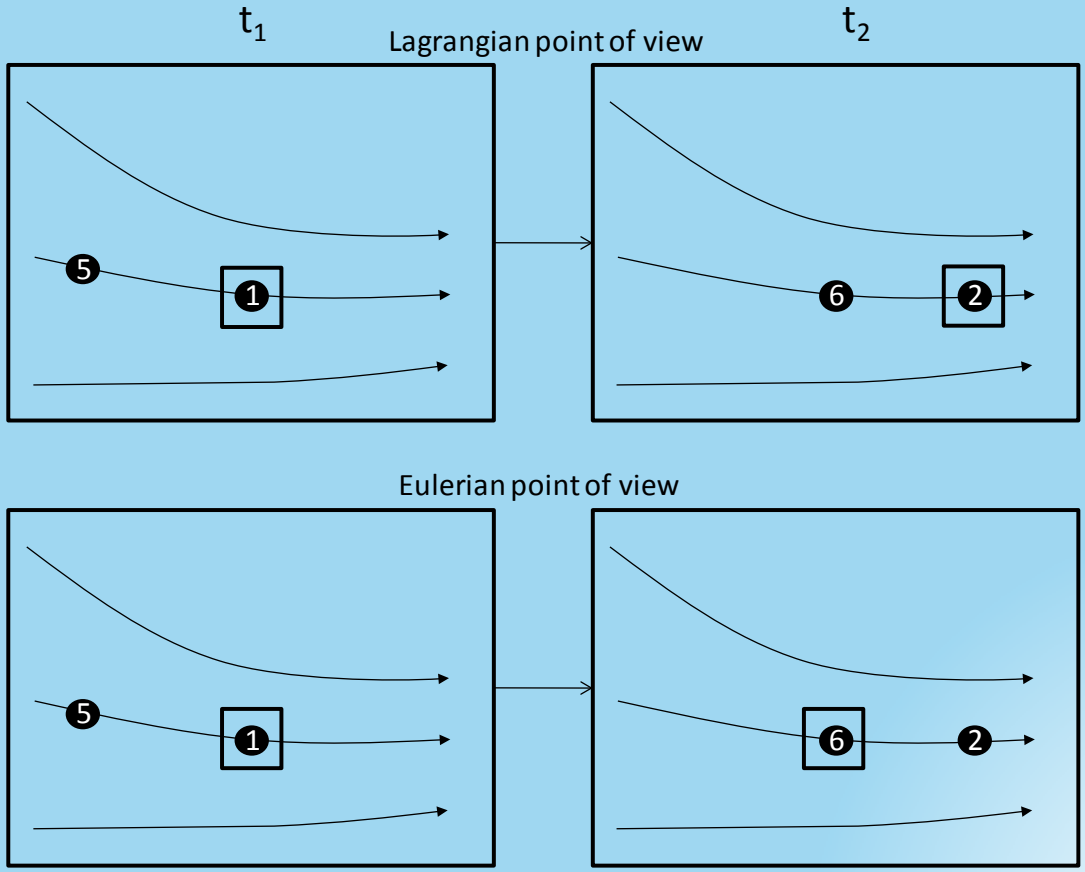
$$\mathbf{F} = m\mathbf{a}$$

- in Newtonian mechanics usually interpreted from the Lagrangian point of view:
  - observer follows a moving object
  - with fluids: watch always the same "amount of fluid"

- alternative: Eulerian point of view
  - observe always the same point in space
  - with fluids: watched "amount of fluid" changes when fluid moves

1) Basics of fluid mechanics

- Eulerian vs. Lagrangian point of view:



1) Basics of fluid mechanics

- acceleration in Lagrangian point of view:
  - (simple) time derivative of velocity
- acceleration in Eulerian point of view:
  - convective derivative of velocity
  - convective derivative:

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

- Newton's second law becomes:

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

1) Basics of fluid mechanics

- a certain volume element is watched
- mass in a volume is given by density ->

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

- split forces in fluid forces and external forces:

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

1) Basics of fluid mechanics

- external forces mainly gravity
- typically specified as acceleration (F=mg)
- mass still depends on density ->

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

1) Basics of fluid mechanics

- for the next step assume:
  - fluid is Newtonian
    - viscous stress proportional to velocity gradient
    - in common words: viscosity is a constant that doesn't change with different shear rates
  - fluid satisfies the incompressible flow condition
    - no sources or sinks in the velocity field $\nabla \cdot \mathbf{v} = 0$
    - counter example: air that heats up
    - also flows of compressible fluid can satisfy the condition

1) Basics of fluid mechanics

- if all that's the case,
  fluid forces can simply be split up:

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

- pressure force
  - depends only on pressure differences
  - points from high- to low pressure areas
  - -> the negative pressure gradient is what we want

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

1) Basics of fluid mechanics

- viscosity force
  - because of the incompressible flow condition it becomes the simple term:

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

  - just believe it or see “Viscous fluid flow” (1999 by Papanastasiou et al.) for mathematical derivation
  - Laplacian operator measures how far a quantity is from the average around it
  - -> the force smoothes the velocity differences over time
  - this is what viscosity should do

1) Basics of fluid mechanics

- inserting viscosity force we arrive at Newton's second law for fluids, the Navier Stokes equation:

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

(more exactly: Navier Stokes momentum equation for incompressible, Newtonian fluids)

  - which hopefully now is somehow understandable

Contents

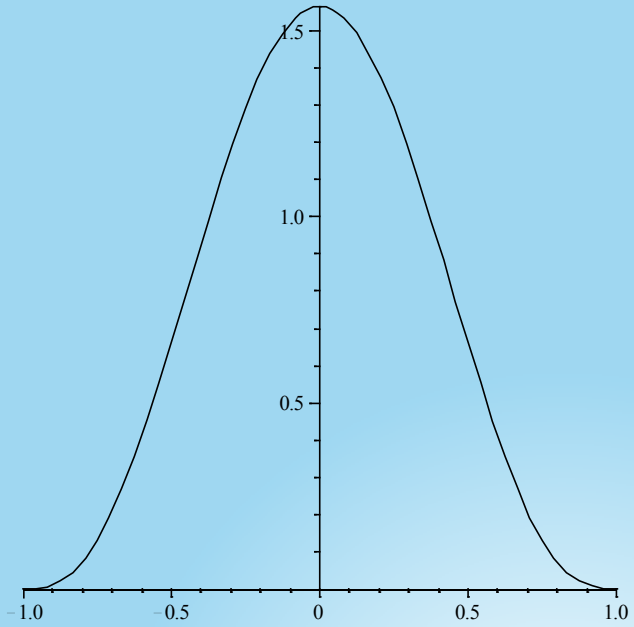
2) Basics of smoothed particle hydrodynamics

- SPH was developed for astrophysical gas dynamics problems
- as in many other numerical solutions, values are interpolated from a discrete set of points
- SPH derives from the integral interpolation:

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

  - W: smoothing function (kernel)

- a word on the smoothing kernel:
  - "spreads a quantity in its surroundings"
  - is even (radial symmetric)
  - is normalised: $\int W(\mathbf{r})d\mathbf{r} = 1$
  - tends to become the delta function for h tending to zero (for h=0 the integral would reproduce A exactly)
  - for us: h = radius of support



- SPH is a Lagrangian method
  - moveable mass elements (particles) that carry properties
  - particle positions are the discrete set of interpolation points
  - interpolation integral becomes interpolation sum:

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

  - mass density coefficient appears because each particle represents a volume: $V_i = \frac{m_j}{\rho_j}$

- interesting example:
  - SPH interpolation applied to density

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

  - -> for SPH the density is simply the smoothed mass of the particles
- in practical use not all particles must be evaluated
  - consider only particles that are within the radius of support (=h for us)

- advantage of SPH: simple derivatives
  - derivatives are very common in fluid equations
  - with SPH only the smoothing function has to be differentiated

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

  - the gradient therefore becomes:

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

- according to Müller this can be applied for the Laplacian too:

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

- this formulations are not “physically bullet-proof”
  - resulting equations can violate physical laws
  - -> correct them were necessary
  - also different definitions for SPH gradient and Laplacian exist in the literature

Contents

3) Particle based, mathematical model of fluid motion

- combine Navier Stokes equation with basic SPH rules
  - discrete number of particles
  - particles carry only:
    - mass (constant and equal for all particles)
    - position
    - Velocity
  - all other quantities derived using SPH and basic physics equations

3) Particle based, mathematical model of fluid motion

- Eulerian models need equations for:
  - conservation of momentum (Navier Stokes)
  - conservation of mass (continuity equation) $\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0$
  - conservation of energy (only sometimes)

- SPH needs only momentum equation
  - particle count and -masses constant -> mass constant

3) Particle based, mathematical model of fluid motion

- Eulerian models need convective derivative
  - because of advection
- SPH can use simple time derivative
  - advection implicit, because properties are carried with the particles
  - -> replace convective derivative in N. S. equation with time derivative
  - leads to momentum equation for a single particle:

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

- particle acceleration therefore becomes simply:

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

- remember the SPH example for the density?

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

- all what's missing is pressure- and viscosity force

- pressure force
  - SPH rules would lead to the following pressure term:

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

  - unfortunately this force is not symmetric (think of two particles with different pressure)
  - fortunately a symmetric pendant is easy to find
  - use the arithmetic mean pressure:

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

3) Particle based, mathematical model of fluid motion

- pressure force continued
  - now we need the scalar pressure at the particle's position
  - Müller suggests the ideal gas state equation:

$$p = k(\rho - \rho_0)$$

  - this introduces some drawbacks (compressibility i.e.)
  - we accept them for a simple simulation model

- viscosity force
  - the SPH rule delivers:

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

  - again asymmetric ☹, but again easy to symmetrise ☺
  - viscosity depends only on velocity differences ->

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

  - the viscosity force in our model hence accelerates each particle to meet the relative speed of its environment

- now we would be finished as we have equations for all what's covered by the N. S. equation... ☺

- ... but because we deal with much free surface, we will consider surface tension too ☹

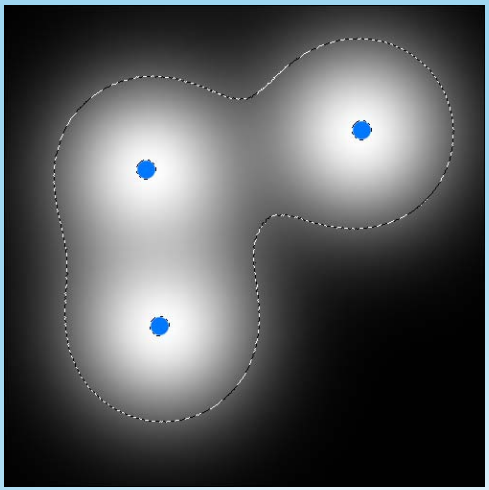
3) Particle based, mathematical model of fluid motion

- surface tension force
  - applies only to particles near the surface
  - pushes particles towards the inner of the fluid (-> negative surface normal)
  - higher surface curvature -> higher surface tension force

- the colour-field can deliver us all that

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

  - its gradient delivers the surface normal and is a measure for the surface-closeness
  - its Laplacian measures the surface curvature



- in short the surface-tension term looks like this:

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

  - $|\nabla c_S|$ is near to zero for inner particles
  - -> surface tension only evaluated where it exceeds a threshold

- now our fluid model is complete ☺ ☺

4) Smoothing kernels

why we need to use the smoothing kernel?

- three different smoothing kernels are used:
  - poly6

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

    - advantage: r appears only squared (no square roots)
    - used everywhere except for pressure and viscosity terms
  - spiky

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

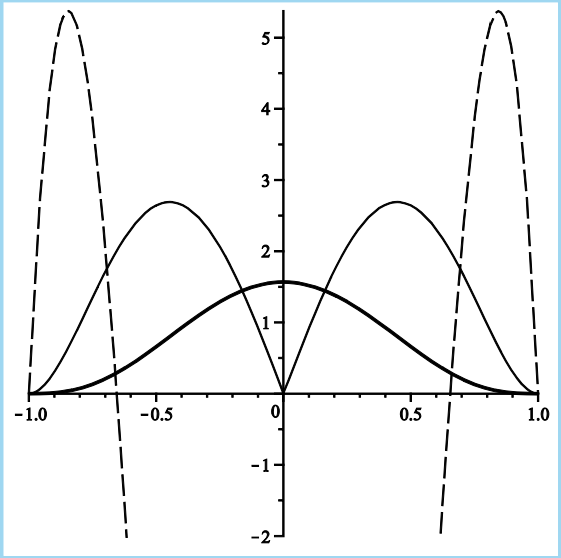
    - gradient doesn't vanish near the centre
    - used for pressure
  - viscosity

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

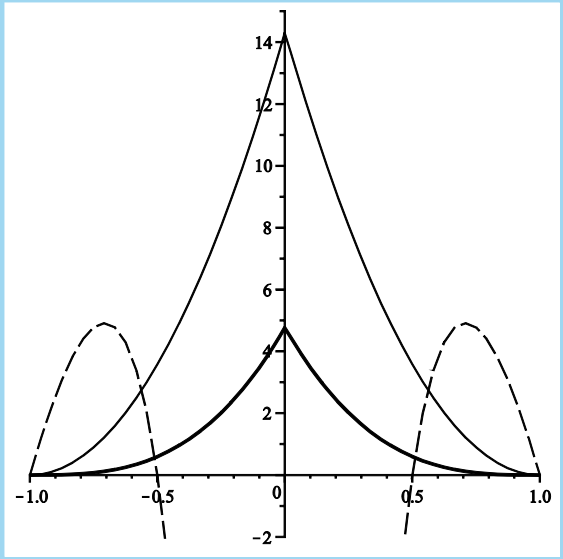
    - Laplacian stays positive everywhere
    - used for viscosity

4) Smoothing kernels

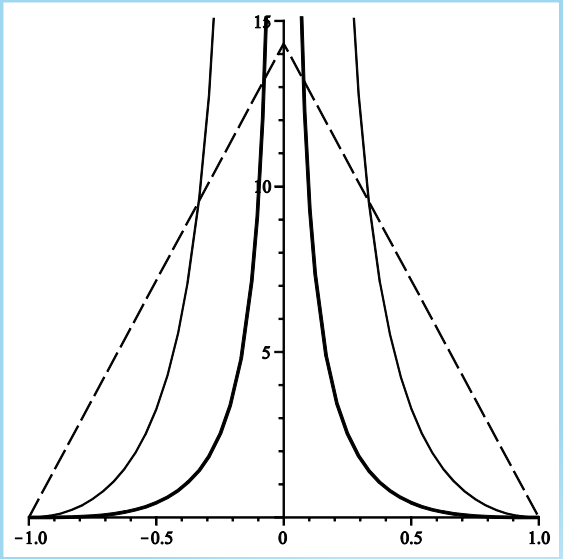
- used smoothing kernels:



poly6



spiky



viskosity

Contents

5) Basic simulation algorithm

- basically the simulation consists of 3 steps

1. compute the density for each particle
   - accumulate the contributions of all neighbours
   - apply density equation

2. compute forces and colour-field for each particle
   - accumulate the contributions of all neighbours
   - apply force and colour-field equations

3. move each particle
   - linearly process one particle after another
   - calculate total force, acceleration
   - leads to new velocity and position

5) Basic simulation algorithm

- step 1

```
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
      end-if
   end-foreach
end-foreach
```

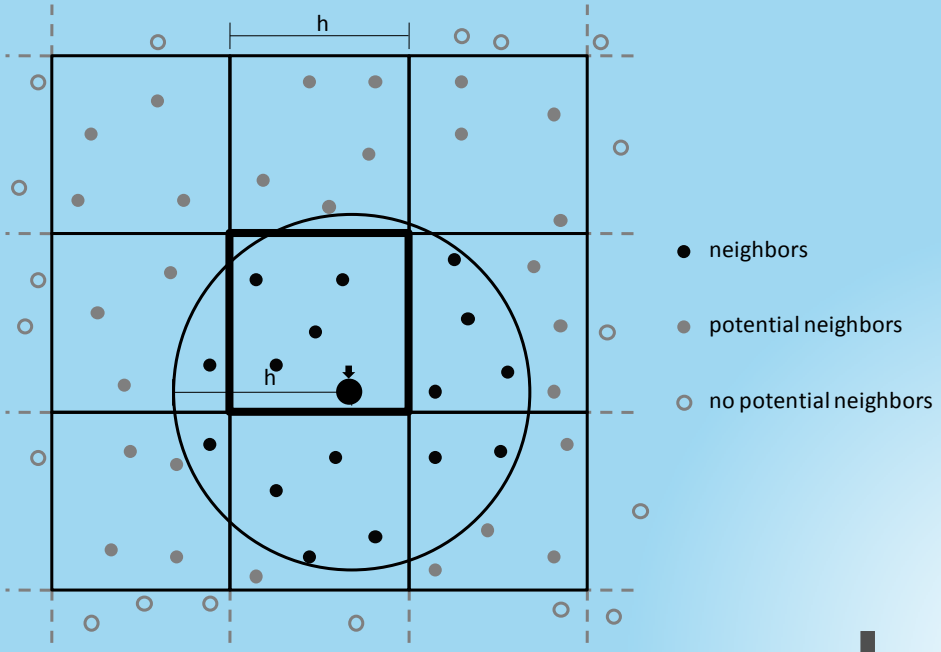
- step 2 uses the same evaluation scheme
  - applies the force and colour-field equations
- step 3 just processes all particles linearly
  - combines the results and calculates new positions and velocities

6) Implementation

- implementation consists of the 3 steps
  plus update of the acceleration structures
  - transferred between simulation steps:
    - only particle position and velocity
  - all other per-particle data stored in separate arrays
  - density calculation is the first step that performs the summation
    - must implement a neighbour-search
      to avoid quadratic computation complexity

6) Implementation

- neighbour-search
  - crucial for the overall performance
  - different solutions in the literature
  - used data-structure:
    grid with cell-size equal to radius of support

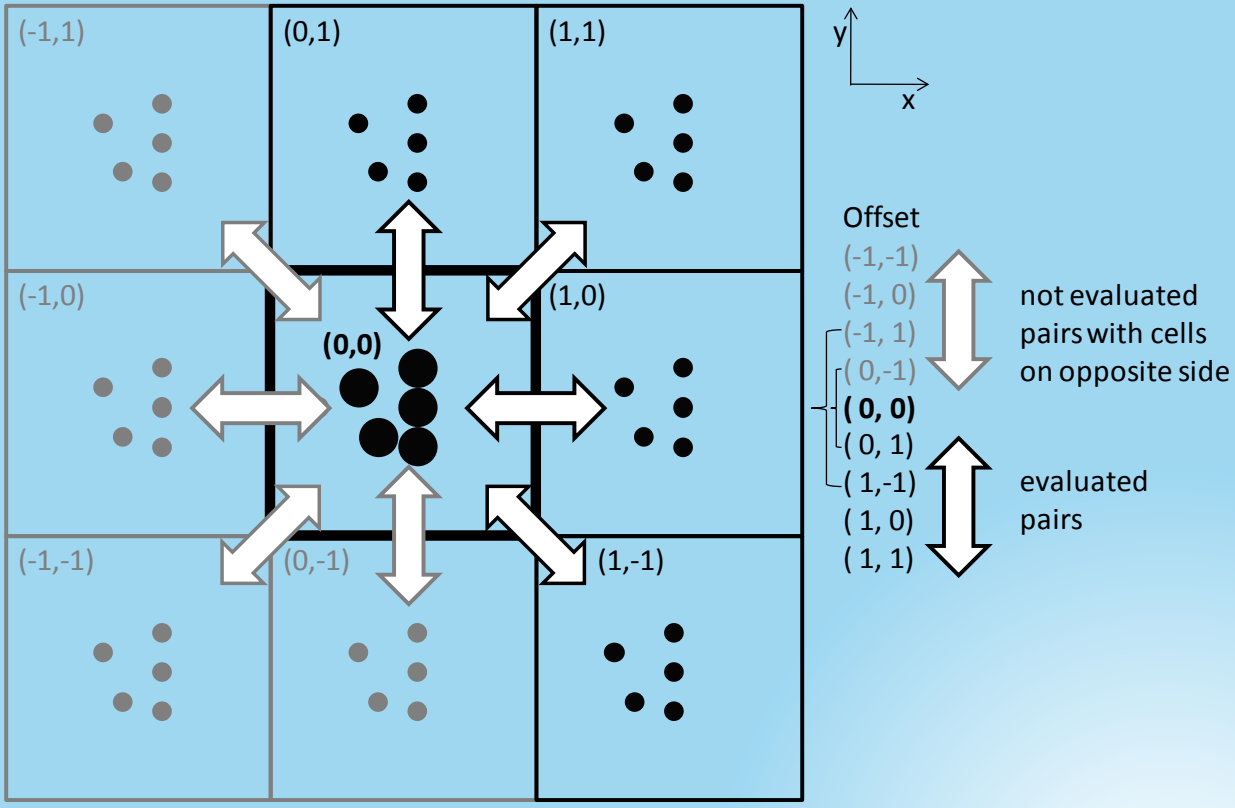


6) Implementation

- further optimisation: exploitation of symmetry

neighbour offsets in 3D case:

----↓----	----↑----
(-1,-1,-1)	(1, 1, 1)
(-1,-1, 0)	(1, 1, 0)
(-1,-1, 1)	(1, 1,-1)
(-1, 0,-1)	(1, 0, 1)
(-1, 0, 0)	(1, 0, 0)
(-1, 0, 1)	(1, 0,-1)
(-1, 1,-1)	(1,-1, 1)
(-1, 1, 0)	(1,-1, 0)
(-1, 1, 1)	(1,-1,-1)
(0,-1,-1)	(0, 1, 1)
(0,-1, 0)	(0, 1, 0)
(0,-1, 1)	(0, 1,-1)
(0, 0,-1)	(0, 0, 1)
(0, 0, 0)	**→(0, 0, 0)**



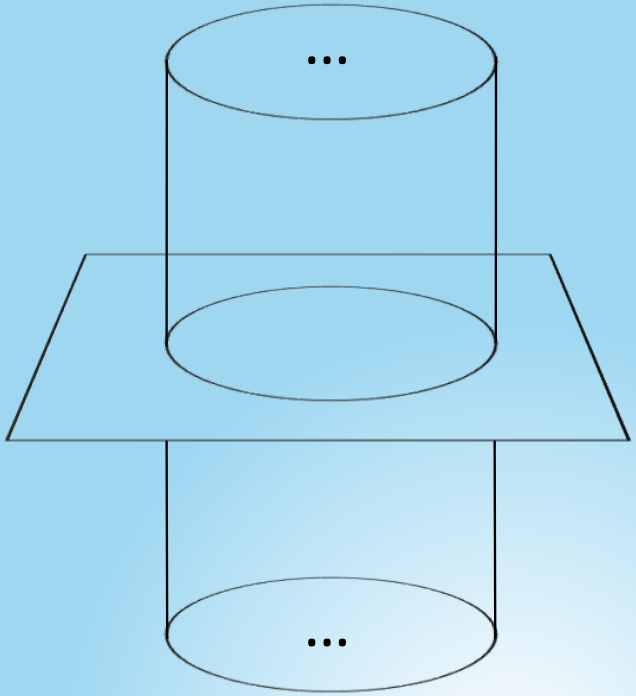
6) Implementation

- all found particle pairs are stored temporarily
  - -> step 2 don't has to perform the neighbour search again
- calculation of movement rather trivial
  - Newton's law leads from total force to acceleration (a=F/m)
  - new velocity = old velocity + elapsed time * acceleration
  - new position = old position + elapsed time * ½ * (old velocity + new velocity)

Contents

7) Environment and user interaction

- environment and user interaction should be simulated
- -> the fluid is contained in a (imaginary) glass, which can be moved with the mouse
  - well known scenario
  - allows long time interaction
  - one way interaction (fluid doesn't influence glass)
  - glass simple to model
  - simple collision code



7) Environment and user interaction

- “glass -> particle” interaction handled in the same was as “particle <-> particle” interaction
- -> extra steps for glass related density calculation and glass related forces calculation added
- -> simulation now consists of 6 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

Contents

8) Multithreading optimisation

- consumer PCs equipped with multi-core CPUs
  - multi-threading necessary to utilise their full power
  - step-based simulation -> very short reoccurring tasks
    - no task-parallelism possible -> utilize data-parallelism
  - -> thread manager (pool) to avoid expensive thread creations and control job handover
  - 5 of the 6 steps where made multi-threaded

8) Multithreading optimisation

- step 1: “particle <-> particle” density calculation
  - neighbour search multi-threaded
  - grid cells distributed among threads
  - problem: particle pairs between different cells
    - two threads could simultaneous try to add their contribution to the same particle
    - add operation not atomic -> summand could get swallowed
    - -> synchronisation (i.e. atomic ops) ?
    - NO: would eliminate performance gain
    - instead: only make density volatile and accept extremely rare collisions (no big failure)

8) Multithreading optimisation

- step 3: force and colour-field calculation
  - each thread gets its own pair-list from step 1
  - arrays volatile again
- steps 2, 4, 5
  - each thread processes the same count of particles
  - no collisions possible
- step 6: update of the neighbour-search-grid
  - not parallelisable in a reasonable way

Contents

9) Results

- performance results:
  - test machine:
    - CPU: Intel Core 2 Quad Q6600 @ 3.2 GHz
    - RAM: 2 GB
  - simulation of all possible forces
  - sprite visualisation active (no measureable overhead)

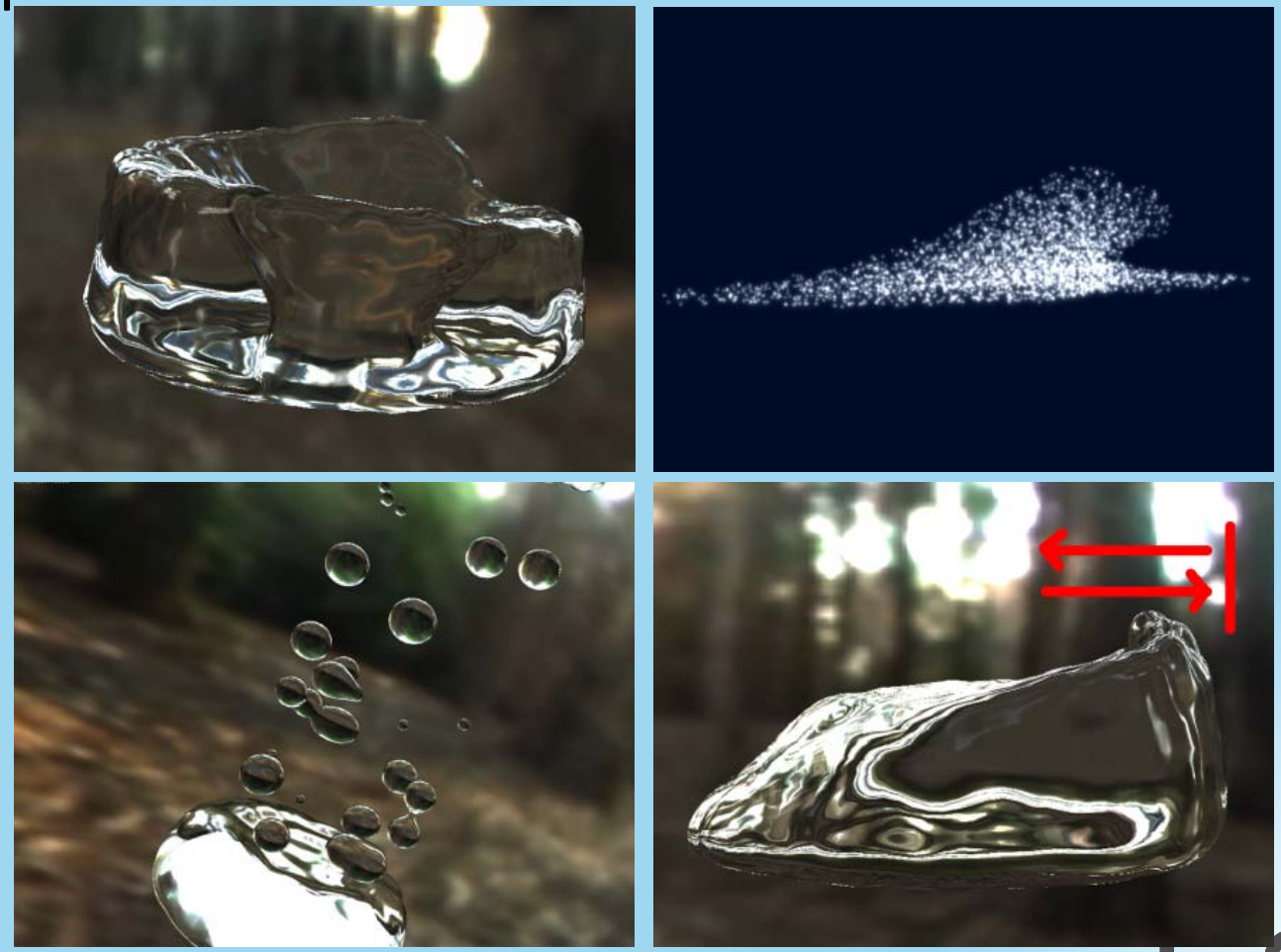
  - 1728 (12^3) particles: 330 FPS
  - 10648 (22^3) particles: 53 FPS
  - 27000 (30^3) particles: 14 FPS

9) Results

- plausibility of the fluids behaviour
  - harder to measure
  - simulates all major effects that are observable at real fluids in real glasses
    - vortex formation
    - wave breaking
    - wave reflection
    - drop formation
    - drops draining down the side of the glass

9) Results

- examples



9) Results

demonstration?

10) Further work

- incompressibility
  - some ideas in the literature
  - but apparently none used in a realtime simulation so far
- surface tension
  - colour-field Laplacian problematic
  - model based on cohesive forces proposed by Becker and Teschner
  - similar model already implemented

10) Further work

- environment interaction
  - support a richer environment
  - simulate obstacles with virtual particles
    - would also make two-way interaction very easy (-> i.e. interaction with rigid body simulation)
    - mapping from conventional geometric objects to particles required
    - high total number of particles
  - alternative: interact with simplified geometry
    - perhaps more efficient

10) Further work

- neighbour search
  - current approach not greatly spatial flexible
    - destroys advantage of particle approach somewhat
  - use hashing algorithms to map unlimited space partitions to limited amount of slots
  - consider different space partitioning algorithms
- target hardware
  - implement simulation on the GPU
  - wait for CPU manufactures to provide SPMD optimised hardware
    - Intel: Larrabee (x86 GPU)
    - AMD: Fusion (later products)

Contents

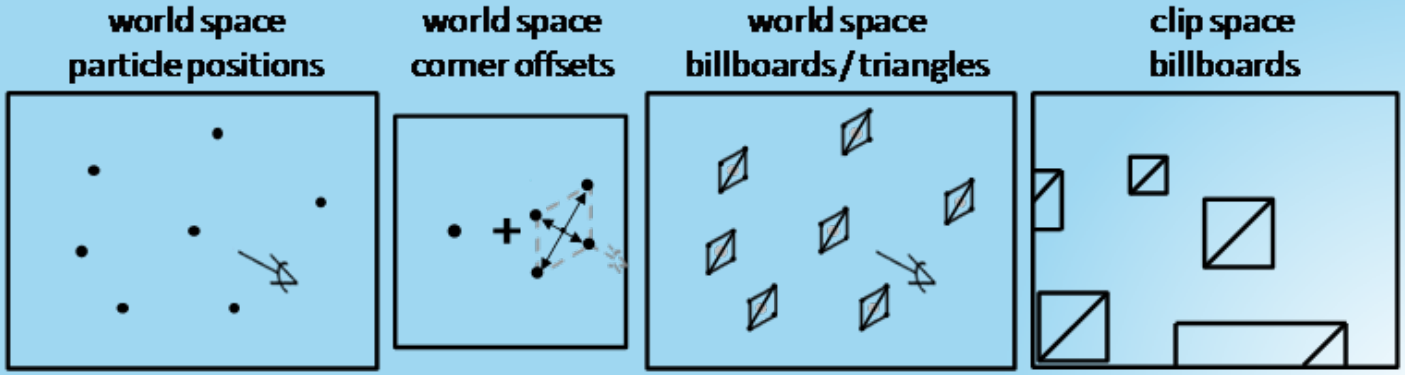
1) Target graphics APIs

- in short:
  - sprite rendering and marching cubes implemented for D3D 9 (entirely fixed function) and D3D 10
  - isosurface ray-casting D3D10 exclusive
    - makes use of geometry shaders
    - makes use of way more flexible render target choice
    - makes heavy use of branching and flow control
    - is portable, but would become far less elegant with D3D9
  - introduction to D3D10 in the thesis (but you can just as well read the material from Microsoft)

Contents

3) Direct particle rendering

- useful for:
  - analysis of the particle behaviour
  - simulation debugging
  - visualisation of fluid properties
    (size-, transparency- or colour-coding)
- D3D9 implementation
  - fixed function point sprite functionality used
- D3D10 implementation
  - geometry shader used



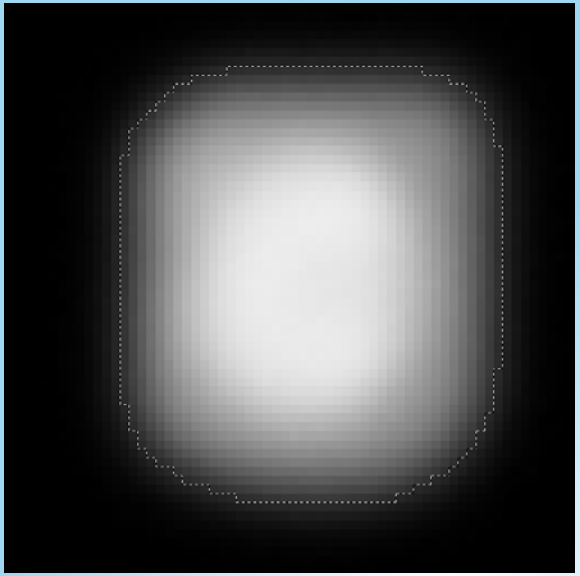
Contents

3) Isosurface rendering with marching cubes

- clear water is fairly transparent
  - the inner of the water don't has to be visualised
- but the water surface exhibits some interesting optical phenomena
  - -> render only the water surface
- how to describe the surface in a fluid simulation?
- common approach:
  - take a discrete density grid as basis
  - find isosurfaces

3) Isosurface rendering with marching cubes

- what is a isosurface?
  - ἴσος / isos is Greek and means “equal”
  - an isosurface is a region of a volumetric scalar-field where all values are equal
  - when the field fulfils some conditions this region has the form of a 2D surface (orientable 2 manifold without boundaries)
  - when domain and range are discrete define isosurface as area between regions lower and regions equal or higher than the isovalue

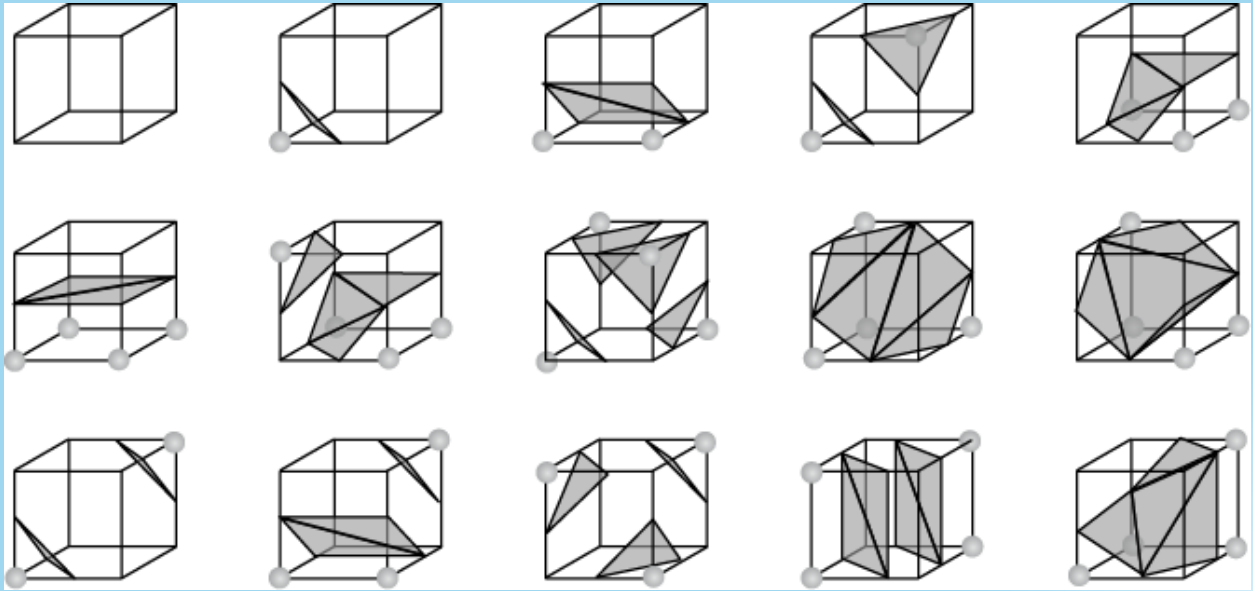


3) Isosurface rendering with marching cubes

- marching cubes
  - common technique to construct a renderable triangle mesh of the isosurface
  - piecewise process the grid
  - 8 densities values form a graph in form of a cube
  - find intersections of the edges with the isosurface
  - for each intersection create a vertex (linear interpolation)

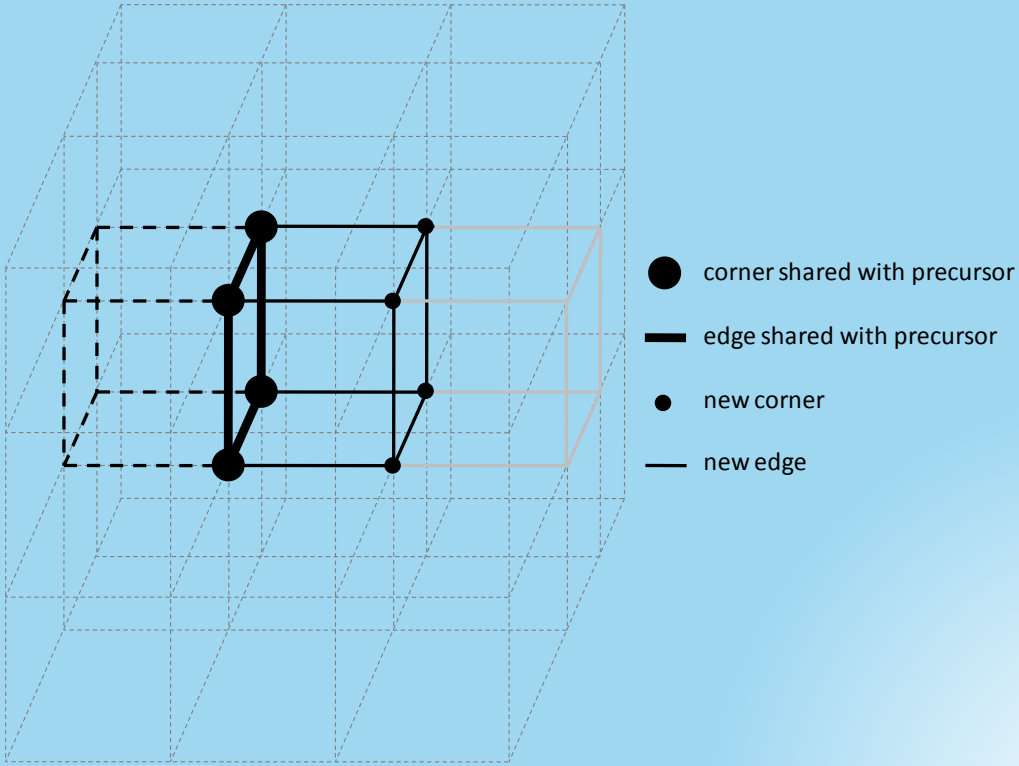
3) Isosurface rendering with marching cubes

- marching cubes continued
  - 8-bit corner-code (1 corner below, 0 otherwise) identifies the 256 possible triangle configurations (15 distinct)



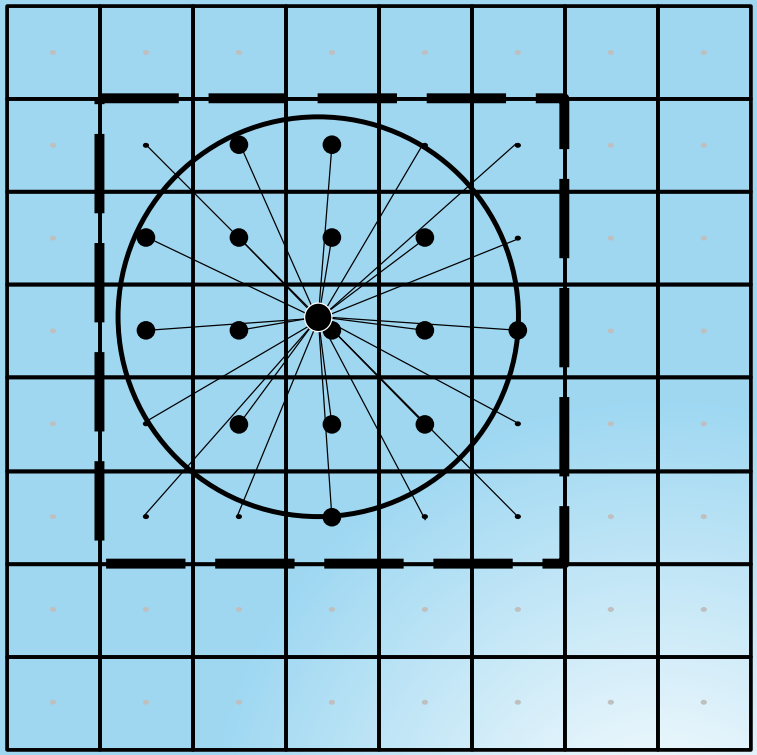
3) Isosurface rendering with marching cubes

- implementation optimisation:
  - reuse the pre-calculated data of precursor cube



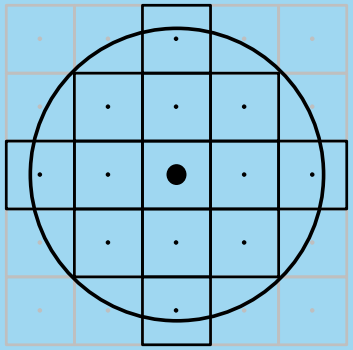
3) Isosurface rendering with marching cubes

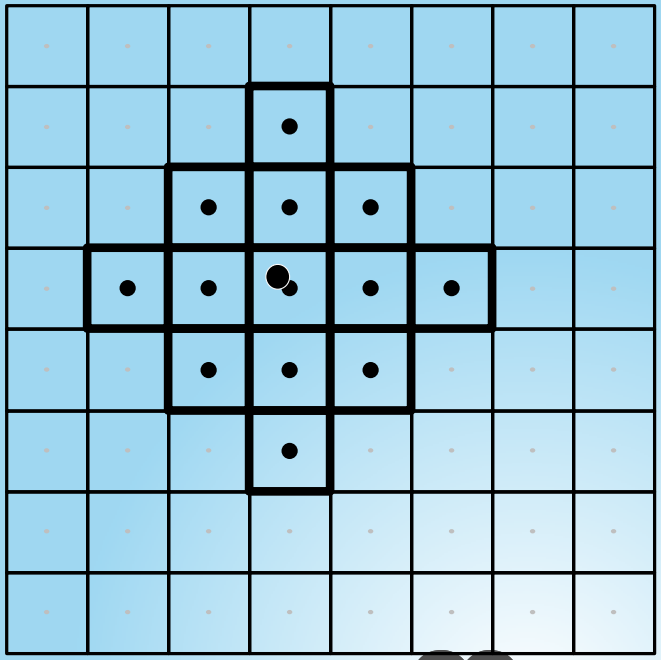
- density field construction, simple approach
  - for each particle consider all voxels within a cube around the particle
  - for each of these voxels calculate the additional density and add it to the voxels value



3) Isosurface rendering with marching cubes

- density field construction, stamp approach
  - assume that particle movement inside one voxel has no significant affect on the densities
  - construct a stamp containing precomputed additional densities
  - during construction, only identify the voxel the particle lies in and apply the stamp



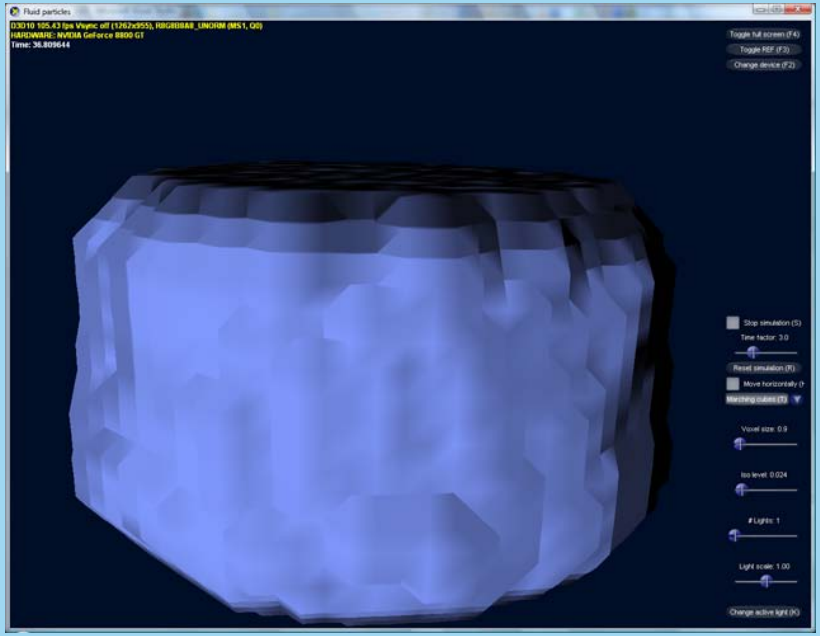


3) Isosurface rendering with marching cubes

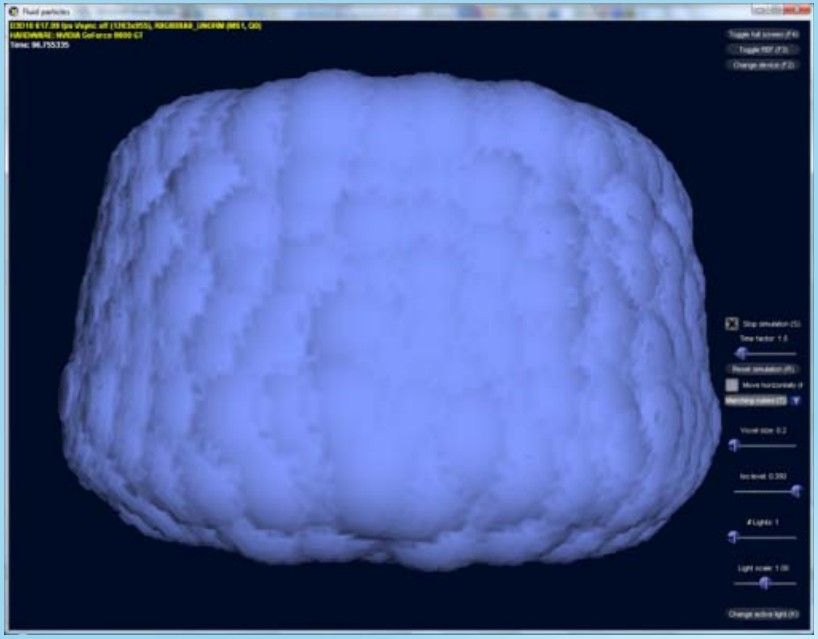
- results
  - test machine runs density field construction, marching cubes mesh creation and visualisation rendering at 150 FPS for 1728 particle and 0.9^3 voxel-size
  - with activated simulation: 113 FPS (-> algorithm is slow)
  - voxel-size has great influence: 1.8^3 -> 499 FPS, 0.4^3 -> 17 FPS
  - mesh construction only (295 FPS) is faster then grid creation only (220 FPS)
  - mesh rendering only is so fast that it’s not measurable (same FPS as program doing “nothing”)

3) Isosurface rendering with marching cubes

- optical results:



voxel-size 0.9^3



voxel-size 0.1^3

3) Isosurface rendering with marching cubes

- pros
  - triangle mesh is exactly what 3D APIs are intended for
  - -> bunch of visualisation method applicable
- cons
  - realtime density and mesh creation to expensive
  - mesh looks square-cut despite interpolations
  - low voxel sizes result in over tessellated meshes

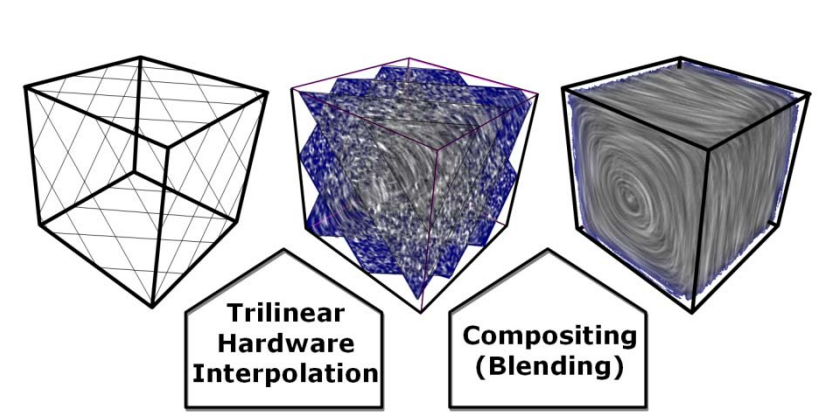
3) Isosurface rendering with marching cubes

- further work
  - use mesh smoothing
  - use GPU-based density grid construction
    - will be introduced later
  - use GPU-based marching-tetrahedra algorithm
    - available in Nvidia SDK 10

Contents

4) GPU-based isosurface raytracing

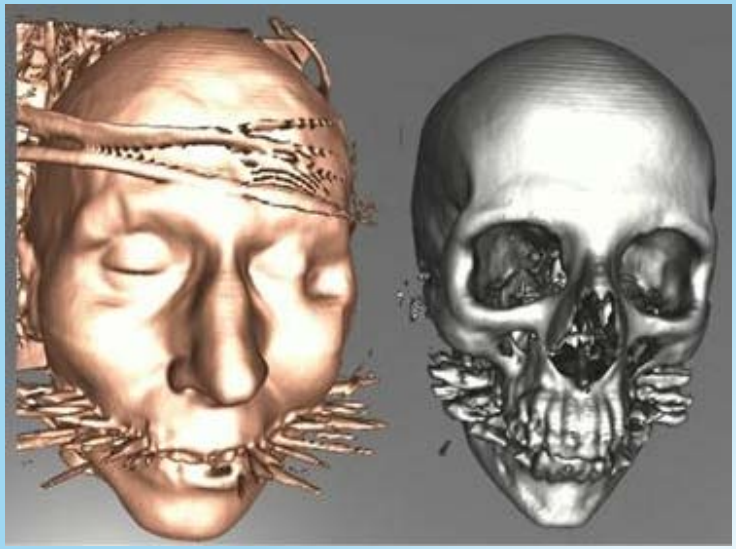
- direct volume rendering (DVR) often implemented via 3D texture slicing:



- disadvantage: only small percentage of the drawn (and textured) fragments contribute to the final image

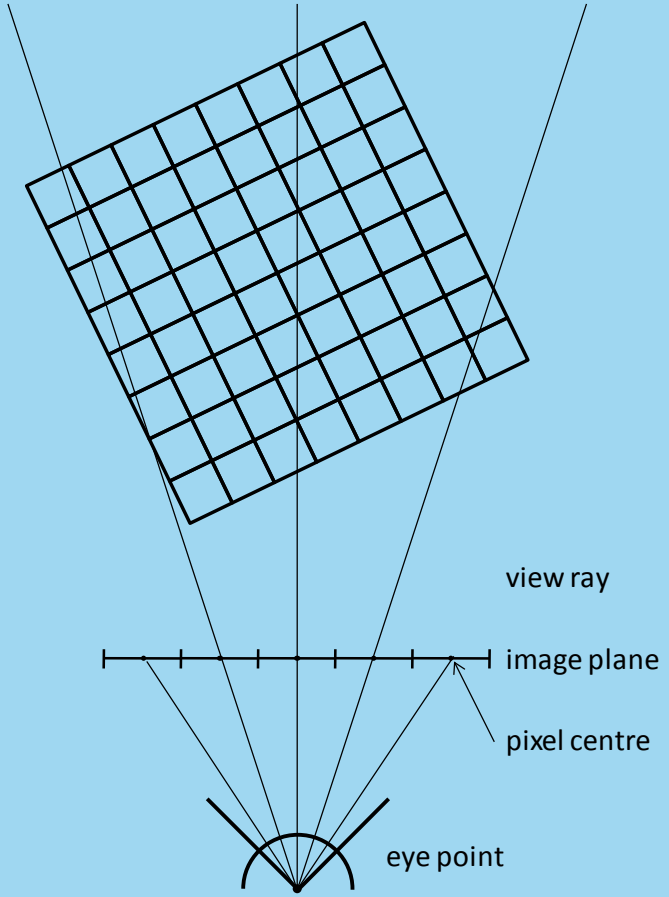
4) GPU-based isosurface raytracing

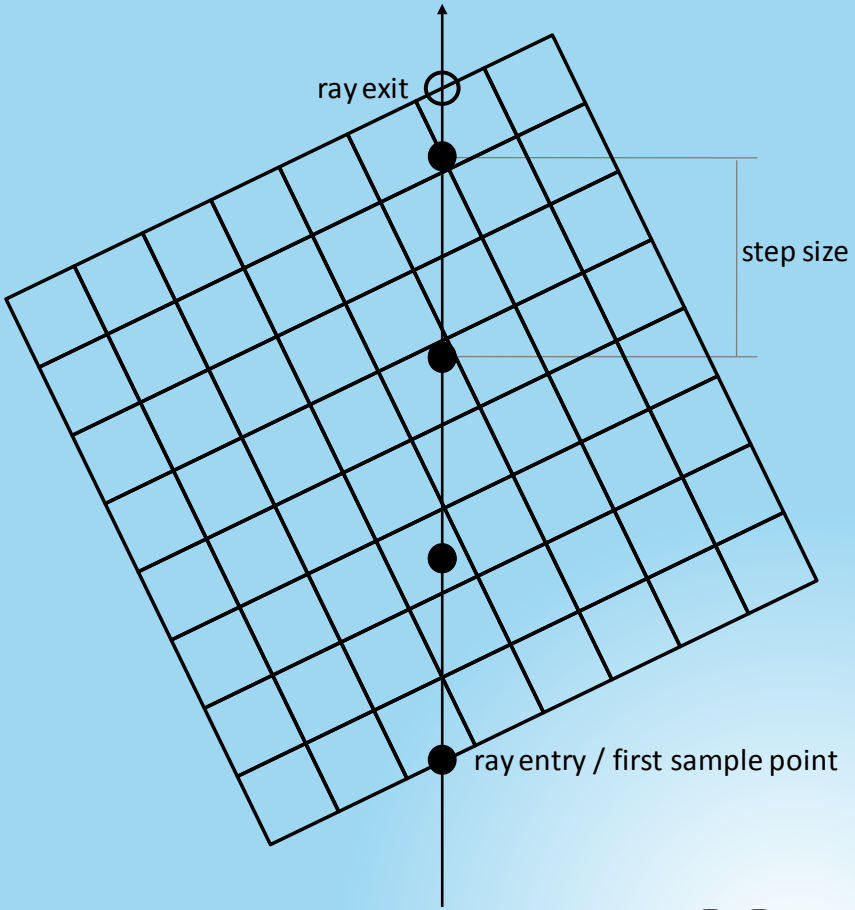
- therefore, Krüger and Westermann suggested volume ray-casting to perform only contributing texture samples
- ray-casting allows also iso-surface visualisation



4) GPU-based isosurface raytracing

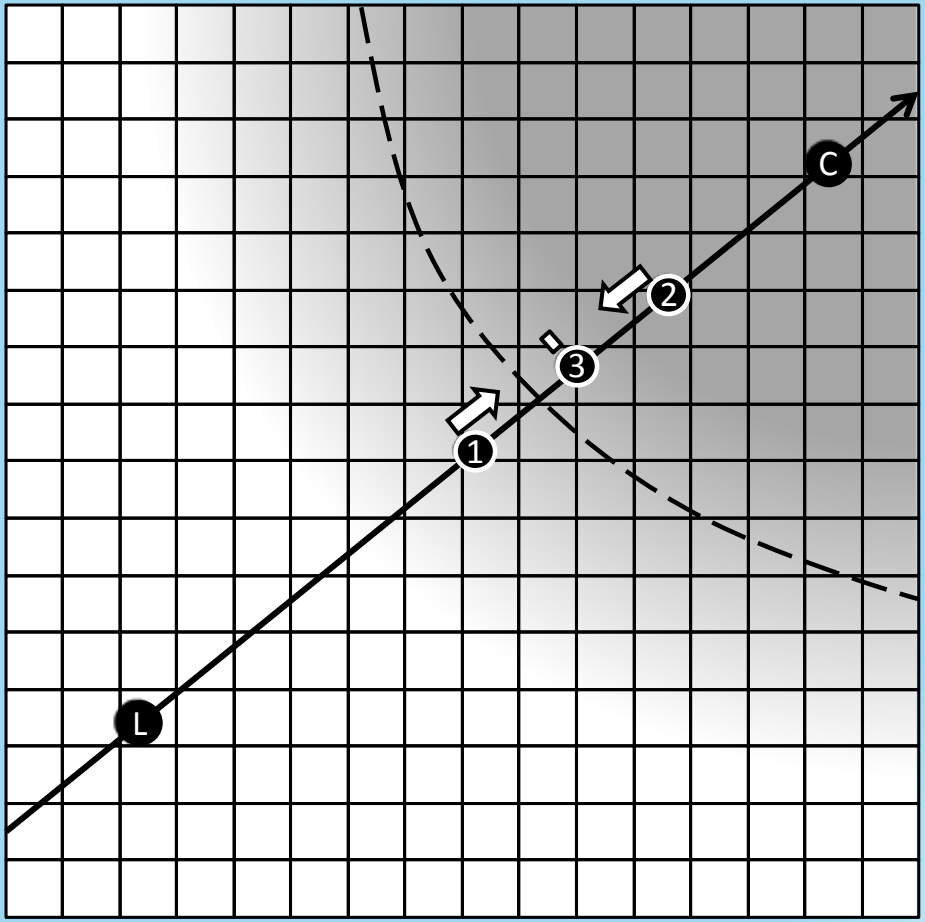
- volume ray-casting principle





4) GPU-based isosurface raytracing

- isosurface ray-casting hit refinement

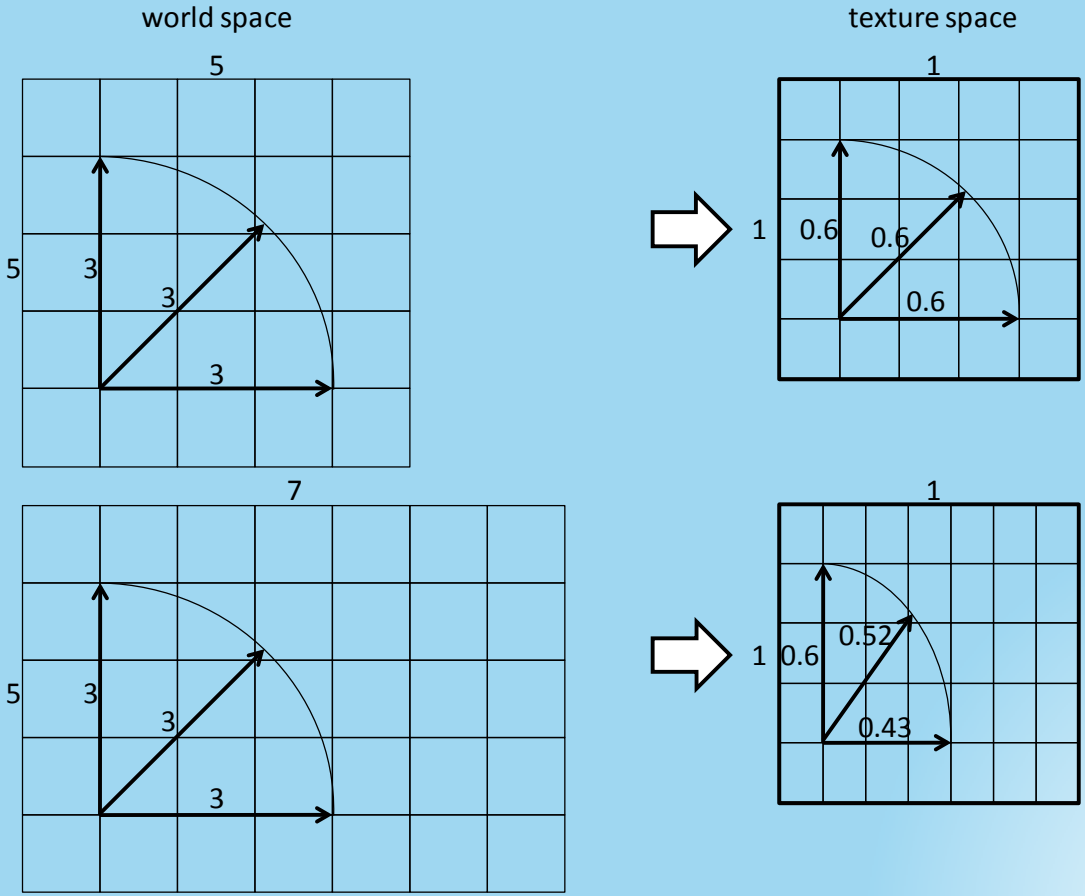


4) GPU-based isosurface raytracing

- ray creation
  - render bounding box
    back faces first, then front faces
  - leads to position of entry and exit point
- sampling
  - ray-cast passes calculate M sampling
    points along the ray and sample the volumetric texture
  - texture size not equal on each dimension -> per pixel step-size required

4) GPU-based isosurface raytracing

- cause of per pixel step-size



4) GPU-based isosurface raytracing

- ray termination
  - after each sampling pass a intermediate pass checks the stop criterion (opacity reached, isosurface hit)
  - when criterion matched, z-buffer is set to maximum
    -> early z-test skips pixel/ray in following passes

4) GPU-based isosurface raytracing

- this was shader model 2.0
- with today's shader model 4.0 things are much more comfortable
  - discard command allows fragment termination everywhere in the pixel shader (no more stop passes)
  - improved flow control and branching allows whole ray-casting (or even ray-tracing in our case) in one pass (no more texture sampling for transfer of results between passes)
  - only remaining extra-pass is exit point determination (because it's sill a clever way to do this)

4) GPU-based isosurface raytracing

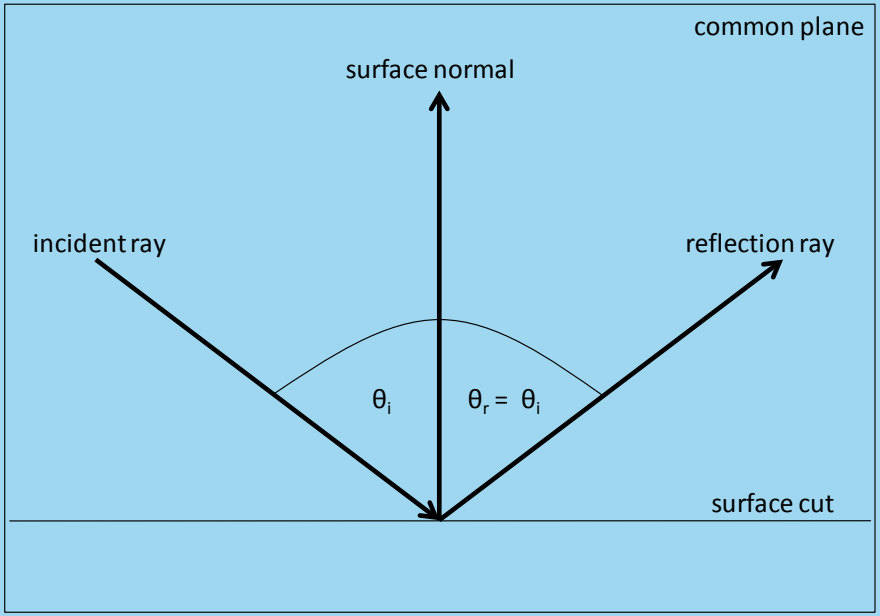
- so we are able to do isosurface ray-tracing now
- what could we use this for?
- answer: ray-tracing is perfect for rendering optical effects at water surfaces

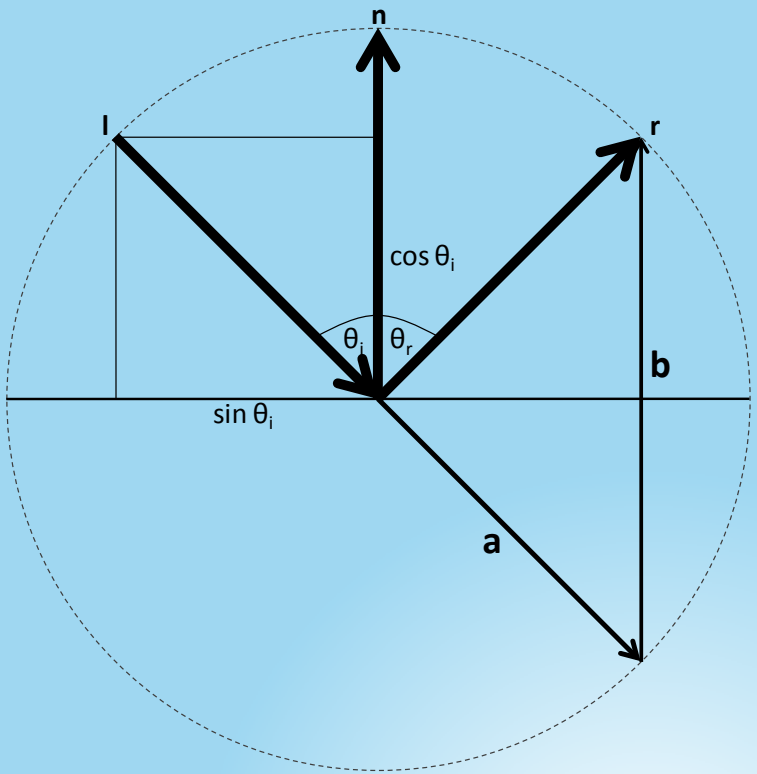
4) GPU-based isosurface raytracing

- optical appearance of water results mainly from reflection and refraction
- three questions must be answered when a view ray (or light ray) hits a water surface:
  - which direction has the reflected ray?
  - which direction has the refracted ray?
  - how is the light distributed among the rays?

4) GPU-based isosurface raytracing

- which direction has the reflected ray?
- law of reflection

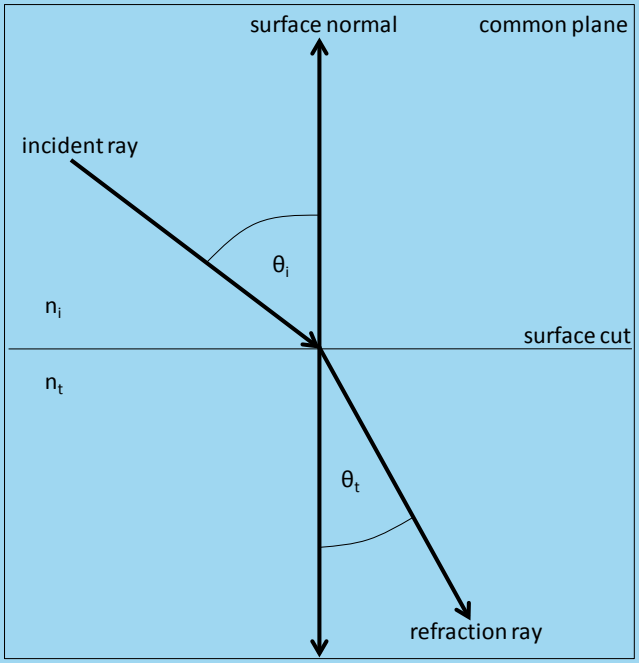


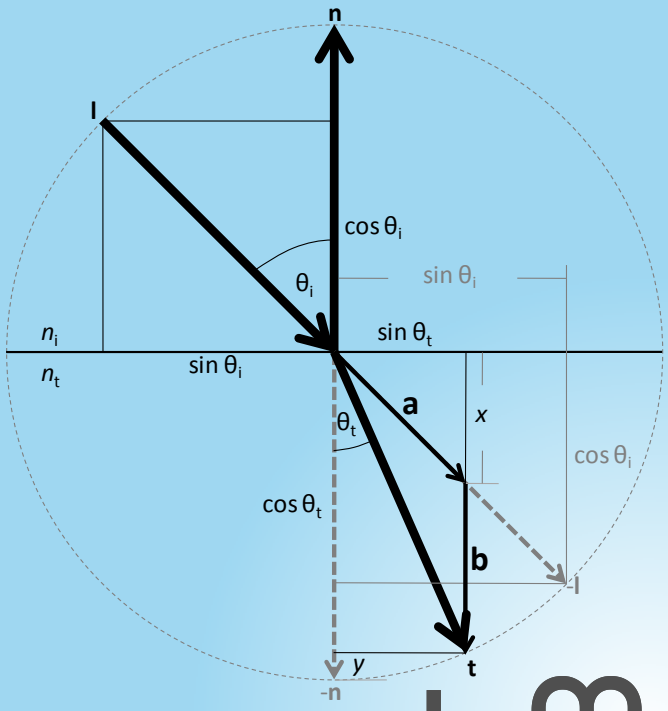


4) GPU-based isosurface raytracing

- which direction has the refracted ray?
- Snell's law (law of refraction)

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





4) GPU-based isosurface raytracing

- how is the light distributed among the rays?
- Fresnel equations

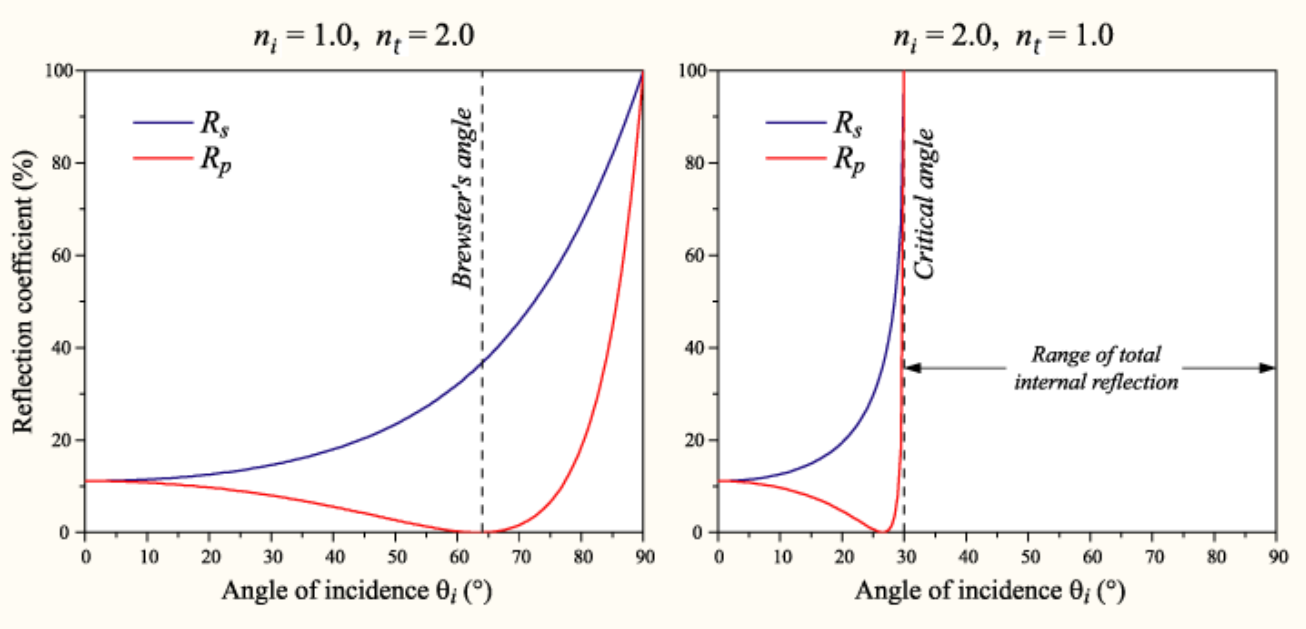
$$R_s(\theta_i) = R_\perp = r_\perp^2 = \left(\frac{n_i \cos\theta_i - n_t \cos\theta_t}{n_i \cos\theta_i + n_t \cos\theta_t}\right)^2 = \left(-\frac{\sin(\theta_i - \theta_t)}{\sin(\theta_i + \theta_t)}\right)^2$$

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

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

4) GPU-based isosurface raytracing

- plot of reflectance for refraction ratio 1/2 and 2/1



- reflectance for unpolarised light: $R = \frac{1}{2}(R_s + R_p)$

4) GPU-based isosurface raytracing

- combining all that equations leads:

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

- remembering Snell's law, define c and g as:

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

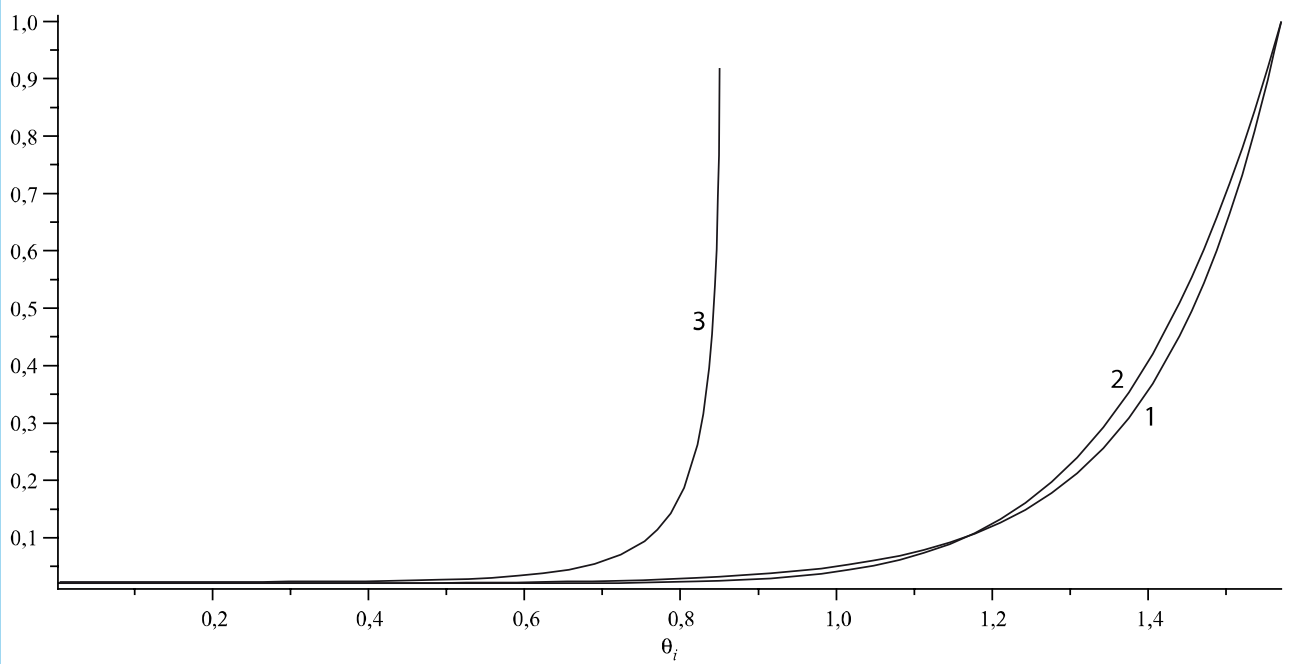
- results in the so called Fresnel term:

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

4) GPU-based isosurface raytracing

- for $n_i < n_t$ it can be approximated with:

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



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

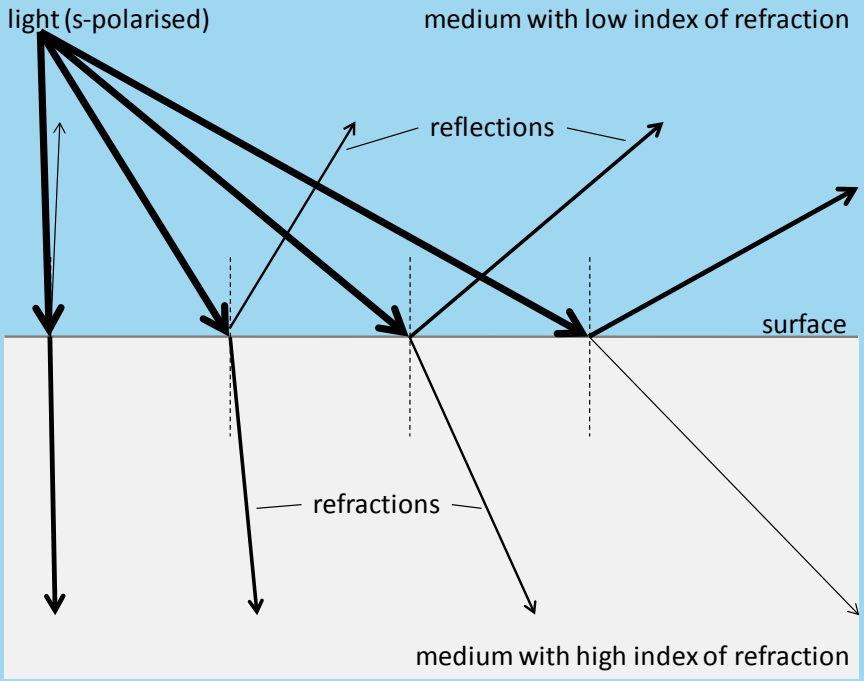
2: $R_a(\theta_i)$ for $\frac{n_i}{n_t} = 0.75$

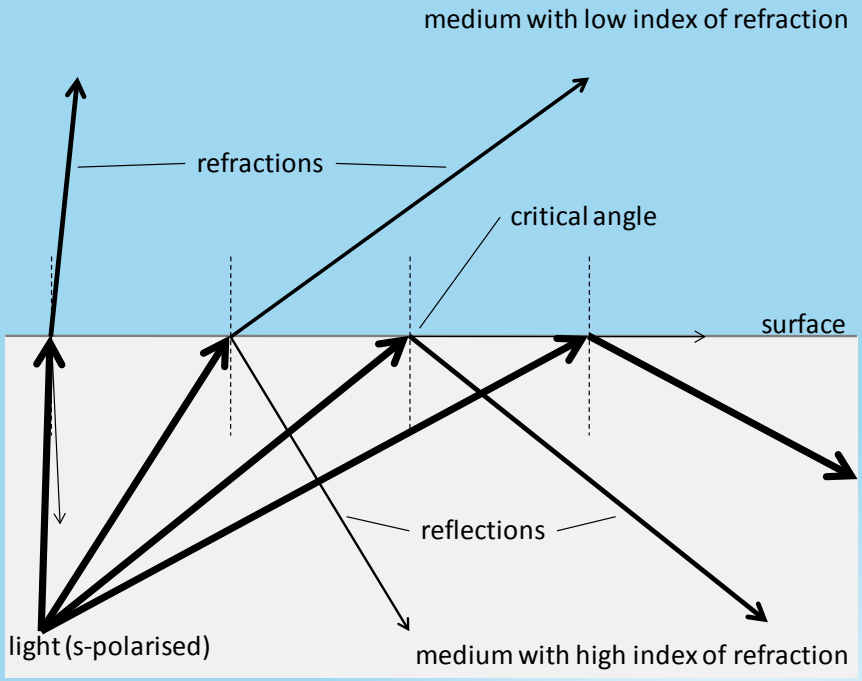
and $\frac{n_i}{n_t} = \frac{1}{0.75} = 1.3333$

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

4) GPU-based isosurface raytracing

- overall effect:





4) GPU-based isosurface raytracing

- overall effect on real water surfaces

1 2 3 4

Figure 1: Water surface - optical effects

1: water to air + low view angle -> total internal reflection

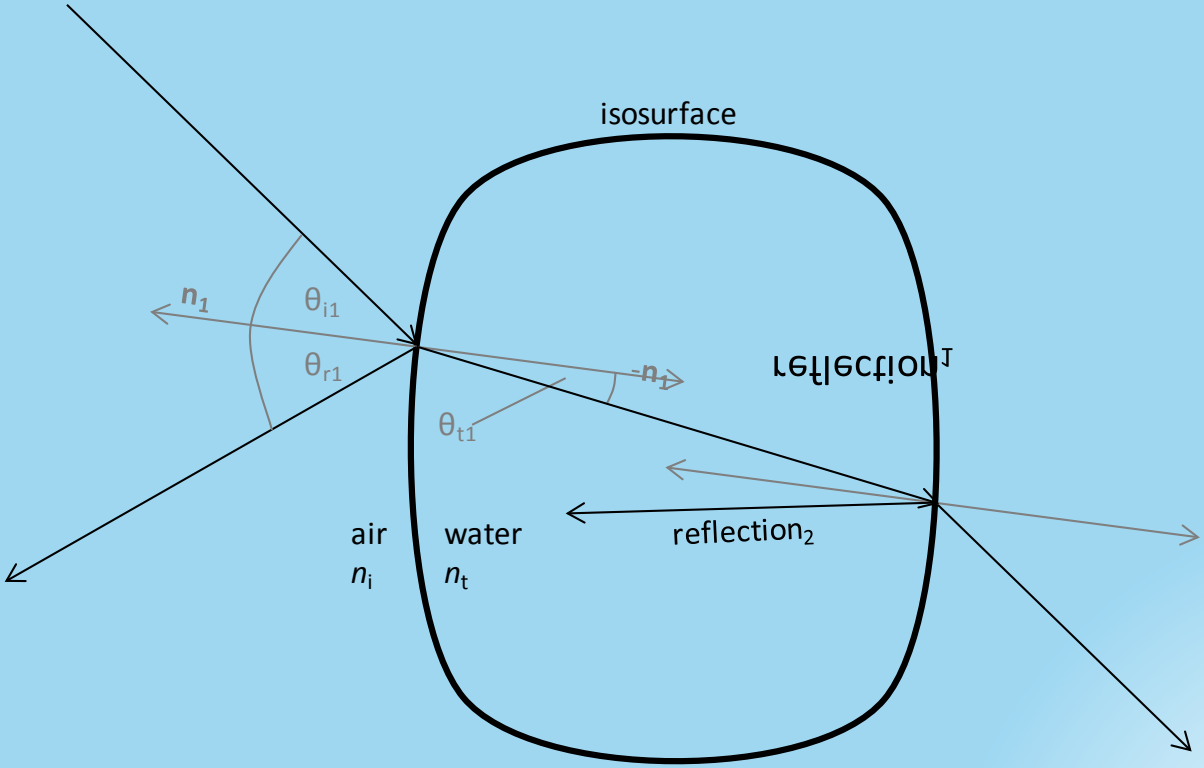
2: low view angle -> reflection dominates; strong refraction on round surface -> mirror-inverted

3: higher view angle -> refraction dominates

4: view rays bend down

4) GPU-based isosurface raytracing

- ray-tracing depth



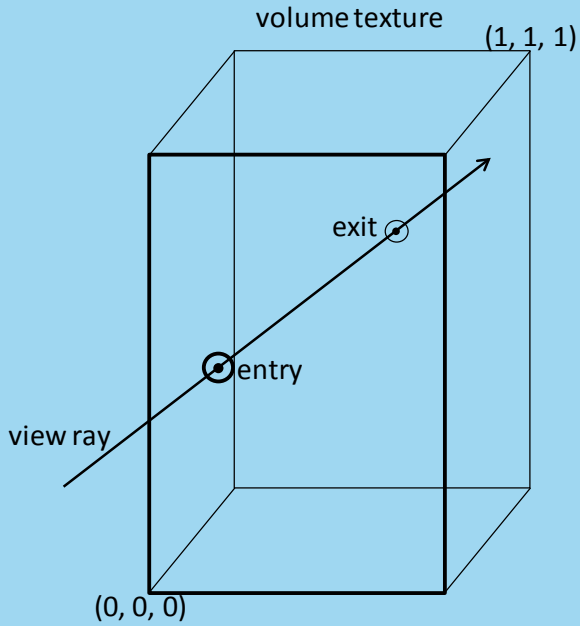
4) GPU-based isosurface raytracing

- implementation

```
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);
}
```

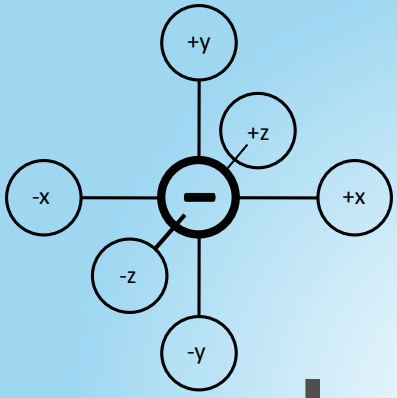
4) GPU-based isosurface raytracing

- RenderVolume simply renders bounding box two times
  - first time: back-faces -> ray exit
  - second time: front-faces -> main ray-tracing
- ray-tracing therefore begins with ray entry and exit point



4) GPU-based isosurface raytracing

- ray-tracing steps
  - define step-size
  - calculate offset vector (from one sample to the next)
  - sample along the ray in a loop (beginning with the entry)
  - stop the loop is a sample value higher than isolevel is found
  - if now intersection is found, discard the fragment
  - refine the intersection with binary search between the last two sample points
  - Calculate surface normal from the gradient



4) GPU-based isosurface raytracing

- ray-tracing steps continued
  - calculate reflection direction (used later for environment-map lookup)
  - calculate refraction direction
  - calculate volume exit point of refraction ray (ray-box intersect test)
  - perform second iso-surface raycast along the refraction ray
  - leads to exit intersection with isosurface
  - calculate reflection direction for exit intersection
  - evaluate Fresnel term for exit i.

4) GPU-based isosurface raytracing

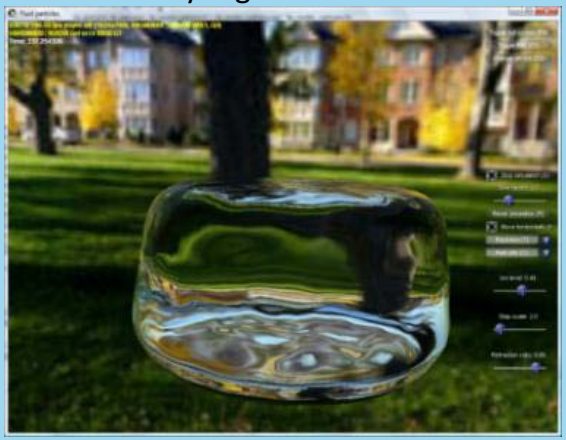
- ray-tracing steps continued
  - if second refraction ray exists (no total internal reflection) calculate its direction
  - calculate the first Fresnel term
  - sample reflection 1, reflection 2 and refraction 2 (if existent) colour-values from the environment map
  - combine the colours according to the two Fresnel terms

4) GPU-based isosurface raytracing

- optical results

only slight distortion

water refraction index



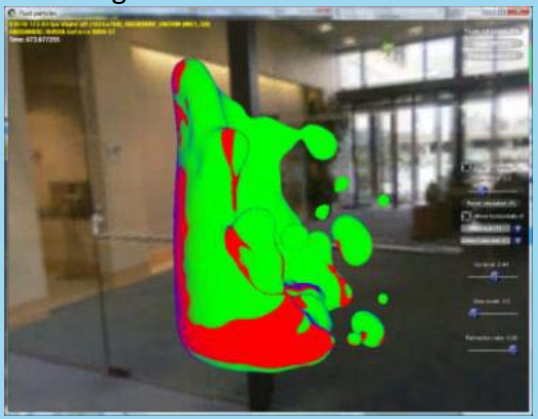
refraction becomes mirror-inverted

diamond refraction index

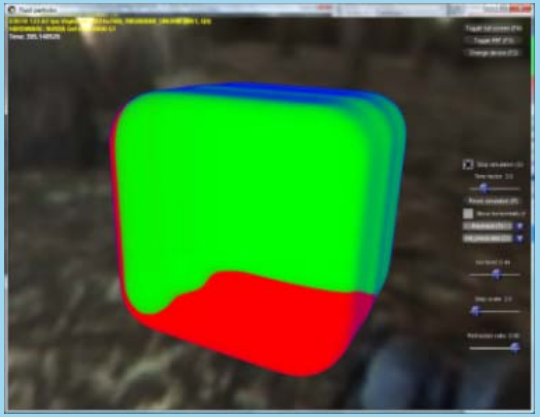
4) GPU-based isosurface raytracing

- colour-coded results

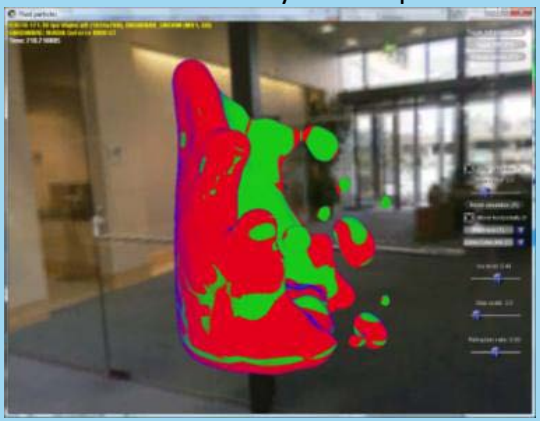
green: transmission blue: external reflection red: internal reflection



internal reflection only on oblique surface



external reflection mainly on top side



heavy internal reflection

slight external reflection everywhere

4) GPU-based isosurface raytracing

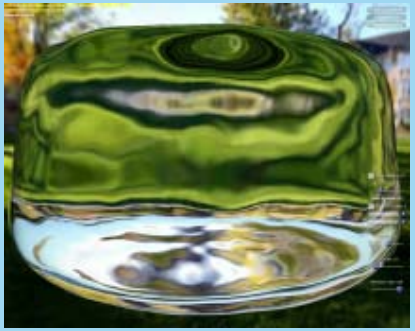
demonstration?

4) GPU-based isosurface raytracing

- performance results
  - test machine:
    - CPU: Intel Core 2 Quad Q6600 @ 3.2 GHz
    - RAM: 2 GB
    - Graphics card: Nvidia Geforce 8880 GT 512 MB video memory
    - GPU clock s.: core 640 / shader 1600 / memory 950 MHz
    - OS: Windows Vista x64 (program is x64 binary)
  - render options
    - screen-mode: 1280x1024 full-screen
    - simulation: all possible forces, 1728 (12^3) particles
    - visualisation: all possible effects

4) GPU-based isosurface raytracing

- performance results continued
  - water surface rendered
    screen filling: 65 FPS
  - 1/9 screen coverage: 161 FPS
  - water surface off-screen: 288 FPS
    (6% slower then sprite rendering)
- memory latency hiding effects:
  - performance depends more on
    core- & shader- then memory clock
  - nonetheless longer computations
    before texture sampling are “for free”



4) GPU-based isosurface raytracing

- pros
  - sufficient performance
    - view dependent approach
    - implicit LOD
  - very good optical results
    - smooth water surface
    - minimal grid related motion artefacts
    - optical effect quality that is normally only known from offline ray-tracers

4) GPU-based isosurface raytracing

- cons
  - no empty space skipping
    - on the fly octree generation costly
  - limited spatial flexibility
    - spatial limited grid needed
  - optical interaction with 3D environment difficult
    - ray-tracing hard to seamlessly integrate into rasterised renderer

4) GPU-based isosurface raytracing

- further work
  - develop real-time empty space skipping
  - break spatial limitations
    - build only one grid for the whole view
    - in normalised device space
  - allow optical interaction with 3D environment
    - outgoing rays:
      - on-the-fly environment map
      - spatially limited again
      - some sort of ray-tracing
    - internal rays:
      - parallax occlusion mapping for depth buffer?
      - some sort of ray-tracing

4) GPU-based isosurface raytracing

- outlook
  - interest in realtime ray-tracing is growing
    - hardware and APIs become more general in the future
  - perhaps isosurface ray-tracing easier to integrate into renderers of the future

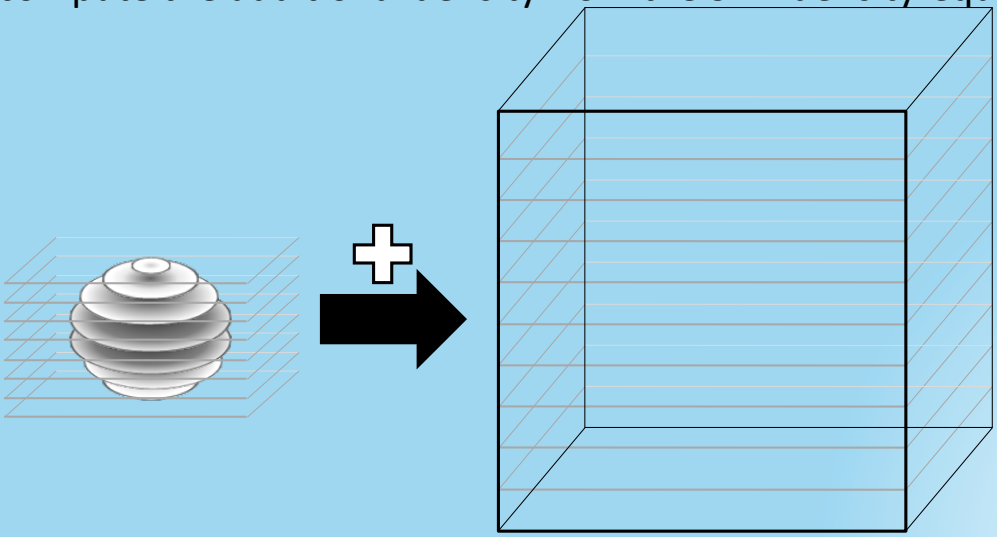
Contents

5) GPU-based volumetric density field construction

- we have left out GPU-based density field construction
- because it's a more general technique
  - could be interesting for many visualisations
- crucial for the overall performance
  - CPU version showed how it shouldn't be
- algorithm is highly parallelisable and extremely dependent on memory performance
  - best case for GPGPU

5) GPU-based volumetric density field construction

- basic principle:
  - for each particle
    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
    5. and compute the additional density from the SPH-density-equation

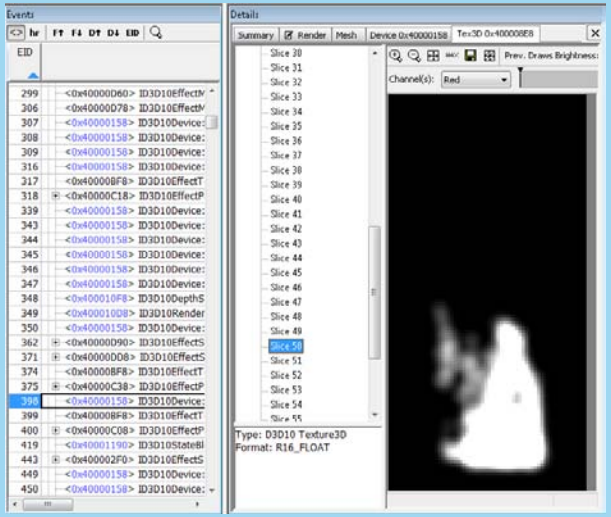


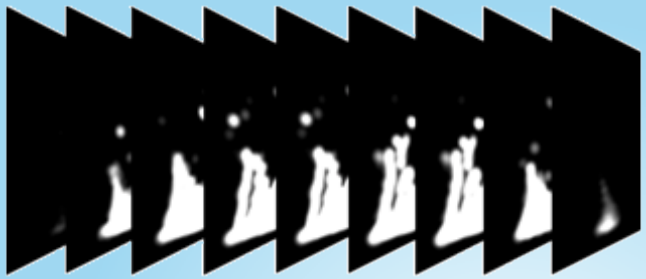
5) GPU-based volumetric density field construction

- implementation
  - use instanced draw
    - instance-count = number of slices influenced by single particle
  - use geometry shader
    - constructs billboards for affected slice regions
    - chooses the target slice (render-target index)
  - pixel shader computes additional density
    - put most of density equation into 1D texture
  - use additive blending
    - mimics summation

5) GPU-based volumetric density field construction

- results
  - 2016 particles
  - grid texture size 64x128x64
  - particle voxel diameter 13
  - texture fill only: 614 FPS
    (compared to 850 “doing nothing”)
  - simulation only: 285 FPS
    sim. + texture fill: 273 FPS
  - -> sufficient performant





6) Alternatives and further work

- many alternative techniques for isosurface rendering
  - surface splatting for example
  - evaluating each for its usability in realtime particle based fluid simulation would be way out of scope
- interesting approach from Müller 2007: screen-space-meshes
  - no density grid built
  - operates only in 2D screen space
  - instead of finding an isosurface, particles are drawn as (imaginary) spheres and a smoothed mesh is built from the result

6) Alternatives and further work

- major advantage of screen-space-meshes:
  - nothing is created that could be used for more than the current view
  - -> good idea for realtime visualisation of realtime simulation
- interesting idea:
  - combine something like s. s. m. with “interactive screen space photon tracing” (Krüger, Bürger, Westermann 2006)
  - supports meshes, supports reflections and refractions, supports related caustics and god-rays

Contents

IV. Conclusion

- thesis demonstrated how it's possible to:
  - interactively simulate fluids (especially liquids)
  - render the results in ray-tracing quality
- prediction:
  - Within a few years, fluid simulation will be seen in many interactive applications (especially video games).
  - interesting side note:
    - Matthias Müller is the head of research at AGEIA
    - Nvidia recently bought AGEIA
    - free CUDA accelerated particle based fluid simulation to come soon?
    - be prepared with good visualisations ;-)

fin