Akademia Górniczo - Hutnicza im. Stanisława Staszica w Krakowie
Wydział Elektrotechniki, Automatyki, Informatyki i Elektroniki
Katedra Informatyki

Tytuł pracy

Wykorzystanie środowiska PhysX do modelowania przepływów

Piotr Janisz
piotrekjanisz@gmail.com

Promotor pracy: dr inż. Witold Alda

Kraków, 2012

Abstract

Computing power growth allows more complex and accurate simulation techniques to be implemented in real-time applications. An example of those techniques are particle-based methods for simulation of fluids. They can provide a new level of realism into computer games. Nowadays most fluids in games are simulated using height field. Although using this method realistic waterbodies (like oceans or ponds) can be simulated, it's hard to achieve effects such as splashing or flooding. Those can be easily simulated with particle-based methods such as Smoothed Particle Hydrodynamics (SPH).

In this paper I would like to present realistic model of fluid that can be used in real time application, especially in computer games. Emphasis will be placed on realistic rendering output from particle simulation. For water simulation NVIDIA PhysX SDK will be used. For rendering particles I will use two different approaches: screen space rendering and isosurface extraction. Second algorithm is described in (1) and it is running on cpu. Authors presented performance of this algorithm running on one CPU core and I will present multithread implementation and its performance analysis. At the end of this paper I will also present comparison of those two rendering techniques.

Contents

1

Introduction

1.1 Motivation

Simulation of reality have always been very important part of computer games. Ambition of developers is to make virtual worlds more and more realistic. This is however a very difficult task, as games are interactive and requires real-time simulations. That prevents from using state of the art models of reality as they are designed for off-line simulations on large computer clusters. New algorithms have to be developed or some simplifications in existing models have to be made to fit them into limited computational resources and time constraints. Over the years tremendous progress have been made in computer games realism. Continuous growth of computer's hardware computational power enabled to perform real time simulations that were unthinkable 10 years ago.

Simulating fluids is an important part of reality simulation. It is also very expensive to run physically accurate simulations of fluid on todays hardware. To overcome that, various efforts have been made. First of them is to decrease simulation dimensionality from 3 to 2 - thus treating water surface as a function of two variables - $f(x, y)$. This approach allows to achieve realistic oceans, lakes etc. (see figure 1.1), however it can't be used to simulate splashes or breaking waves. It is also hard to incorporate interactions with other object (like rigid bodies). For achieving such effect particle based simulations can be used. They approximate fluid as a large number of particles which interact with each other and with other objects. This is better than Eurlerian approaches as we can only focus on areas where fluid is present and don't have to cover entire domain

with grid. Eulerian, Lagrarian and particle base approaches will be further described in chapter 2.

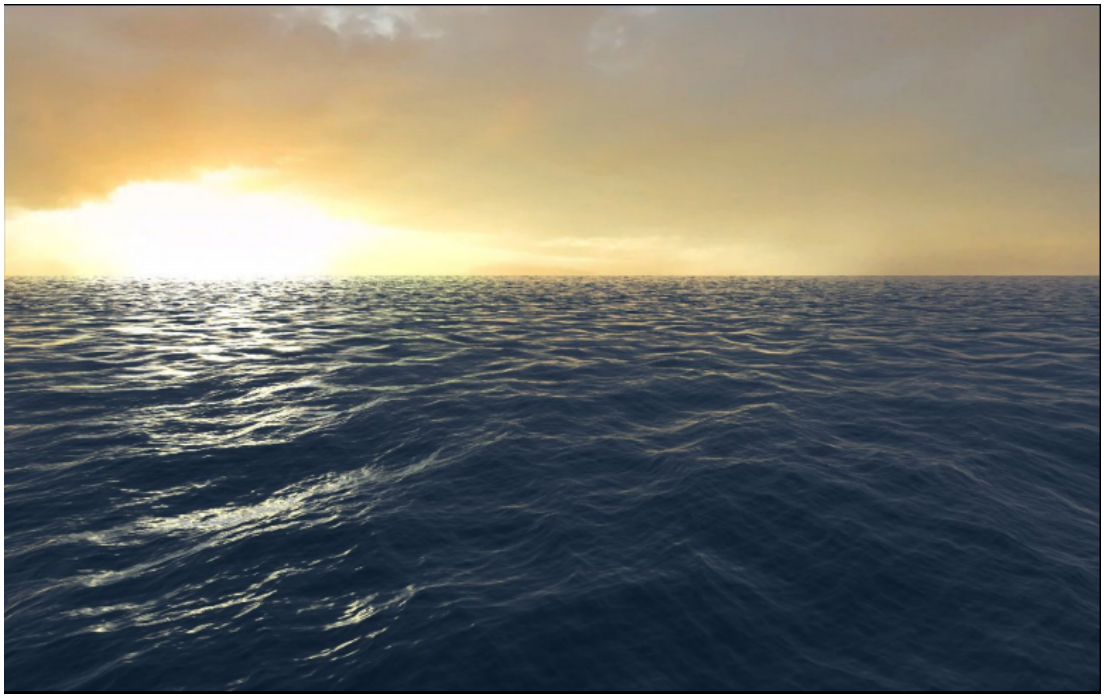


Figure 1.1: Ocean surface simulation - Procedural approach to simulate ocean surface. Image taken from (2)

One of particle base methods is SPH (see chapter 2). It produces realistic fluid simulations with effects that are unavailable for procedural or height-field models. What is more it is based on a set of simple rules and governing equations. Realistic simulations using this method requires tens of thousands particles to be used. This cannot be done in real time using CPU, however it can be efficiently implemented on modern GPUs. Their massively parallel architecture allow to simulate over 100 thousands of particles in real time.

Although fluid simulation using particle based approach is easy, rendering output from the simulation is harder. Unlike height field based approach this does not produce triangle mesh. Some effort is required to make the output from simulation look like a water surface instead of bunch of spheres. The fluid surface has to be extracted somehow each time.

The goal of this paper is to investigate latest rendering techniques for fluids. I will

also compare performance of those techniques on latest hardware. PhysX physics engine will be used to perform SPH simulation of fluid and OpenGL to render the results.

2

SPH

This chapter will provide brief introduction to modeling method used in my project - Smoothed Particle Hydrodynamics (SPH).

2.1 Introduction

To perform numerical simulation of natural phenomena several steps have to be performed (see (3, section 1.1.2)). First the mathematical model has to be obtained from observation of physical phenomena. The mathematical model consists of a set of governing equations, boundary and initial conditions. Next governing equations have to be discretized in order to solve them with a numerical algorithm.

For describing governing equations of fluid flows two basic techniques exists: Lagrarian and Eulerian. In the first one we look at the fluid as a set of parcels which moves though space and time. Each fluid parcel has it's own mass which is fixed, and is identified by it's position x_0 at time $t = 0$. We also assume that we can obtain a function of fluid parcels position $x = x(t, x_0)$. This is demonstrated on figure 2.1. Eulerian description assumes that we observe fluid that moves through fixed spatial points. Thus we are given velocity field v at every spatial point x and time t - $v = v(x, t)$ (figure 2.1). More on fluid field description can be found in (4, section 2.1) and (5)

For discretization there are several approaches: grid based (Lagrarian grid and Eulerian grid) and meshfree methods (SPH, particle in cell). Lagrarian grid is attached to material and moves with it. Each mesh cell represents one fluid parcel from Lagrarian description. The advantages of this method are simpler equations given by lagrarian

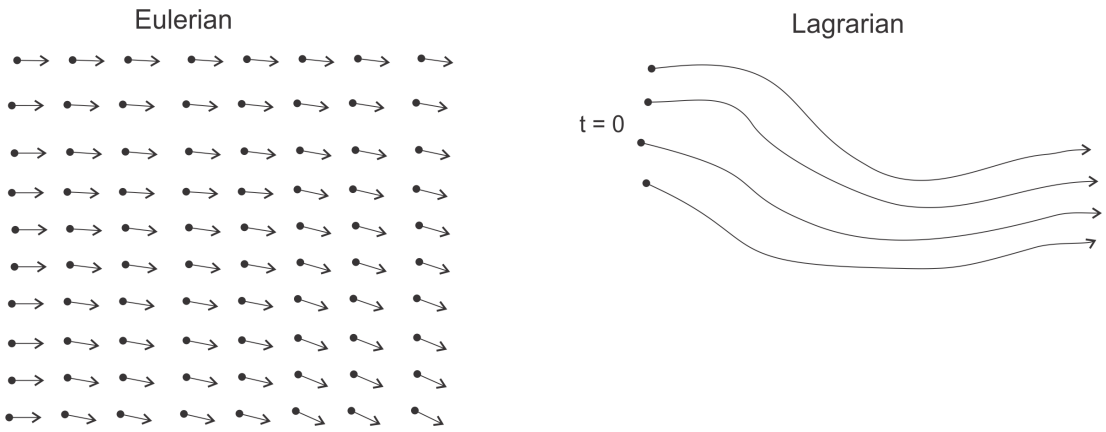


Figure 2.1: Different fluid observation techniques - on the left we can see Eulerian velocity field, on the right Lagrarian pathlines.

description, less memory usage (as the mesh is specified only where fluid or other material is present), easiness of tracking movement of fluid parcels. However this method does not handle large material deformations well (like for instance water splashes). In such cases mesh must be recomputed periodically and it is an expensive procedure.

Eulerian grid, on the other hand, is fixed in space and covers entire domain in which fluid can move. In each grid node we calculate mass momentum and energy flux. This method handles large deformations of material easily but its computational complexity is large, especially in three dimensions. The amount of required memory is also large as whole domain must be covered with grid.

Third way of discretization are mesh free methods. The idea behind is to combine strong points of both eulerian and lagrarian grids, but without using a grid or a mesh. This is done usually by discretizing material as a set of particles. At each particle values of functions, derivatives and integrals are approximated with particle approximations. An example of mesh free method is SPH which will be further described in this chapter. Lagrarian and Eulerian grids as well as meshfree methods are described in more details in (3, chapter 1)

2.2 Fundamentals of SPH

The main idea of SPH is to replace given field A with it's integral interpolant (or kernel interpolant):

$$A_I(r) = \int A(r')W(r - r', h)dr' \tag{2.1}$$

Where W is a smoothing kernel and has following properties:

$$\int W(r - r', h)dr' = 1 \tag{2.2}$$

$$\lim_{h \to 0} W(r - r', h) = \delta(r - r') \tag{2.3}$$

$$(\exists \kappa \in \mathbb{R}_+)(W(r - r', h) = 0 \; when \; |r - r'| > \kappa h) \tag{2.4}$$

Integral interpolant 2.1 can be further approximated by sumation interpolant:

$$A_s(r) = \sum_b m_b \frac{A_b}{\rho_b} W(r - r_b, h) \tag{2.5}$$

where $\frac{m_b}{\rho_b}$ is a volume of particle b. What SPH does, it divides space into set of particles, each particle has it's own mass and density. Mass is fixed and density can change thus changing particle's volume. When we want to compute value of A in some point in space r we compute weighted average of A at neighboring particles. Weight depends on distance of particle's center form r and on particles volume.

It's also worth noting that 2.4 defines finite domain in which smoothing kernel is non-zero. This allows to speed up computations by summing over particles in nearest neighborhood.

Replacing field A with it's kernel interpolation A_I yields error of h^2 (see (3, section 2.2.1)). On the other hand accuracy of sumation interpolant 2.5 is hard to predict (see (6, section 12.1)).

Density - ρ - that appears in sumation interpolant 2.5 varies in time as opposed to mass. Thus it must be computed at each time step. Density field can be obtained using sumation interpolant as follows:

$$\rho(r) = \sum_b m_b \frac{\rho_j}{\rho_j} W(r - r_b, h) = \sum_b m_b W(r - r_b, h) \tag{2.6}$$

In many physical simulations we need to compute gradient of some field A. It turns out that using integral interpolant 2.1 we only need to compute gradient of smoothing kernel W ((3, section 2.2.2)):

$$\nabla A_I(r) = \int A(r')\nabla W(r - r', h)dr' \tag{2.7}$$

As earlier we can aproximate integral with sumation:

$$\nabla A_s(r) = \sum_b m_b \frac{A_b}{\rho_b} \nabla W(r - r_b, h) \tag{2.8}$$

This simplifies computations as W is known and it's gradient can be given analitically.

SPH gives a flexibility in choosing smoothing kernel. Gaussian function would be preferable however it doesn't meet compact condition 2.4. Thus Gaussian-like functions are usually constructed, although any function that satisfies 2.2, 2.3 and 2.4 can be used. It is even possible to utilize variable width kernels (6, section 6) which can be useful in case of large fluctuations in particles distributions. (3, chapter 3) is entirely devoted to smoothing kernels and their construction, (7, section 3.5) discuss usage of different kernels.

2.3 SPH for fluids

Fluid motion can be described by two equations. First one is conservation of mass:

$$\frac{DM}{Dt} = 0 \tag{2.9}$$

second one is Navier-Stokes conservation of momentum equation:

$$\rho \frac{Dv}{Dt} = -\nabla p + \mu \nabla^2 v + f^{surface} + f^{external} \tag{2.10}$$

Both equations are presented in Lagrarian form. $\frac{D}{Dt}$ is a material derivative (see (4, section 2.2)), p is the pressure, μ is fluid viscosity, $f^{surface}$ represents surface tension force and $f^{external}$ represents external forces acting on the fluid. Usually external forces consists only of gravity giving $f^{external} = \rho g$. When using SPH, conservation of mass is guaranteed by particles, as mass of the particles does not change. Pressure term $-\nabla p$ at particle i can be estimated using 2.5 as follows:

$$-\nabla p_i = -\sum_j m_j \frac{p_j}{\rho_j} \nabla W(r_i - r_j, h) \tag{2.11}$$

Pressure force computed as above is not symmetric which can be seen when only two particles interact. Gradient of the kernel W is zero at particle's center and thus only particle j contributes to the pressure of particle i. As density differs between particles, this may lead to pressure forces not being symmetric. Many ways of symmetrizing equation 2.11 are proposed in literature (see (6, section 3.1), (3, section 4.3.2)). Very simple one was used by (7):

$$-\nabla p(r_i) = -\sum_j m_j \frac{p_j + p_i}{2\rho_j} \nabla W(r_i - r_j, h) \tag{2.12}$$

Pressure at particle i is defined as:

$$p_i = k(\rho_i - \rho_0) \tag{2.13}$$

where k is gas constant and ρ_0 is rest density.

SPH formula for viscosity force - $f^{viscosity} = \mu \nabla^2 v$ - acting on particle i will be:

$$f_i^{viscosity} = \mu \sum_j m_j \frac{v_j}{\rho_j} \nabla^2 W(r_i - r_j, h) \tag{2.14}$$

Again, this yields asymmetric force as velocity varies between particles. Fortunately it can be simply symmetrizied as follows:

$$f_i^{viscosity} = \mu \sum_j m_j \frac{v_j - v_i}{\rho_j} \nabla^2 W(r_i - r_j, h) \tag{2.15}$$

Surface tension force $f^{surface}$ can be modeled as presented in (7):

$$f^{surface} = -\sigma \nabla^2 c_s \frac{n}{|n|} \tag{2.16}$$

where σ is surface tension of fluid. c_s is so called color field which is 1 at particle locations and 0 everywhere else. It can be smoothed, which will give:

$$c_s(r) = \sum_j m_j \frac{1}{\rho_j} W(r - r_j, h). \tag{2.17}$$

n is a surface normal field pointing into the fluid and it is computed as gradient of c_s:

$$n = \nabla c_s \tag{2.18}$$

External force which is equal to gravity force, does not require using SPH and it's simply:

$$f_i^{external} = \rho_i g \tag{2.19}$$

It is also important that choosing smoothing kernel have significant impact on stability and correctness. (6) discusses impact of smoothing kernel on error of integral interpolant. (7) points out that gradient o kernel used for computing pressure force should not approach zero when r approaches zero. This would lead to creation of particle clusters as pressure force would decrease when particles are closer to each other.

Another essential restriction must be imposed on smoothing kernel of viscosity. As viscosity is the force that tends to decrease particles relative velocity. That is why it should only be dependent on particles velocity differences. However if Laplacian of the smoothing kernel gets negative the direction of the viscosity force gets inverted which leads to increase of particles relative speed. (7) proposed smoothing kernel which Laplacian is positive within radius of influence.

3

Tools

This chapter will present brief introduction of tools used in building visualization system. This is no intended to be a tutorial as those information can be found on the Internet.

3.1 OpenGL

OpenGL is an API specification for interacting with graphic hardware in order to produce 2D and 3D graphic. As a specification it is not bound to any programming language, operating system or hardware. OpenGL bindings exists for many languages including C/C++, Java, Python, Ruby and C♯ (see (8)).

Since introduction of version 3.2, specification has been divided into core profile and compatibility profile. Core profile is smaller and contains only modern style API introduced with version 3.0. Compatibility profile implements all legacy functionality prior to OpenGL 3.0, including fixed pipeline.

Fixed function pipeline is built-in to hardware and there is no control over how operations are performed on GPU. That means operations like vertex transformation, algorithms for shading surfaces are already implemented and can't be changed. The behavior may only be changed by choosing one of predefined methods or adjusting available parameters. Pros are that it's faster and less error prone to build an application. Programmers only have to choose existing technique and specify geometry primitives. On the other hand when new algorithm is invented there is no possibility to use it. There are also no possibilities to optimize existing techniques for the particu-

lar application. That is why programmable pipeline was introduced in modern GPUs. Rendering stages can be customized by writing programs called shaders. At first those programs were very specific and allowed little customization. Over time more and more capabilities were added to shaders and now their usage goes beyond graphic rendering (9, Part VI). Currently programmers have full control over how primitives are rendered on the screen.

Apart from core and legacy profiles there are extensions. This is a mechanism of including vendor specific functions, implemented only on some GPUs, that can make use of some new hardware features.

OpenGL is widely available on different hardware from graphic cluster, through desktop computers, game consoles and portable devices. For the last group dedicated standard has been created - OpenGL ES. It a subset of regular specification designed to run efficiently on devices with limited hardware resources. The most frequently used and the most useful functionality of OpenGL was included in OpenGL ES. That was desired as larger API requires more complex and larger driver to support it. It's worth mentioning that OpenGL ES is the only officially supported 3D API for portable devices running under Android and iOS operating systems.

Figure 3.1 shows diagram of OpenGL 3.3 pipeline. Three stages are fully programmable - vertex shader, geometry shader and fragment shader and those will be described in more details in following subsections. Also some of more important features of OpenGL used in implementing visualization algorithms will be described.

3.1.1 Vertex Shader

Input to the vertex shader program is a single vertex (vertices are processed in parallel) with optional attributes (which can vary between vertices - i.e. normal vector, color, texture coordinate) and uniforms (which have the same value for all vertices - i.e. projection matrix, model matrix). This stage typically transforms vertices from model space to view space and applies perspective projection. Other typical applications are calculating perspective lighting and texturing calculations. More advanced operations includes procedural animation by modifying the position of the vertex. This can be used to animate water surfaces (like pond or oceans), clothes or skin. On modern GPUs the vertex shader has also access to texture data - especially useful when using textures as data-structures in physical simulations (see (10, pages 412-419)).

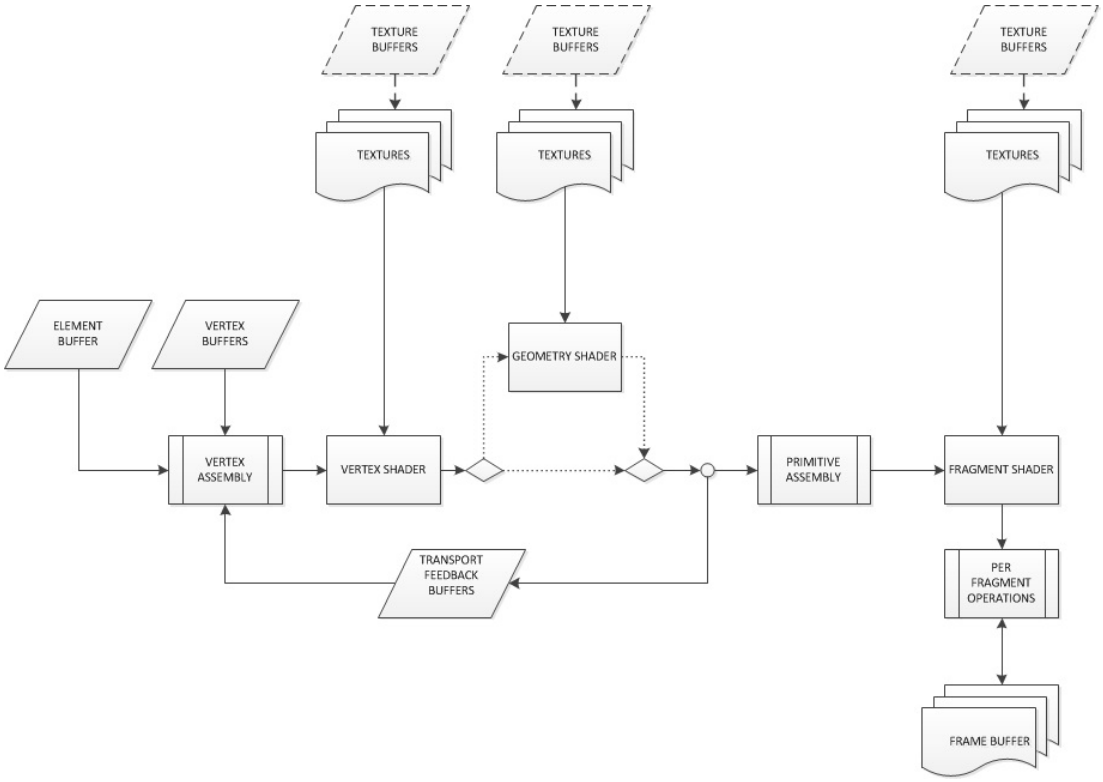


Figure 3.1: OpenGL 3.3 pipeline - taken from (10, chapter 12)

3.1.2 Geometry Shader

This stage takes entire primitives (triangles, lines, points) as an input. The geometry shader can produce new primitives, discard existing or change their type. Thus it is capable of changing amount of data in pipeline, which is unique in contrast to vertex and fragment shaders (fragment shaders can only discard fragments). There are many examples of using geometry shader - from simple as rendering six faces of a cube map (11), drawing vertices normals (10, pages 434-437) to more advanced as shadow volume extrusion (11, section 10.3.3.1), isosurface extraction (12) and dynamic tessellation.

3.1.3 Fragment Shader

This stage operates on fragments. One fragment is usually one pixel on the screen, except cases when multisampling or supersamplings are used - then several fragments may contribute to one pixel. The fragment shader's job is to process pixels by computing per pixel lighting or applying textures. Input to this stage are per-fragment attributes passed from vertex or geometry shaders, uniform attributes which are passed before rendering and which have the same value for all fragments and textures. As vertex and geometry shaders operates on vertices, per-fragment attributes passed to fragment shader have to be interpolated. There are two types of interpolation flat and smooth. Flat means that all fragments in the primitive has the same value of attribute. Smooth means that values are interpolated between edges of primitive.

As it operates at pixel level it is perfect for post processing effects (the one added on rendered scene) like blurring, bloom, different sort of filtering. In my project I make heavy use of fragment shaders.

3.1.4 Perspective projection and homogeneous space

Perspective projection mimics the kind of image produced by a typical camera by providing effect known as perspective foreshortening. With this effect the farther object lies from camera the smaller it appears on the screen. Perspective projection transforms points form view space into homogeneous clip space. In OpenGL view space uses right handed coordinate system (figure 3.2) and homogeneous space uses left handed coordinate system (figure 3.3). The region that is visible from camera is called the view volume or view frustum (figure 3.2). It is defined by six planes - near plane, far

plane and four side planes. Only objects inside view volume appears on the screen after rendering. Objects are projected on near plane, part of which lies inside four side planes is a virtual screen. Virtual screen left, right, bottom and top edges lies at $x = l, x = r, y = b, andy = t$. It's z coordinate is equal to -n. Those parameters are used to computed projection matrix $M_{V \to H}$ (equation 3.1) ($V \to H$ means transformation from view space into homogeneous space):

$$M_{V \to H} = \begin{bmatrix} \frac{2n}{w} & 0 & 0 & 0 \\ 0 & \frac{2h}{n} & & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{3.1}$$

where $w = r - l$ and $h = t - b$.

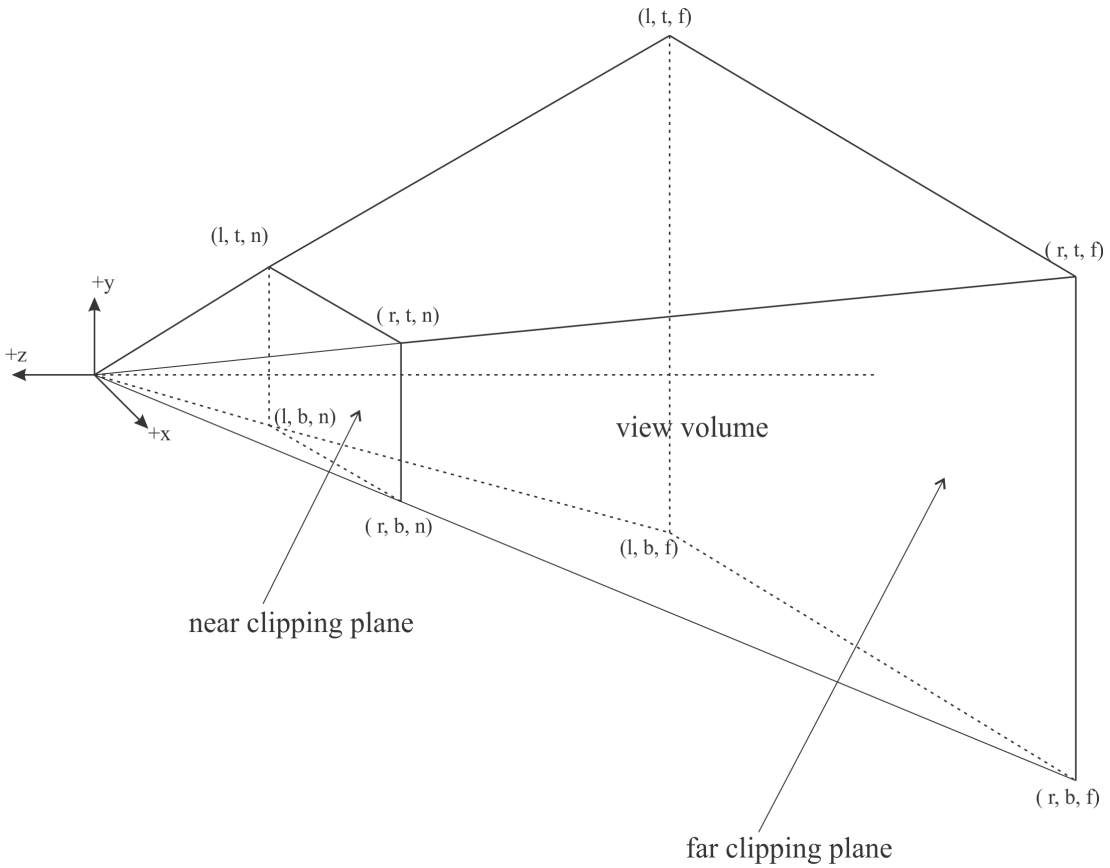


Figure 3.2: A perspective view volume (frustum)

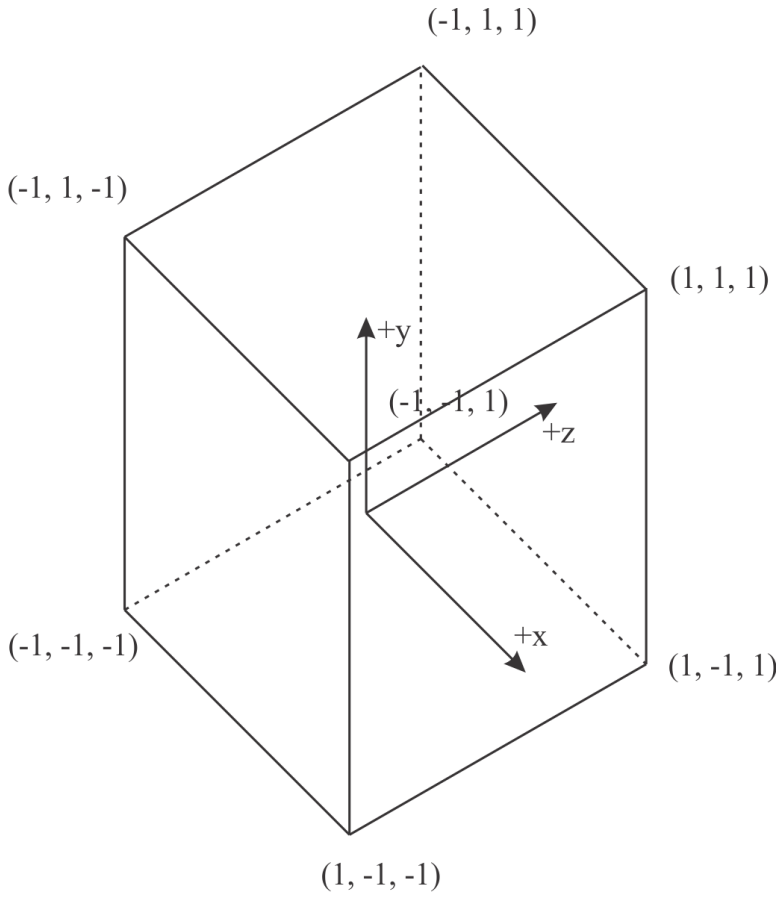


Figure 3.3: The canonical view volume in homogeneous clip space

3.1.5 Depth buffer

Dept buffer (also called z-buffer) is used to determine which fragments occlude other fragments - thus which should be drawn on the screen. This is done by comparing values of z component of fragments from homogeneous space. After multiplying point from view space - $p_V = [x, y, z, 1]$, by $M_{V \to H}$ we get point p_H:

$$p_H = \begin{bmatrix} \frac{2n}{w} & 0 & 0 & 0 \\ 0 & \frac{2h}{n} & & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} * [x, y, z, 1]^T \tag{3.2}$$

thus we get:

$$p_H = [\frac{2n}{w}x, \frac{2n}{h}y, -\frac{f+n}{f-n}z - \frac{2fn}{f-n}, -z] \tag{3.3}$$

To transform p_H into 3D space we need to divide x, y and z components by ω component, which is equal to $-z$. That gives us:

$$p_H = [-\frac{2nx}{wz}, -\frac{2ny}{hz}, \frac{f+n}{f-n} + \frac{2fn}{z(f-n)}] \tag{3.4}$$

After transformation, z component of points inside view frustum varies from [-1, 1]. This has to be transform to [0, 1], thus value of depth buffer d(z) as follows:

$$d(z) = \frac{\frac{f+n}{f-n} + \frac{2fn}{z(f-n)} + 1}{2} \tag{3.5}$$

Equation 3.5 shows that value of depth buffer is a nonlinear function of z component of the point. Figure 3.5 shows how this value changes with changing near plane distance from the viewer. It can also be seen that precision is not evenly distributed and it decreases with the distance from near plane. This can lead to so called z-fighting which occurs when two objects have close z-buffer values. When the objects are close to the viewer z-values are distinguishable, however when objects get further, values can become equal due to lack of precision (see 3.4).

Z-fighting can be avoided when reasonable values are chosen for far and near planes. However there are cases when evenly distributed values are required (see section 4.1.2.1). Alternative technique to Z-buffering is W-buffering which offers better distribution of depth values (11). Some hardware supports W-buffering but most of it doesn't and it doesn't seem to be supported in future. A workaround to this problem was presented

in (13). It customizes projection transformation in vertex shader in a way that depth buffer values have linear distribution at the end.

Linear depth value can also be obtained from equation 3.4 by dividing ω component of vector p_H by f:

$$d_{lin}(z) = \frac{-z}{f} \tag{3.6}$$

since $-z$ is in range [n, f] value of $d_{lin}(z)$ will be in range $[\frac{n}{f}, 1]$.

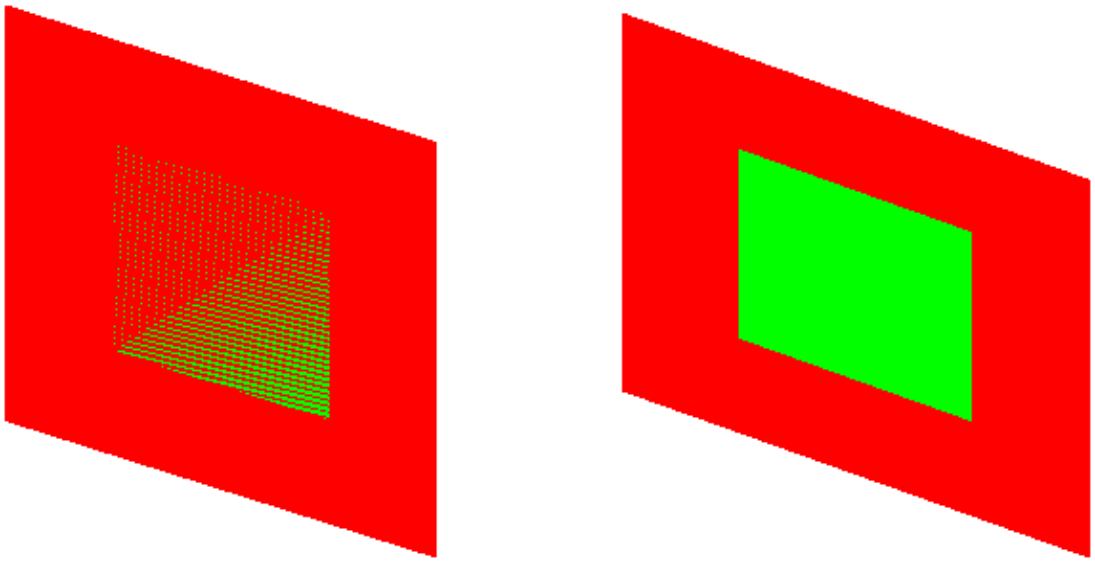


Figure 3.4: Z-fighting - two polygons rendered - on the left when z-fighting occurs, on the right the same polygons rendered correctly.

3.1.6 Framebuffer Object

Normally after performing drawing operations, output is directed to framebuffers provided by windowing system. From those buffers data can be displayed on the computer screen. However, sometimes it's not desired for data to be displayed on the screen but to be used as an input to the rendering pipeline again. This technique is called off screen rendering. It can be achieved by copying data from default framebuffers to textures and than used as an input to the pipeline again but this requires additional copy operation that should be avoided. Framebuffer objects are mechanism for dealing with situation when data shouldn't be send directly to the display. Frame buffer objects are containers that can hold other object that have memory storage and can be rendered

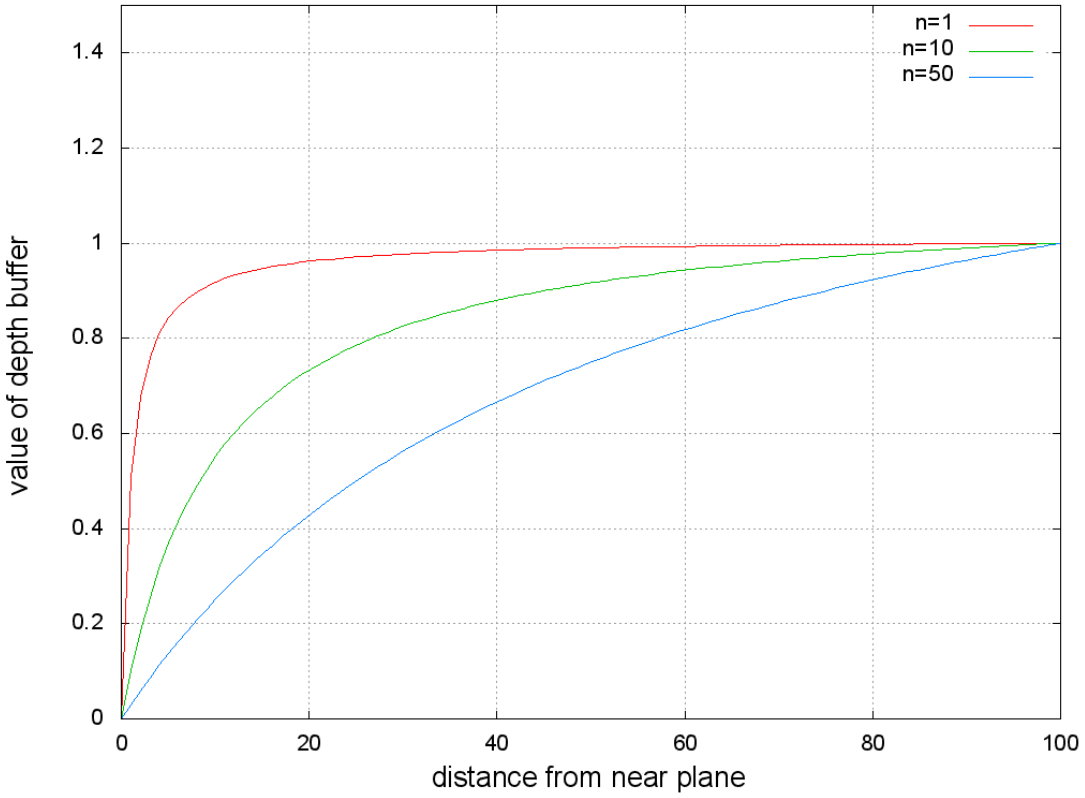


Figure 3.5: Value of z buffer as a function of distance from near plane for different values of n. $f - n$ is kept constant at 100

to (10, chapter 8). Two kinds of objects that can be rendered to are renderbuffer objects and textures. Rendering to textures is particularly useful as data doesn't have to be copied to be used as an input to the pipeline. This allows to efficiently implement rendering techniques that requires multiple phases. Unlike default framebuffer object, the one created by user is not limited in size to the size of rendering window. This can be used to speedup some computationally expensive processing by rendering with lower resolution into texture and than applying this texture to the original window with linear filtering.

Details about framebuffer objects, their creation and attaching storage to them can be found in (10, chapter 8) and (14, chapter 10).

3.2 Boost library

Boost is a set of libraries extending capabilities of C++ language. Libraries provides classes and functions for most common task and algorithms, like regular expressions, hash maps, threading. They are also cross platform.

The advantage of boost libraries is that they are designed for maximum flexibility and speed. They make heavy use of C++ template programming to be as general as possible. The other advantage of boost libraries is that they often become included in C++ standard library - like regular expressions (see (15)) .

On the other hand relying on templates makes compiling times longer. Template classes and functions must be defined entirely in header files what makes it impossible to compile them into object files. Thus whenever file has to be recompiled template classes have to be generated again.

Boost libraries used in my project are *boost regex* and *boost threads*. The first one is used to process configuration files. *Boost threads* library is used to parallelize isosurface extraction algorithm presented in section 4.2.

3.3 PhysX

PhysX is a framework for performing physical simulations in real time. It's designed to be used in computer games and is optimized for performance. It provides most necessary models required in games: rigid bodies, soft bodies, cloths, fluids and joints. What is

more it allows simulations to be performed on GPU which is much faster especially when large amounts of object acts in a simulation.

The main drawback is that hardware acceleration can only be performed on Nvidia's GPUs.

PhysX works in asynchronous way - that means simulation step is computed either on GPU or CPU thread and other CPU cores can perform some other work in the meantime - for example render scene using data from previous step. The asynchronous workflow of PhysX is presented on figure 3.6.

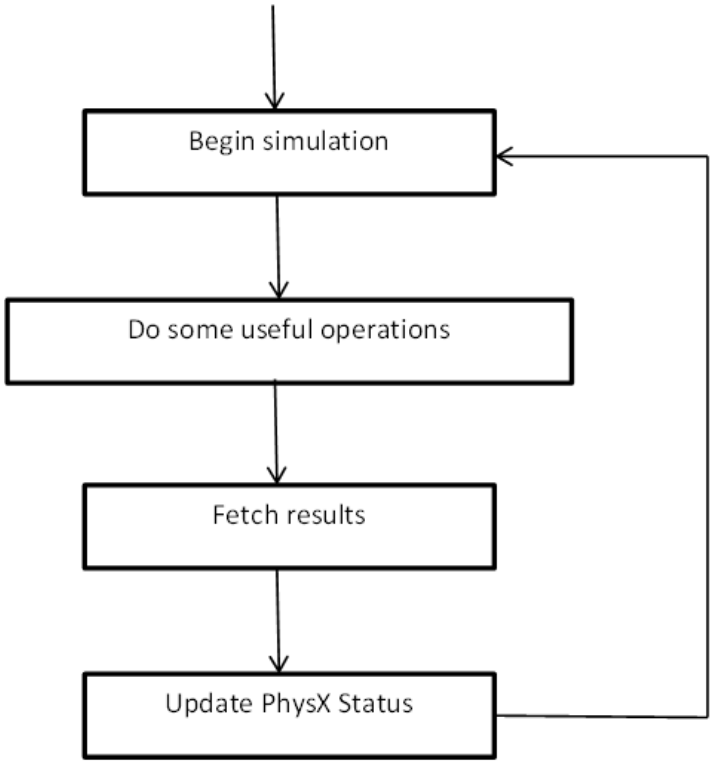


Figure 3.6: Diagram of PhysX SDK workflow

My project uses two PhysX features - rigid body and water simulations and thus those will be described in more details.

3.3.1 Rigid bodies

Basic entity of simulation is an actor. Actors can be either static objects, fixed in space (walls, ground, etc.) or dynamic rigid bodies. Figure 3.7 shows different ways the

same actors can be represented. Actors have shapes assigned to them which are used by collision detection system. This shape can be the same mesh as used by rendering system, however it is much more efficient to approximate it by set of bounding volumes (figure 3.7). Bounding volumes are simple shapes for which collision detection is easy and fast - usually rectangles and spheres.

For physics computation, actor is represented as a tensor located at the center of mass (figure 3.7). In case of collision with other object contact points are provided as points in space with normal vector of collision planes (red arrows on figure 3.7). Center of mass is automatically computed from actor's shape, however there is a possibility to set it manually.

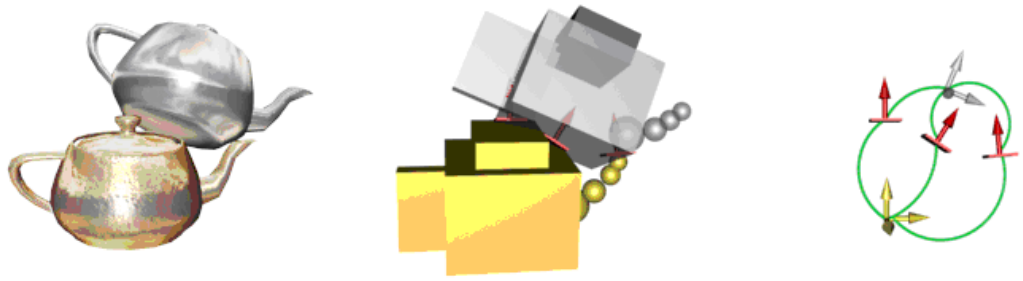


Figure 3.7: Different ways of representing actors - From left: graphical representation for rendering engine, bounding volumes used by collision detection and finally representation used for dynamics.

Important property of actor is its frame. It defines actor local coordinates system. Actor's frame is given as a matrix which transforms points from actor's local coordinates into world coordinates. All actor's shapes positions and orientations are given relatively to actor's frame.

Body movement can be a combination of rectilinear (along path) and angular (around center of mass) motions. Dynamic actors have set of properties used for rigid body simulation. All properties have a linear and angular version. They are gathered in table 3.1

Physical properties presented in table 3.1 controls how body reacts when force or torque is applied. Actors behavior after collision is determined by theirs material properties. Those parameters are: restitution, static friction and dynamic friction. Normally

Table 3.1: Comparision of sphere rendering methods

Linear property	Angular property
mass	**interia** - mass distribution of the body along its three dimensions
position - center of mass position of rigid body	**orientation** - 3x3 matrix representing principal moments relative to the actor's frame
velocity - linear velocity vector	**angular velocity** - vector representing axis of rotation which length is equal to magnitude of angular velocity
force - vector representing force acting on body	**torque** - angular force acting on a body represented as axis of rotation with length equal to magnitude of angular velocity

material is applied to entire actor's shape, however there is a possibility to apply materials per triangle. Additionally friction parameters can be anisotropic.

3.3.2 Fluids

PhysX provides three methods for simulating fluids: SPH, simple method and mixed method. First one was described in chapter 2. Second one does not take interparticle forces into consideration and thus simplifies computations. This simplified method is good for simulating phenomenas such as smoke or fog, but for realistic water SPH must be used. Third one - mixed method, according to documentation (16), alternates between SPH and simple mode, providing more performance than SPH but still maintaining some dense characteristics.

Implementation details of PhysX's SPH method are covered in (17). It solves conservation of momentum equation 2.10 to move particles. External force includes only gravity. Symmetrization of pressure and viscosity forces are done as in equations 2.12 and 2.15. Surface tension (see 2.16) is also modeled, however there are no implementation details given on that.

There are two ways of creating fluids in PhysX - either directly specifying particles initial positions, velocities, etc. or by using emitters (figure 3.8). Emitters are objects that generates particles. They can be static objects or they may be attached to a rigid

body and interact with other objects in the scene. Emitters can generate particles in two modes:

- *Constant pressure* - in which state of surrounding fluid is taken into account. Emitter tries to match rest spacing of particles (which is one of fluid's parameters).
- *Constant flow* - emitter keeps generating the same number of particles each frame.

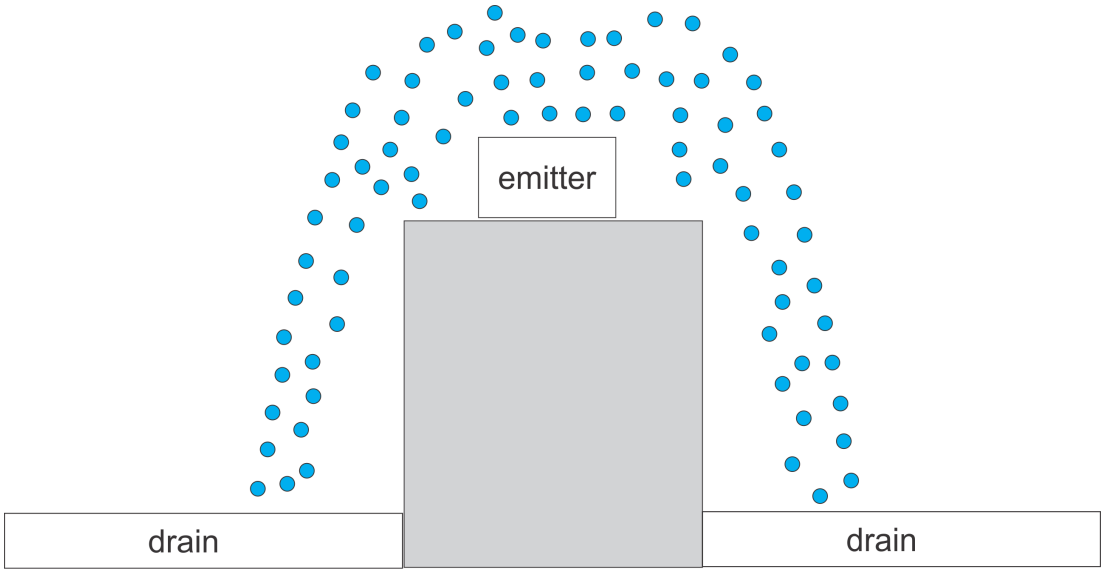


Figure 3.8: PhysX emitters and drains - Simple 2D scene with one emitter and two drains.

Removing particles can be done in three ways:

- By removing particles manually.
- By setting particles lifetime parameter which is time in seconds an emitted particle lives.
- By creating drain objects (figure 3.8). When particle hits drain it is removed from simulation. Those object are useful to prevent fluid from spreading too far and negatively affecting performance.

Fluids can interact with other objects in the scene. The interaction is handled by-passing conservation of momentum equation 2.10 by applying velocity changes directly

to particles. Interaction with rigid bodies can be either one-way (rigid body acts on fluid) or two-way (when fluid also acts on rigid body).

Most important fluid parameters that can be set are:

Kernel Radius Multiplier Corresponds to k parameter of SPH smoothing kernel. Together with *Rest Particle Per Meter* defines particle's radius of influence (area within smoothing kernel is non-zero): $radius = \frac{kernelRadiusMultiplier}{restParticlesPerMeter}$

Rest Particles Per Meter Defines number of particles per volume unit when fluid is in rest state.

Rest Density Corresponds to ρ_0 from equation 2.13. Defines density of fluid in it's rest state.

Viscosity Corresponds to μ constant in equation 2.10. Lower viscosity will produce more runny fluids like water, higher viscosity will produce more sticky fluids like honey.

Stiffness Corresponds to k constant in equation 2.13. Defines how compressible the fluid is. Higher values means less compressible fluids (for instance water).

Surface Tension Equivalent of σ from equation 2.16. The higher this parameter is the smoother fluid surface will be.

PhysX allows creating different fluids with different parameters and simulating them in the same time. However those fluids can't interact. That means it is not possible to simulate mixing two fluids together. Another limitation of PhysX's fluids is limit of 64000 particles per fluid.

4

Rendering techniques

Output from particle base simulations is a list of particles containing positions. It can also contain additional parameters - for example PhysX fluid simulation returns velocity, lifetime and density, in addition to position, for each particle.

Traditionally triangle meshes are used for rendering objects. Using this approach requires constructing fluid's triangle mesh for given particle positions. This method is called isosurface extraction and will be described in section 4.2. Another possibility is to render each particle as a point (or a billboard in general) with opacity so when large amount of particles is rendered the result would give illusion of a smooth surface. Such technique has one major drawback - it does not produces surface preventing us from adding effects of reflection and refraction. Similar technique is to extract visible surface by rendering each particle as a sphere into depth buffer. This will be described in section 4.1.

4.1 Screen space

As mentioned before this approach extracts visible surface by rendering each particle into depth buffer. High level overview ot this method is presented on figure 4.1 and consists of following steps (19): rendering depth texture (section 4.1.1) and thickness textures (section 4.1.3), depth smoothing (section 4.1.2) and assembling fluid surface with rest of the scene (section 4.1.4).

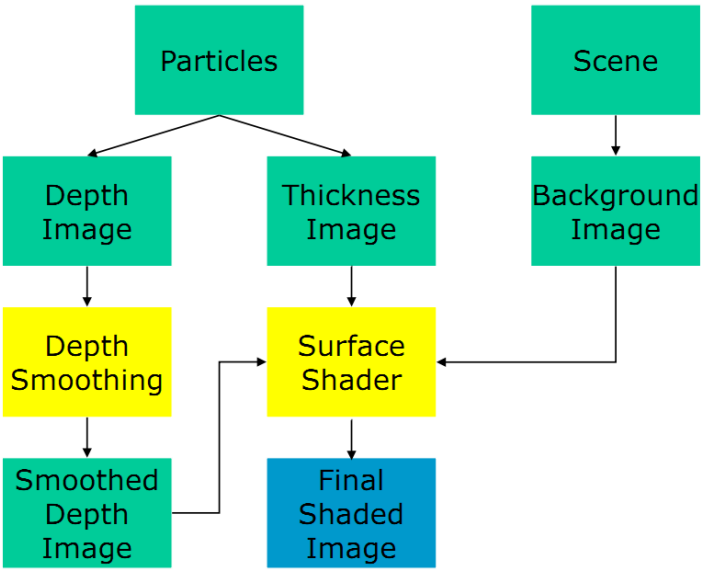


Figure 4.1: Screen Space Fluid Rendering - The figure shows high level overview of the method, taken from (18)

4.1.1 Surface depth

To obtain fluid's surface visible from the viewpoint of camera each particle is rendered as a sphere into depth texture (figure 4.2). At each pixel only closest value is kept using hardware depth test. In order to avoid rendering large amount of geometry each

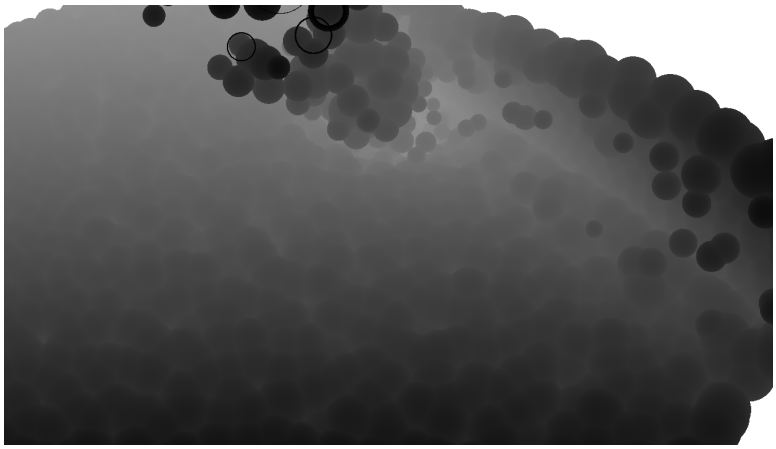


Figure 4.2: Particles rendered as spheres into depth texture -

particle is rendered as a point sprite and it's depth is generated in fragment shader. This common technique speeds up rendering process significantly as well as improves

quality of rendered spheres (as can be seen on figure 4.3). Rendering spheres as point sprites is 10 to 100 times faster comparing to rendering them as triangle meshes (see table 4.1).

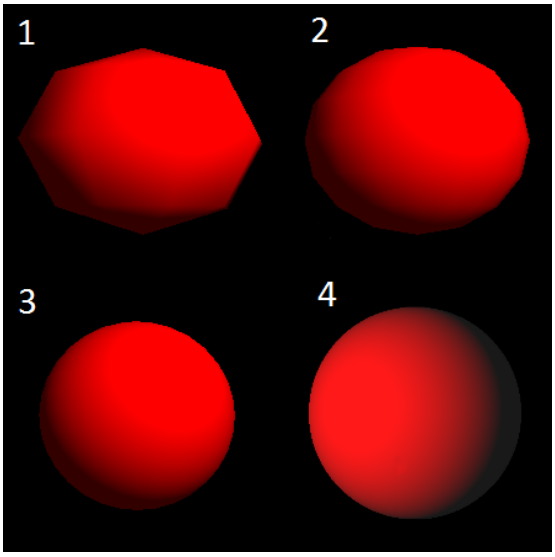


Figure 4.3: Spheres rendered with different methods - 1 - mesh with 128 triangles, 2 - mesh with 512 triangles, 3 - mesh with 2048 triangles, 4 - point sprite with normals generated in fragment shader

Table 4.1: Comparison of sphere rendering methods

Method	Frames per second
Mesh, 128 triangles	20
Mesh, 512 triangles	6
Mesh, 2048 triangles	1
Point Sprites	200

4.1.2 Smoothing

Although previous step produces surface, its quality is not sufficient. As can be seen on figure 4.4 individual spheres can be seen giving fluid's surface unnatural appearance. To remove this artifact some kind of smoothing has to be applied. Section 4.1.2.1 will describe Gaussian smoothing and section 4.1.2.2 curvature flow smoothing.

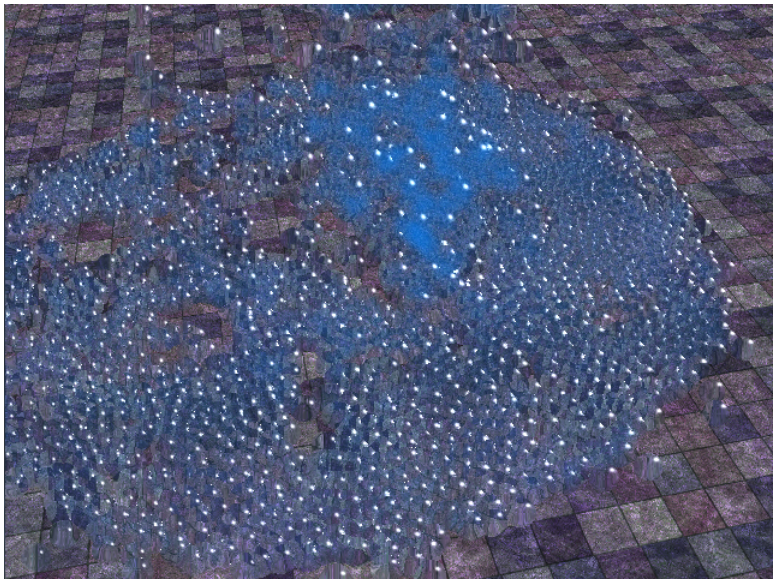


Figure 4.4: Fluid surface rendered without smoothing phase - Individual spheres can be seen, giving surface unnatural appearance

4.1.2.1 Gaussian smoothing

Most obvious way to smooth values in depth texture is to apply Gaussian filter. It's easy to implement and can be computed fast due to its linear separability. However this filter produces undesired effect of blending drops of fluid with background surfaces (see figure 4.5). Thus edge-preserving filters (also called bilateral filters) needs to be used. Bilateral Gaussian filter is a modification that changes wages of pixels depending on difference between their tonal value $I(s)$ and tonal value of central pixel $I(s_0)$. This can be described by following formula (from (20)):

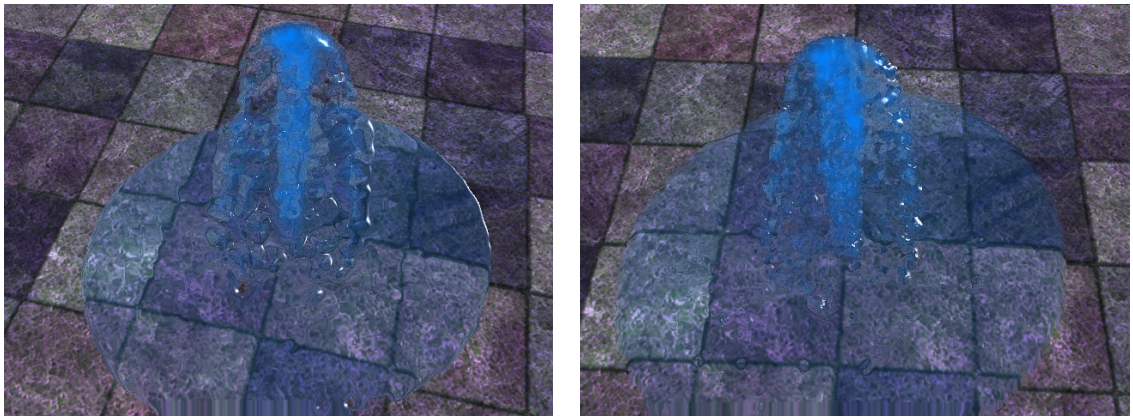
$$O(s_0) = \frac{\sum_{s \in S} f(s, s_0) I(s)}{\sum_{s \in S} f(s, s_0)} \tag{4.1}$$

where

$$f(s, s_0) = g_s(s - s_0) g_t(I(s) - I(s_0)) \tag{4.2}$$

is the bilateral filter for the neighborhood around s_0, g_s is a spatial weight, g_t is a tonal weight and they both are Gaussian functions:

$$g_s(s) = g(x, \sigma_S) g(y, \sigma_S) \quad g_t(I) = g(I, \sigma_t) \tag{4.3}$$

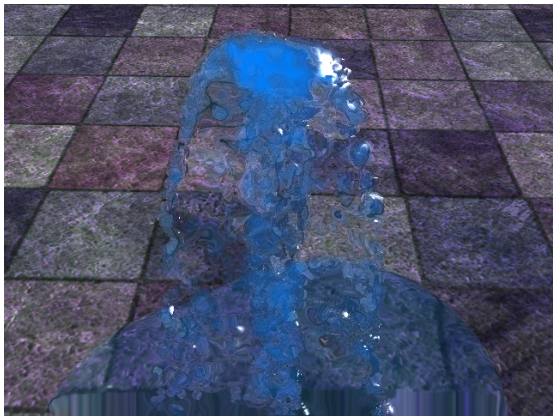


(a) Gaussian smoothing (b) Bilateral Gaussian smoothing

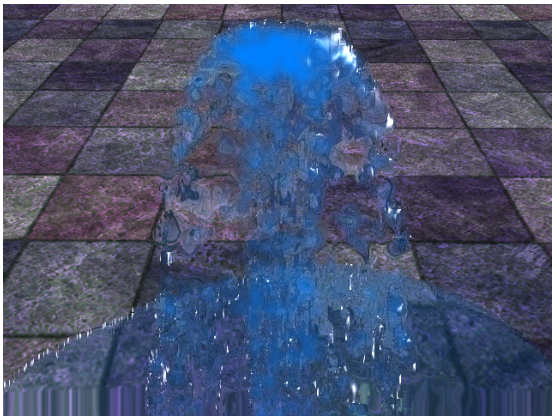
Figure 4.5: Bilateral smoothing vs regular smoothing - (a) Regular Gaussian smoothing it can be seen that surface of fountain is blended with fluid that lies on the ground, (b) bilateral filtering does not produce this undesired effect.

As can be seen in equation 4.2 only change in bilateral filter (in comparison to regular bilateral filter) is introduction of tonal weight g_t. This change makes filter space-variant - that means it can't be computed as a product of two one-dimensional filters (g_s in equation 4.3). Computational complexity of space-variant filters is $O(Nm^d)$ comparing to only $O(Nmd)$ for space-invariant (d is image dimensionality, m is the size of filtering kernel and N is the number of pixels in the image). For smoothing fluid surface kernel with $m = 20$ must be used, which, when using bilateral filtering on 2 dimensional image, gives about 400 operations for each pixel. In comparison separable implementation requires only 40 operations. It turns out however that computing bilateral filter as if it was space-invariant gives good approximation for real time applications (see figure 4.6). Approximation gives some artifacts but they are not visible when fluid is moving and other effects (reflection and refraction) are applied.

Bilateral filter has some undesired effect when applied on z-buffer output. The problem is that depth values are not evenly distributed. This means that difference between two fragments depth values is dependent on their distance to the viewer. As can be seen on equation 4.2 bilateral weights takes into account difference between depth values. As a result the further surface lies from camera the less edges are preserved (see figure 4.7). In that situation linear depth values have to be generated in fragment shader as described in section 3.1.5. To produce linearly distributed depth values I am using



(a) Bilateral Gaussian smoothing



(b) Bilateral Gaussian smoothing computed as if it was separable

Figure 4.6: Comparision of normal bilateral filter with it's separable approximation - Computing bilateral Gaussian filter as if it was separable produces some artifacts on edges (b), however they are not very disturbing.

equation 3.6.

Another issue is that when using fixed kernel size filter. As particles are rendered smaller with the distance to the viewer further surfaces will be more smooth than closer surfaces (see figure 4.7). Solving that problem requires using filter with variable kernel size, depending on fragment distance from viewer. This is implemented using several precomputed kernels which are passed as array of 1D textures into shader. Section 4.1.2.2 will describe another smoothing technique that doesn't suffer from this problem.

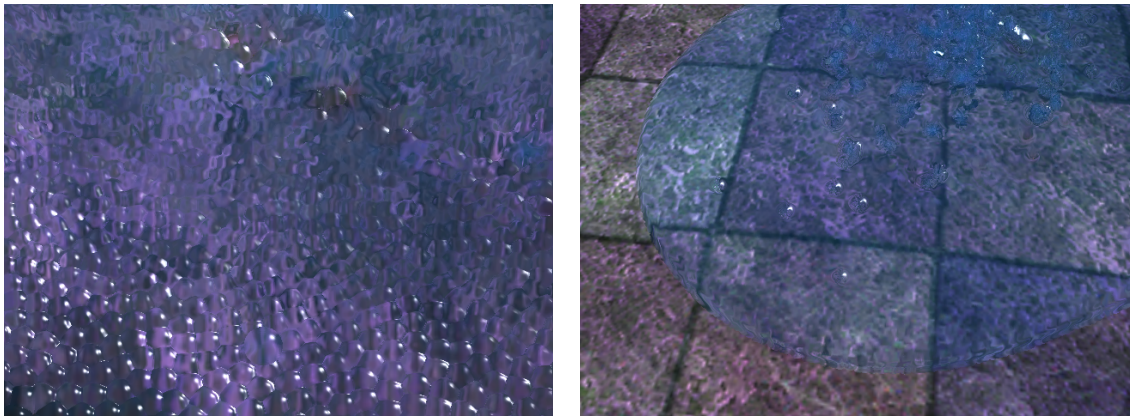
4.1.2.2 Curvature flow smoothing

This technique was presented in (19). It utilizes curvature flow to smooth surface. Curvature is a differential geometry concept which measures how geometric object deviates from being flat. Curvature flow evolves surface by moving it's points with velocity given by:

$$\vec{v_p} = -\kappa_p \hat{n}_p \tag{4.4}$$

where κ_p is mean curvature at point p and $\hat{n}_p$ is unit normal vector at point p. We can then write equation of motion under mean curvature for surface u:

$$\frac{du}{dt} = -\kappa\vec{n} \tag{4.5}$$



(a) Closer to camera (b) Farther from camera

Figure 4.7: Bilateral filtering results for different distances to camera - The more distant fluid is from viewer the more smoothed its surface is. Also less edges are preserved due to z-buffer used resulting in more particles got blended into background surfaces.

Since we are working with depth texture ($z(x, y)$) we only move surface in direction of z axis. This allows to omit surface normal:

$$\frac{dz}{dt} = -\kappa \tag{4.6}$$

In depth buffer, areas with highest curvature occurs in contact places of particles (see figure 4.8). Those are also areas that requires the most smoothing. Boundary conditions have to be enforced in equation 4.5 to ensure that different patches of fluid won't be blended together (see figure 4.8).

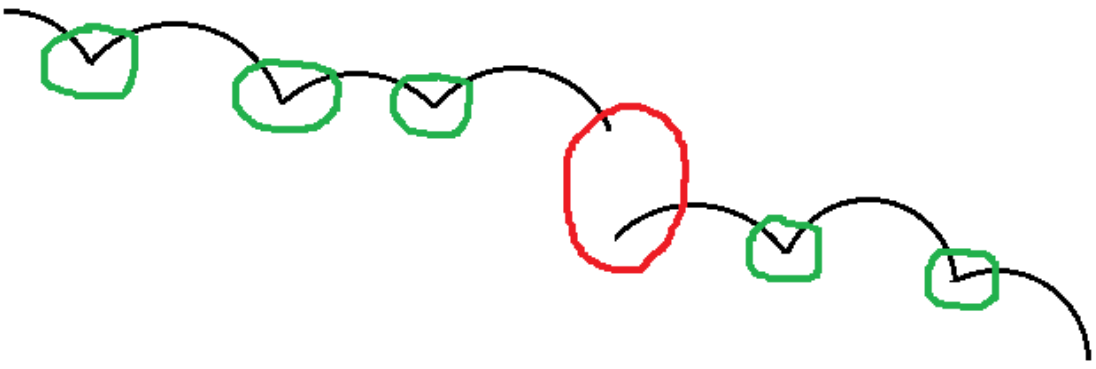


Figure 4.8: Depth texture in 2D - Areas with highest curvature marked by green circles. In red circle there is area that shouldn't be smoothed.

Mean curvature is defined as:

$$\kappa = \frac{1}{2}\dot{\nabla n} \tag{4.7}$$

To compute surface normal from depth texture we need to invert perspective projection to obtain point P:

$$P(x,y) = \begin{pmatrix} \frac{2x-1.0}{F_x} \\ \frac{2y-1.0}{F_y} \\ 1 \end{pmatrix} z(x,y) = \begin{pmatrix} W_x \\ W_y \\ 1 \end{pmatrix} z(x,y) \tag{4.8}$$

where F_x and F_y is camera's focal length in the x and y direction. To compute surface normal we have to take cross product between partial derivatives of P in x and y directions:

$$\begin{aligned} n(x,y) &= \frac{\partial P}{\partial x} \times \frac{\partial P}{\partial y} \\ &= \begin{pmatrix} C_x z + W_x \frac{\partial z}{\partial x} \\ W_y \frac{\partial z}{\partial x} \\ \frac{\partial z}{\partial x} \end{pmatrix} \times \begin{pmatrix} W_x \frac{\partial z}{\partial y} \\ C_y z + W_y \frac{\partial z}{\partial y} \\ \frac{\partial z}{\partial y} \end{pmatrix} \\ &\approx \begin{pmatrix} C_x z \\ 0 \\ \frac{\partial z}{\partial x} \end{pmatrix} \times \begin{pmatrix} 0 \\ C_y z \\ \frac{\partial z}{\partial y} \end{pmatrix} = \begin{pmatrix} -C_y \frac{\partial z}{\partial x} \\ -C_x \frac{\partial z}{\partial y} \\ -C_x C_y z \end{pmatrix} z \end{aligned} \tag{4.9}$$

where $C_x = \frac{2}{F_x}$, $C_y = \frac{2}{F_y}$. (19) suggested to discard terms of derivative of P that depend on view position W_x and W_y. Unit normal is given by:

$$\hat{n} = \frac{n(x,y)}{|n(x,y)|} = \frac{\left(-C_y \frac{\partial z}{\partial x}, -C_x \frac{\partial z}{\partial y}, -C_x C_y z\right)^T}{\sqrt{D}} \tag{4.10}$$

in which

$$D = C_y^2 \left(\frac{\partial z}{\partial x}\right)^2 + C_x^2 \left(\frac{\partial z}{\partial y}\right)^2 + C_x^2 C_y^2 z^2 \tag{4.11}$$

Now $\hat{n}$ can be substituted into equation 4.7. The z component of divergence will be 0 as $\frac{\partial z}{\partial z} = 0$. Thus we have:

$$\kappa = \frac{1}{2}\left(\frac{\partial \hat{n}}{\partial x} + \frac{\partial \hat{n}}{\partial y}\right) = \frac{C_y E_x + C_x E_y}{D^{\frac{3}{2}}} \tag{4.12}$$

in which

$$E_x = \frac{1}{2}\frac{\partial z}{\partial x}\frac{\partial D}{\partial x} - \frac{\partial^2 z}{\partial x^2} D \tag{4.13}$$

$$E_y = \frac{1}{2}\frac{\partial z}{\partial y}\frac{\partial D}{\partial y} - \frac{\partial^2 z}{\partial y^2}D \tag{4.14}$$

To modify z values in each iteration Euler integration of equation 4.6 is used. Amount of smoothing can be controlled by number of iterations - the more iterations the smoother surface will be. Since equation 4.6 is stiff, integration time step has to be very small to retain stability. If it is too large undesirable visual artifacts appears (figure 4.9). My experiments shown that time step should be less than 10^{-7}.

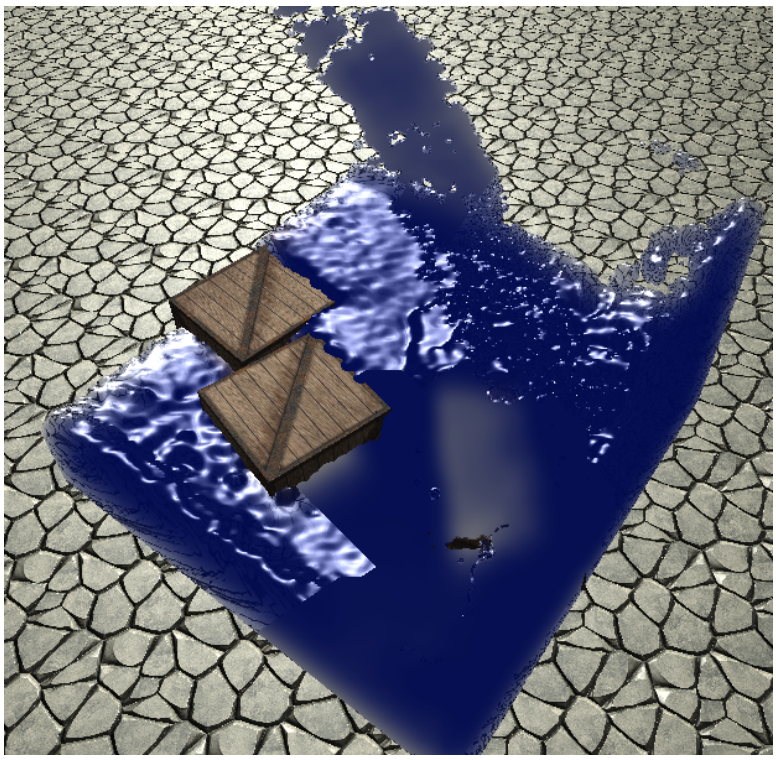


Figure 4.9: Curvature flow smoothing artifacts - artifacts occurs when integration time step is too big

4.1.3 Thickness

This step is performed to determine how opaque is surface in given point. It is done by rendering particles as circles into depth buffer with additive blending enabled. Resulting thickness texture is then smoothed with Gaussian filter (this time regular one). In order to speed up rendering process this texture can be rendered with lower resolution and then applied with linear interpolation.

4.1.4 Rendering

Last step assembles fluid surface with background image. As an input it takes smoothed fluid's depth texture, thickness texture and texture with rest of the scene rendered.

First normal vectors have to be computed using finite differences of surface depth $d(x, y)$. This is done as in equation 4.10. Finite differences are computed in one direction, except edges to avoid undesirable smoothing (figure TODO).

Shading is done using Fresnel equation and a Phong specular highlight. Output color C_{out} is given by:

$$C_{out} = a(1 - F(n \cdot v)) + bF(n \cdot v) + k_s(n \cdot h)^{\alpha} \tag{4.15}$$

where α is refracted fluid color, β is reflected color, k_s and α are constants for Phong specular highlight, n is a normal vector of the fluid surface, v is view vector (from surface to camera), h is the half vector between light vector and the view vector and $F(n \cdot v)$ is Fresnel function. $F(n \cdot v))$ is computed using Schlick's approximation:

$$F(n \cdot v)) = F_0 + (1 - R_0)(1 - n \cdot v)^5 \tag{4.16}$$

where F_0 is the value of reflectance when angle between normal vector and view vector is 0.

Refracted fluid color a is given by:

$$a = lerp(C_{fluid}, S(x + \beta n_x, y + \beta n_y), e^{-kT(x,y)}) \tag{4.17}$$

where $S(x, y)$ texture with rest of the scene, β is the parameter which controls how much background scene is refracted, C_fluid is the color of the fluid itself and k is a given constant. As can be seen thickness texture is used to control fluid opacity - the thicker fluid is the less opaque it is. Refraction is approximated by sampling underlying scene texture $S(x, y)$ with texture coordinates perturbed by normal of the surface. Amount of perturbation is controlled by β parameter which depends on thickness:

$$\beta = T(x, y)\gamma \tag{4.18}$$

where γ is a constant which depends on the kind of fluid.

Reflected color b is computed using environmental mapping.

4.2 Isosurface extraction

Classic algorithm for isosurface extraction is marching cubes (see (21)). It takes 3d array with scalar field values as an input and produces list of triangles. Algorithm divides space into cubes, and proceeds through scalar field taking one cube at a time (each cube consists of 8 vertices with scalar field values). Field value at each cube vertex is tested to be above or under given threshold and then appropriate triangle configuration is taken from lookup table (there are 256 triangle configurations).

This method is not suitable for extraction surface from particles. Firstly scalar field values at cube corners are not given. Secondly visiting each cube is not efficient as most of cubes doesn't contain surface. Several variations of marching cube algorithm were created to overcome those problems. The algorithm used is a multithreaded version of the one presented in (1). This is a surface following version of marching cubes with a fast particle lookup cache technique.

4.2.1 Overview

Algorithm takes as an input a list of particles positions, particle's radius of influence (R_c), isosurface threshold and produces list of triangles. Particle's radius of influence is something different than particle size (or simply particle radius - R), because its value is usually greater. Every particle produces a field that has non-zero values within R_c from particle center. The field value in given point of space p is a sum of field values generated by all particles within R_c. This can be represented by:

$$F(p) = \sum_{s \ \epsilon \ S_{R_c}(p)} f(dst(s,p)) \tag{4.19}$$

where s is a center of particle, $S_{R_c}(p)$ is a sphere with p as a center and radius of R_c. $f(r)$ is a field function of single particle. It can be any function that is radially monotonic, continuous and has non-zero values within R_c from particle center. Following function is often used as it can be computed efficiently using few operations:

$$f(r) = \begin{cases} (\frac{r}{\sqrt{2}R_c})^4 - (\frac{r}{\sqrt{2}R_c})^2 + 0.25 & \text{if } r < R_c \\ 0 & \text{otherwise} \end{cases} \tag{4.20}$$

When $R_c > R$ neighboring particles combines into one smooth surface (figure 4.10). This is a common technique known as metaballs (22). The biggest problem here is to

efficiently compute field values at corners and to traverse space in the most optimal way. Those topics will be described in following subsections.

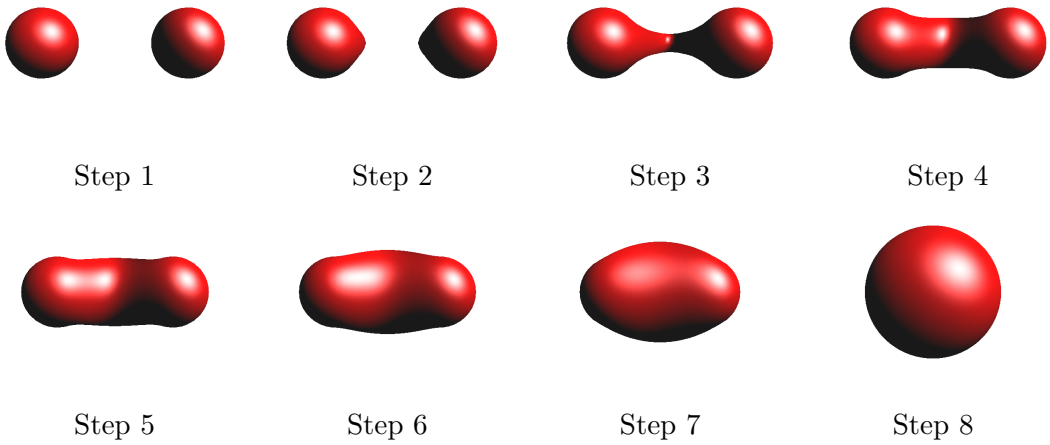


Figure 4.10: Metaballs - illustrates how particles combines into one surface as their approach closer and closer together.

4.2.2 Block subdivision

Space is divided into blocks containing cubes (figure 4.11). Such division aims for:

- Reduce memory consumption - smaller blocks contains less particles and cubes.
- Allowing easy way of parallelization of algorithm - because processing one block is independent of processing other blocks.

Blocks are expanded to contain also particles that lies outside them but may influence the field values inside block. That means the block must also contain particles lying within R_c distance from it - this area is called margin and has special treatment, which will be described in next section. Expanding blocks makes parallel processing very easy as dependencies between block disappear.

Block subdivision is a first step of the algorithm. For every particle containing blocks are determined. One particle can be contained in at most 8 blocks due to blocks overlapping. Each block has a list of particles lying within its boundary.

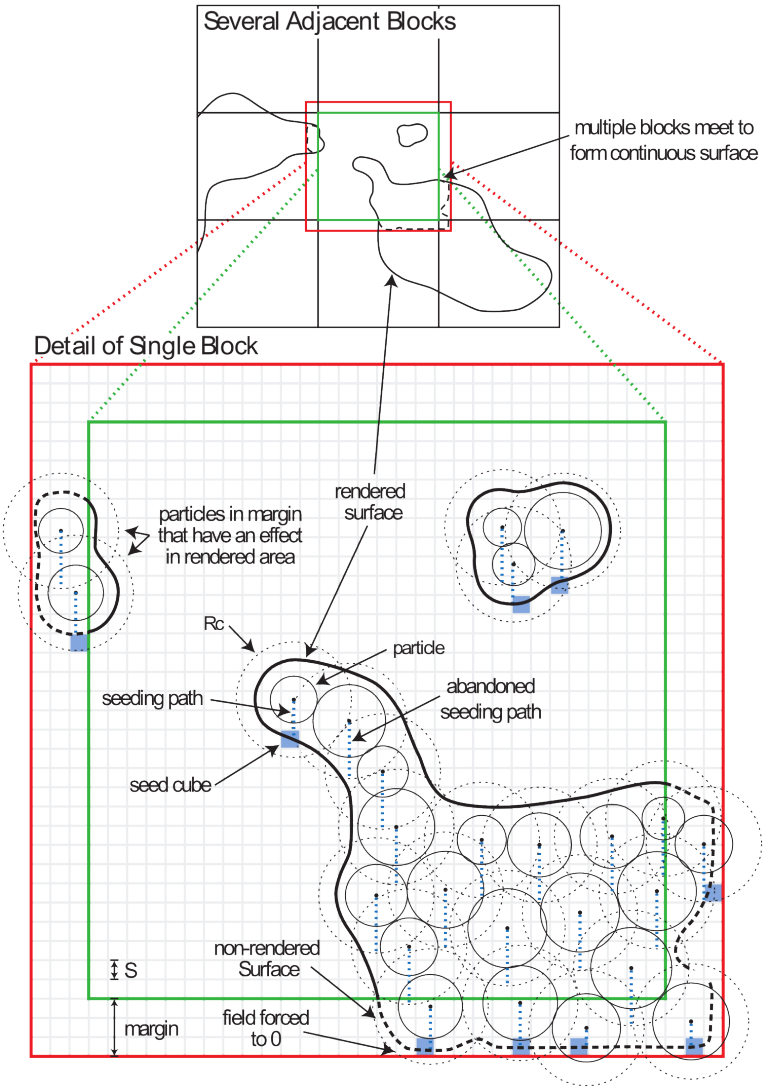


Figure 4.11: Block subdivision - space is divided into blocks which overlaps. Each block is divided into cubes of size S. Image taken from (1)

4.2.3 Block processing

Processing of block starts from building a particle lookup cache which will be described in section 4.2.4. Next, surface following marching cube algorithm is run inside block. The algorithm is named *marching slices* in (1). Block containing $n \times n \times n$ cubes is divided to n slabs and n+1 slices (figure 4.12). Division is made in parallel to XY plane. Each slab contains $n \times n \times 1$ cubes and each slice contains $(n+1) \times (n+1) \times 1$ cube corners. Slices lies between slabs and serves as a cache for field values computed at corners. In addition neighboring slabs share slices that lies between them. Each slab has a list of cubes, lying within it, that should be visited - a *todo list*.

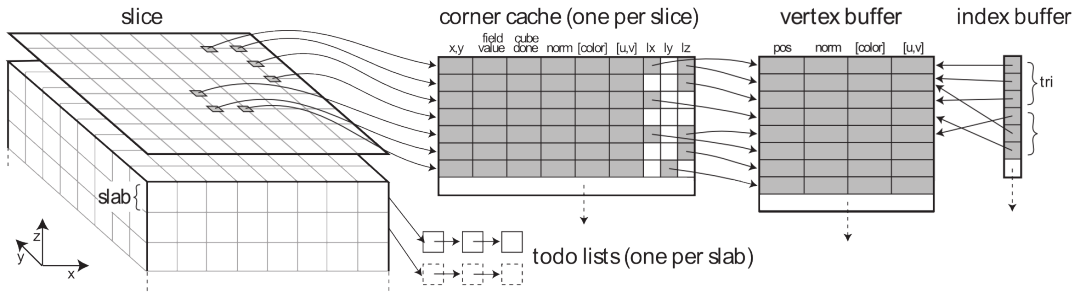


Figure 4.12: Structure of a block. Image taken from (1) -

To speed up a marching cubes algorithm we would like to traverse only cubes that contains surface. This requires finding so called *seed cubes* - initial cubes at which algorithm will start traversing block. We then choose one of the slabs that contains one or more seed cubes and pick up one cube. Field values at cube corners are calculated and put into slice caches, surface is polygonized and neighboring cubes that contain surface are added to a *todo lists* of corresponding slabs.

If we picked up a random slabs and cubes each time then we would have to keep all cached values in slices while processing a cube. The idea of marching slices is to decrease memory, that cache consumes, by traversing cubes slice by slice and deallocate caches as soon as possible. This can be done by always picking the lowest slab with not empty *todo list* and finding seeding cubes in the lowest possible slabs.

The optimum for finding seed cubes would be to find surface's all local minimas - then we would only have to store cache in slices below and above current slab. However this approach would be time consuming. (1) used heuristic approach with complexity O(P) where P is a number of particles. The algorithm is as follows:

```
for every particle:
        take a corner closest to particle center
        while F(c) > treshold and |c - c_start| < 2 do
                c = corner below c
        if F(c) < treshold
                add cube to it's slab todo list
```

This method will not find local minimas every time as it always goes down from the center of a particle (see figure 4.13).

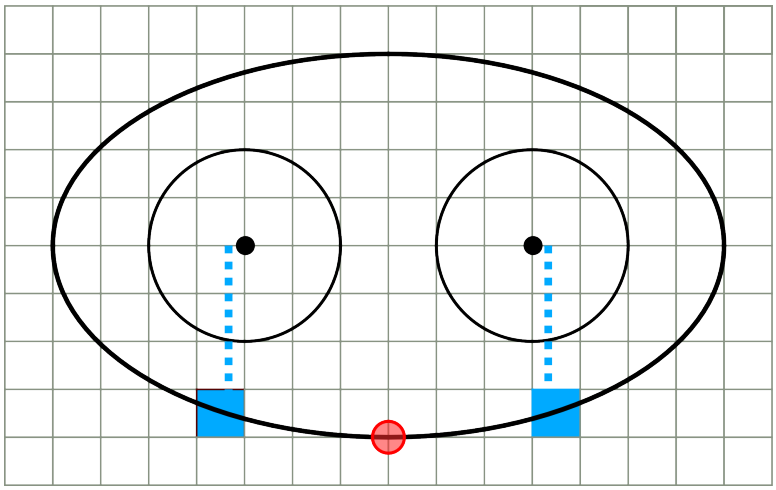


Figure 4.13: Finding local minimum - In this example algorithm fails to find local minimum. We have two particles forming metaball. Seeding cubes and paths to find them are marked blue, actual local minimum is marked in red circle

4.2.4 Lookup cache

To compute field value for a given corner all particles that lies within r_c from it has to be found. This task can be accomplished by building space partitioning data structure like octtree or kdtree (23). However costs of building spatial data structures are relatively high, and the nearest neighbor finding algorithm requires floating point computations. (1) presented another way to accomplish this task. Presented method uses only integer comparison. The main observation is that we are looking nearest particles in discrete points. Thus obvious approach would be to create a linked list for every corner containing all particles lying within r_c from it. So for every particle we would traverse all corners lying inside a sphere of radius r_c and update their particles list by

inserting the current one. The main disadvantage of this approach is that we would have to store that information for each corner, and for every corner we would have to store a separate list. This can be improved by using more sparse data structure. For a block we can store only one slice (so called projection slice) as a 2D array of $N + 1$ by $N + 1$. Element (x, y) of the array represents a column (x, y) of corners and stores a linked list of particles that have an influence in that column (figure 4.14). To make particle lookup procedure more efficient each list is sorted by particles z coordinate. Each element of the list contains three integer values - min_z, max_z, mid_z and a pointer to particle. min_z and max_z specifies range of slices in which particle has influence and mid_z is a slab that contains center of the particle.

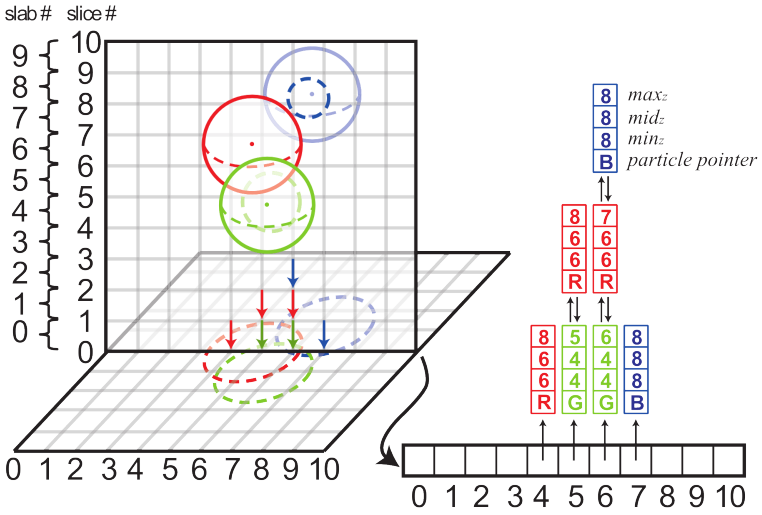


Figure 4.14: Particle lookup cache - Each cell of projection slice represents a column in block and stores linked list of particles that have influence in that column. Image taken from (1)

Initialization of cache starts with sorting all particles by z coordinate. Then we iterate over sorted particles and each particle is inserted into all columns in which it has an influence.

To lookup particles that have influence in given corner at location (x, y, z) we have to check all columns from projection slice that lies within R_c radius from that corner. This narrows search to cylinder of radius $\left\lceil \frac{R_c}{S} \right\rceil$ and height equal to height of block. Next each column is iterated from begin to search for first element (E_f) with $mid_z \geqslant z - \left\lceil \frac{R_c}{S} \right\rceil$. Then iteration is continued until topmost element (E_l) with $mid_z \leqslant z + \left\lceil \frac{R_c}{S} \right\rceil$. This

narrows height of searching cylinder to $2R_c$. In order to further narrow this into sphere of radius R_c every element between E_f and E_l is tested for having $min_z \leqslant z$ and $max_z \geqslant z$. Such a procedure return only particles that have influence in given corner.

Further discussion concerning performance and memory consumption of this lookup technique can be found in (1). Authors also provide comparison with other lookup methods.

4.2.5 Normals generation

In order to properly shade extracted surface normal vectors have to be computed for each vertex. The simplest approach is to generate one normal for each triangle, perpendicular to its surface and assign it to all three triangle vertices. This requires that vertices are not shared between triangles thus it's not applicable with surface extraction method used.

Another approach is to compute normals at corners of a grid as a gradient of scalar field:

$$\vec{n}_{i,j,k} = \begin{bmatrix} \frac{F_{i-1,j,k} - F_{i+1,j,k}}{W} \\ \frac{F_{i,j-1,k} - F_{i,j+1,k}}{H} \\ \frac{F_{i,j,k-1} - F_{i,j,k+1}}{D} \end{bmatrix} \tag{4.21}$$

where W, H and D are dimensions of cube, $\vec{n}_{i,j,k}$ is normal at corner (i, j, k) and $F_{i,j,k}$ is value of scalar field at corner (i, j, k). As values of field are given only at corners and vertices in most cases lies between them the interpolation is required. Also this increases amount of corners at which scalar field value must be computed. Another disadvantage is that in order to speed up process of normal generation in this case, normals can't be generated after extracting isosurface. This is due to clearing caches, and thus normals have to be computed in the same time vertex are added to *todo list*. This results in more complicated code.

Yet another approach is to compute normal for every triangle that includes given vertex and take an average or weighted average as a vertex normal. Simple average can be computed effectively however it produces visible artifacts for sparse grids. This is because some triangles are long and narrow and should not have so much impact on final normal. To solve this weighted average can be used. Weights can be the area of triangle or an angle by given vertex.

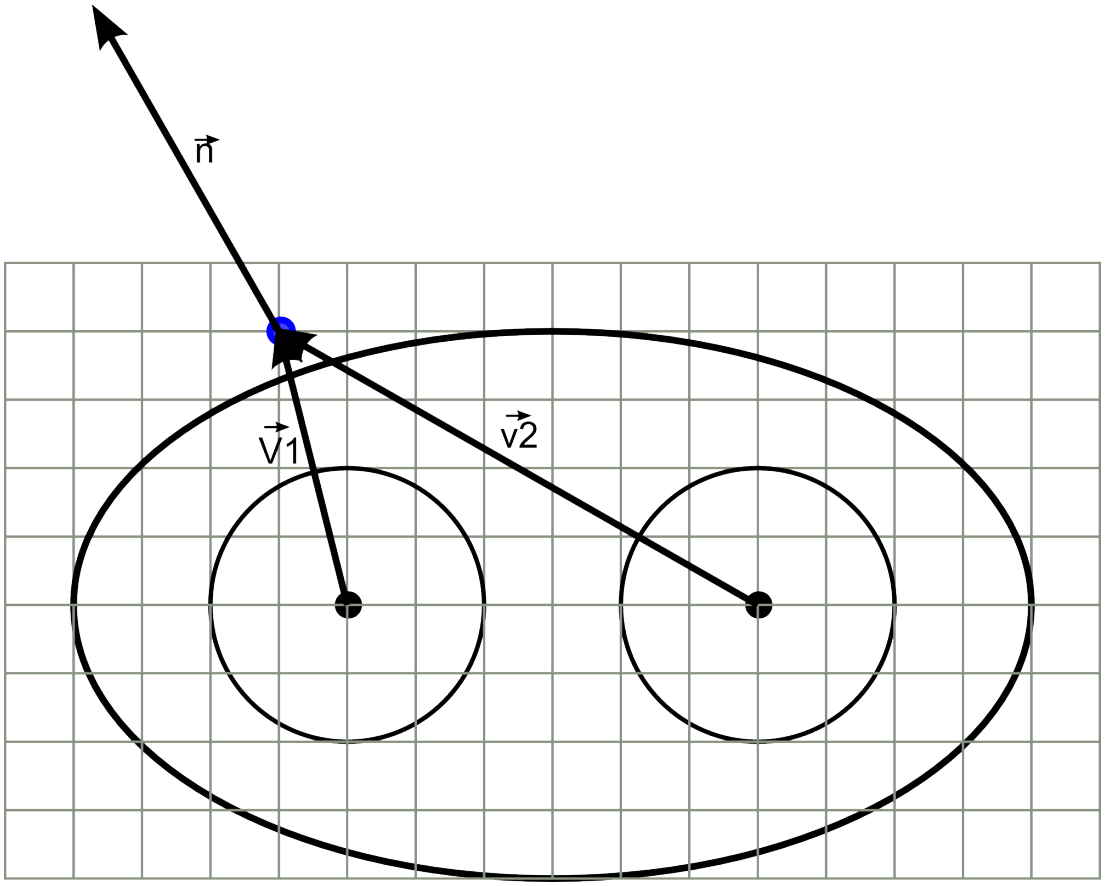
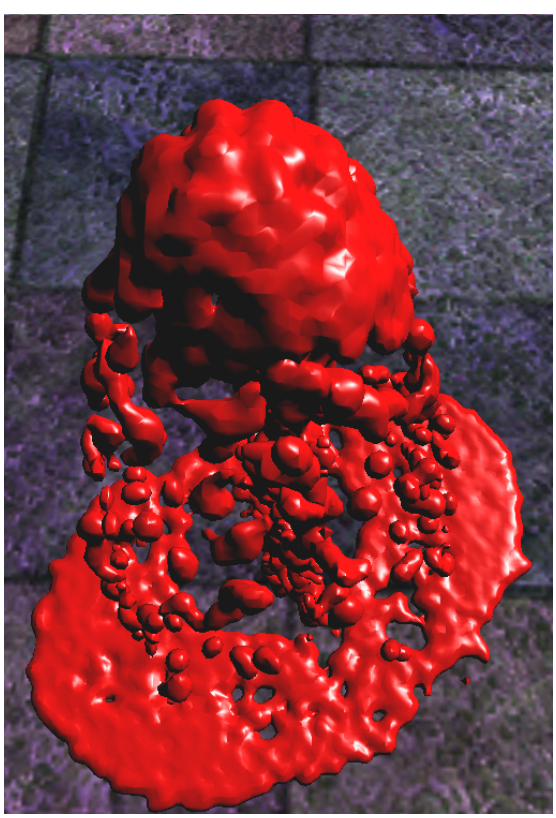
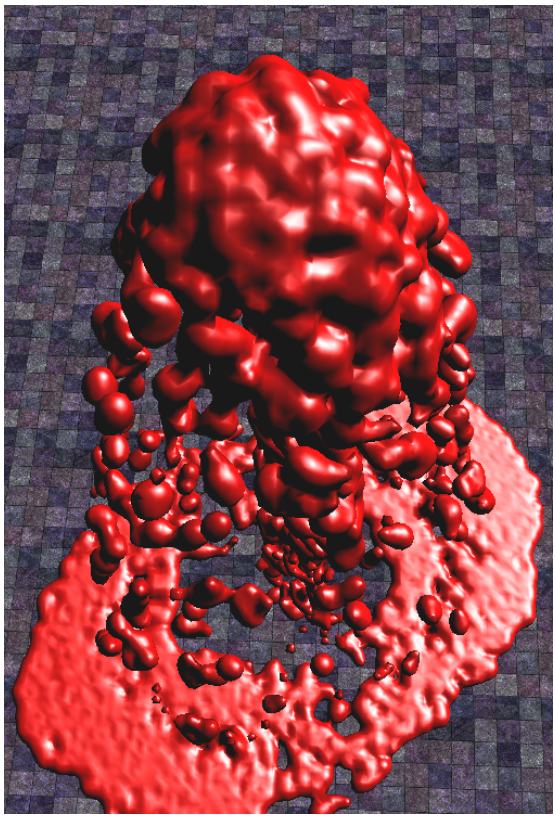


Figure 4.15: Normal vector computation for corner - $\vec{n} = f(|\vec{v1}|)\frac{\vec{v1}}{|\vec{v1}|} + f(|\vec{v2}|)\frac{\vec{v2}}{|\vec{v2}|}$

As approximating surface normal at given vertex as weighted average of adjacent triangles normals doesn't give satisfying results I have used different approach. It is illustrated on figure 4.15. For every visited corner normal vector is computed as a weighted average of vectors pointing from particle centers to this corner. The weights are field values generated by each particle in this corner. When vertex is generated between corners normal vector is computed as weighted average of corners normals. Figure 4.16 shows comparison between two last normal generation algorithms



(a) Weighted average based on length



(b) Vectors computed at each visited corner

Figure 4.16: Comparison of normal generation techniques - (a) - weighted average of normals of adjacent triangles, (b) - weighted average of normals computed for corners.

4.2.6 Multithreading

(1) presented algorithm that runs in one thread. However algorithm has a potential of parallelization due to independence of block processing. Parallelization of this algo-

rithm is even more desired as nowadays CPU manufacturers increase performance by multiplying CPU cores instead of increasing clock frequency of a single core.

As mentioned before processing of block is independent of one another. The algorithm uses thread pool which size can be configured and the single task is a single block. The architecture is illustrated on figure 4.17. The single thread checks if there are still blocks to process in blocks queue, takes one and extracts surface in this block. Result of extracting surface in one block is put into results list. Thread stops when there are no more task in queue. As the main thread doesn't take part in surface extraction there is possibility of working asynchronously. Main thread can render surface extracted in previous frame while working threads are extracting surface for the current frame.

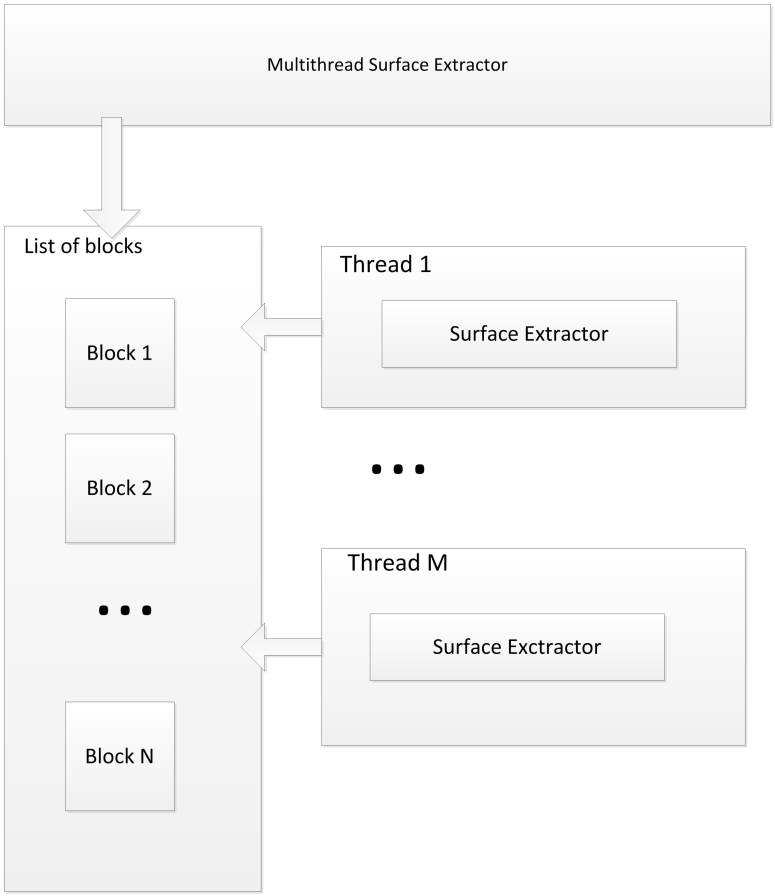


Figure 4.17: Threading architecture - architecture diagram of parallelization of surface extraction algorithm

4.2.7 Rendering

Rendering fluid as a mesh is a simpler task since we already have surface and normals. Only one problem arises when fluid has to be rendered with opacity. Traditionally it would require rendering graphic primitives in back to front order with alpha blending enabled. However algorithm used for surface extraction doesn't produce primitives in such order. (1) noted that this algorithm can be transformed to produce primitives with desired ordering. Since this modification produces some extra overhead I used different approach - similar to one used in section 4.1.4.

First background scene is rendered into texture $S(x, y)$. Then when fluid meshes are rendered this texture is sampled with texture coordinates perturbed using normals of the surface. This computation is done in fragment shader. As scene texture cannot be applied to meshes itself but it should be applied to the whole screen texture coordinates are computed in fragment shader based on screen coordinates of fragment:

$$(x, y) = (\frac{p_x}{width}, \frac{p_y}{height}) \tag{4.22}$$

where (p_x, p_y) are screen coordinates of fragment, $width$ and $height$ are dimensions of the screen.

5

Results and conclusions

5.1 Performance

Performance test were performed on two machines which configuration is presented in table 5.1. PhysX scene with one emitter was used. Results were rendered in two resolutions - 640 x 480 and 1280 x 960

Figures 5.1, 5.2 and 5.3 presents graphs of frames per second on the number of simulated particles. Maximum frame rate was restrained to 60. Those three graphs takes both simulation and rendering time into consideration. First thing to notice is that screen space techniques are much more vulnerable to increasing resolution than isosurface extraction. This is due to much more complex shader programs used, especially in curvature flow smoothing.

Isosurface extraction (figure 5.3 runs on CPU and is it much more dependent on number of particles rendered than on resolution. Rendering is smooth for less than

Table 5.1: Configuration of test machines - Machine 1 is a high performance gaming desktop, machine 2 is a middle class multimedia laptop.

	machine 1	machine 2
CPU	Intel Core i7 2600k 3.4 GHz, 4 cores	Inte Core i7 2630M 2.0 GHz, 4 cores
GPU	Nvidia GeForce GTX 560 Titanium	Nvidia GeForce GT 540M
Physical memory	8 GB	6GB

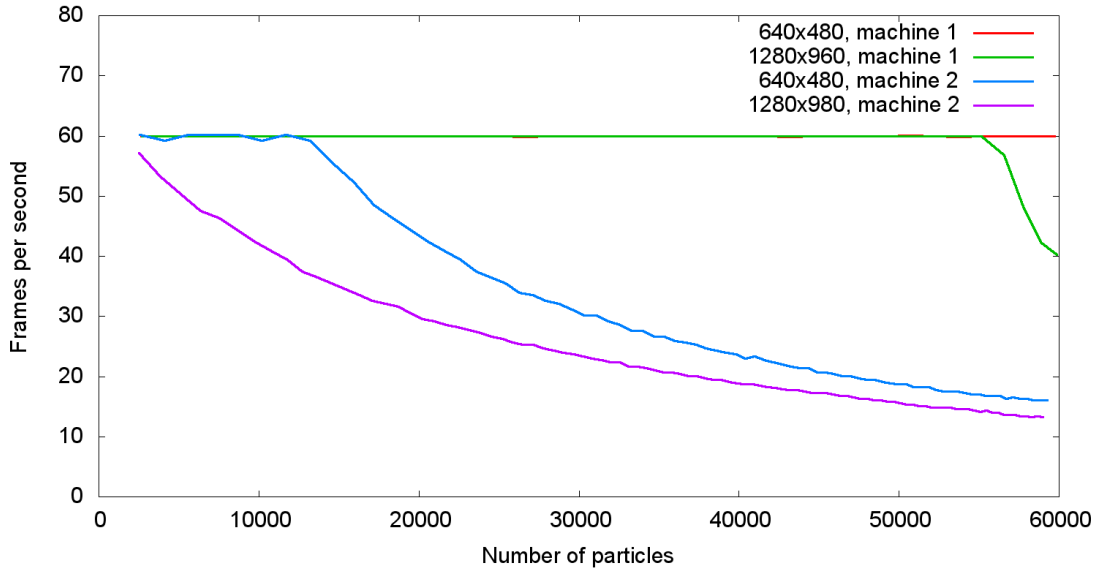


Figure 5.1: Performance of screen space rendering with bilateral Gaussian smoothing - Some description

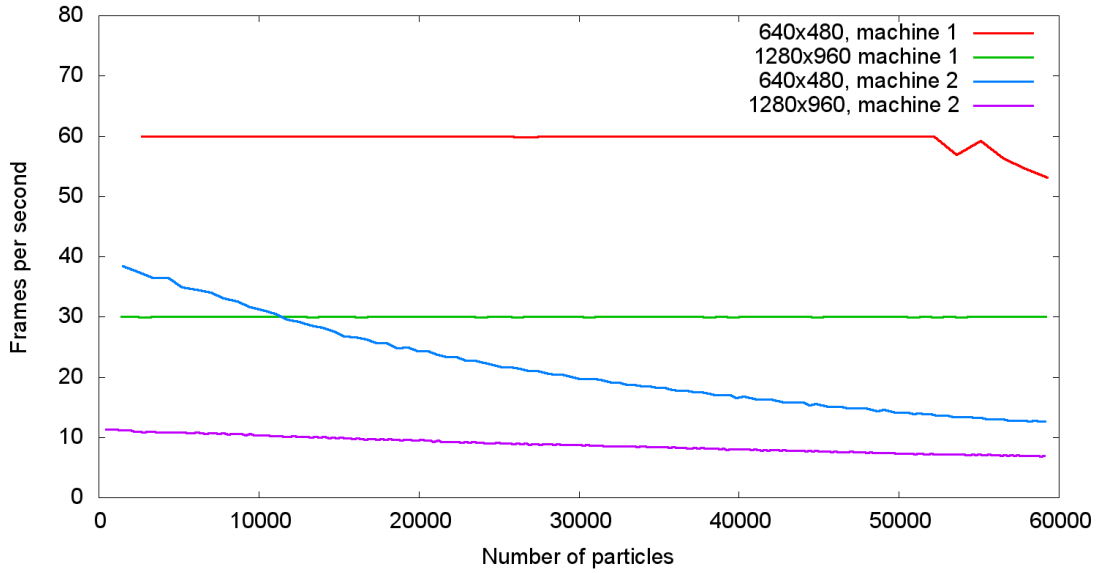


Figure 5.2: Performance of screen space rendering curvature flow smoothing - 100 iterations were performed for each frame.

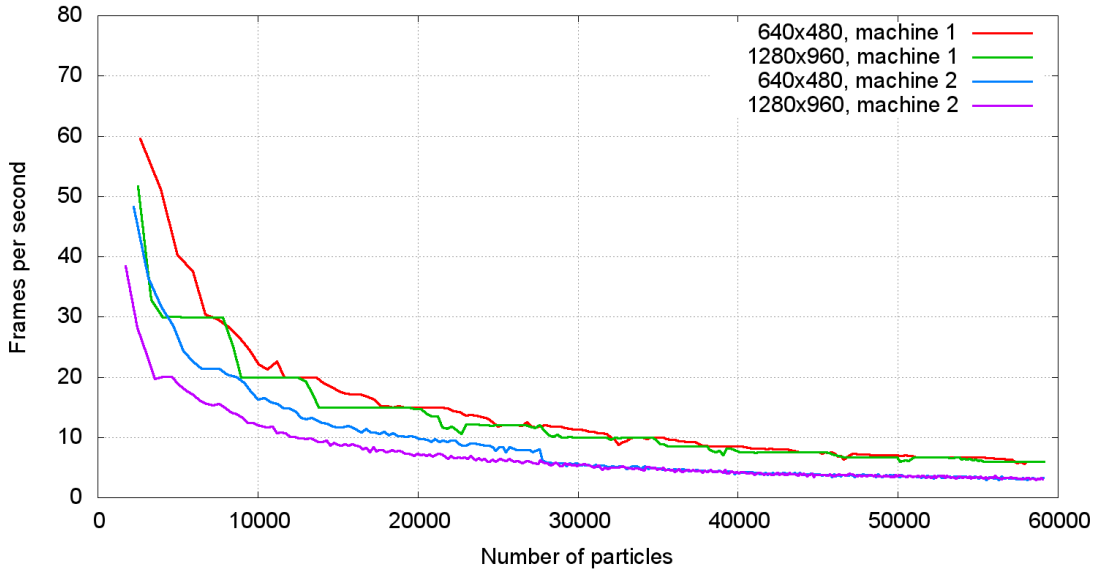


Figure 5.3: Performance of isosurface extraction - 3 threads were used

10000 particles.

Figure 5.4 presents graph of time to render single frame (in milliseconds, excluding simulation time) on number of particles for all three algorithms implemented. It can be seen that all of them have linear complexity, however isosurface extraction has much higher constant. Curvature flow takes around 15 milliseconds more than bilateral Gaussian smoothing, regardless of number of particles simulated.

In table 5.2 times for rendering one frame for different algorithms, resolutions and machines are gathered. They were measured for 60000 particles. It confirms statement that isosurface extraction is not suitable for rendering large number of particles. Only screen space techniques are fast enough, although on lower performance machines curvature flow is also to expensive.

Figure 5.5 shows graph of isosurface extraction algorithm speedup. Entire domain was divided into 4x4x4 = 64 cubes. The performance stops growing after 3 threads. This is due to the fact that CPU has only 4 cores and PhysX simulation was performed simultaneously on one of them. What is more cube division was quite coarse and around half of 64 blocks were empty. On the other side decreasing cube size would lead to increased overhead due to cubes overlaping.

Table 5.2: Performance comparison of all rendering algorithms. Screen space methods were run with 60000 particles, isosurface extraction was run using 3 threads.

Algorithm	resolution	machine	Frame (ms)
Bilateral Gaussian smoothing	640x480	1	10.0
Bilateral Gaussian smoothing	1280x960	1	13.8
Bilateral Gaussian smoothing	640x480	2	51.9
Bilateral Gaussian smoothing	1280x960	2	63.5
Curvature flow smoothing 100 it.	640x480	1	15.6
Curvature flow smoothing 100 it.	1280x960	1	27.8
Curvature flow smoothing 100 it.	640x480	2	73.1
Curvature flow smoothing 100 it.	1280x960	2	134.1
Curvature flow smoothing 60 it.	640x480	1	13.7
Curvature flow smoothing 60 it.	1280x960	1	22.0
Curvature flow smoothing 60 it.	640x480	2	65.7
Curvature flow smoothing 60 it.	1280x960	2	104.5
Isosurface extraction	640x480	1	160.4
Isosurface extraction	1280x960	1	161.5
Isosurface extraction	640x480	2	302.1
Isosurface extraction	1280x960	2	312.3

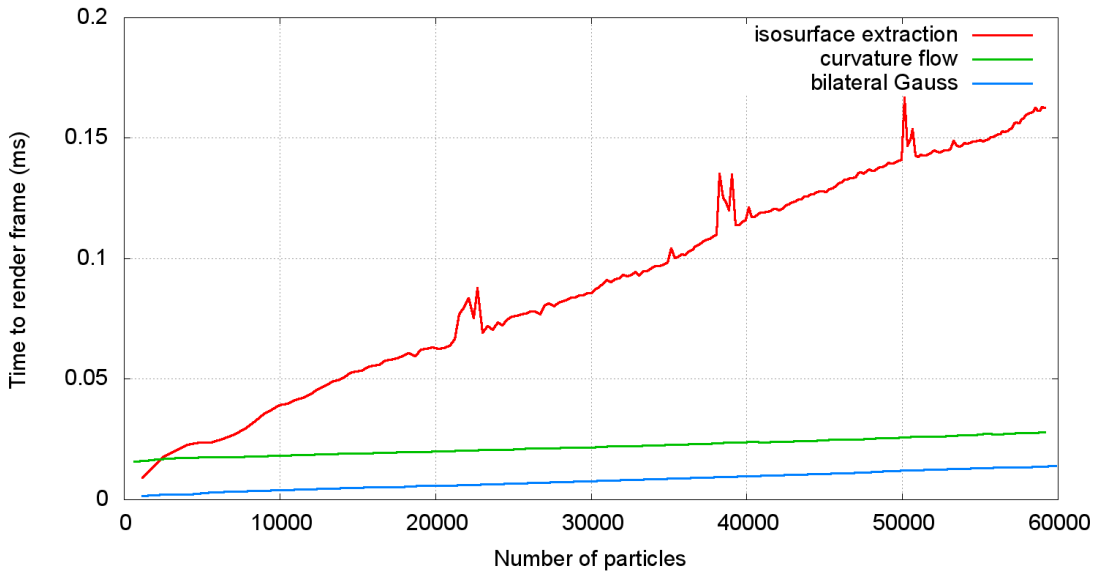


Figure 5.4: Performance comparison of algorithms - Time to render one frame (excluding simulation time) on number of particles. Tests were run on machine 1 in 1280 x 960, isosurface extraction was run using 3 threads, curvature flow smoothing performed 100 iterations.

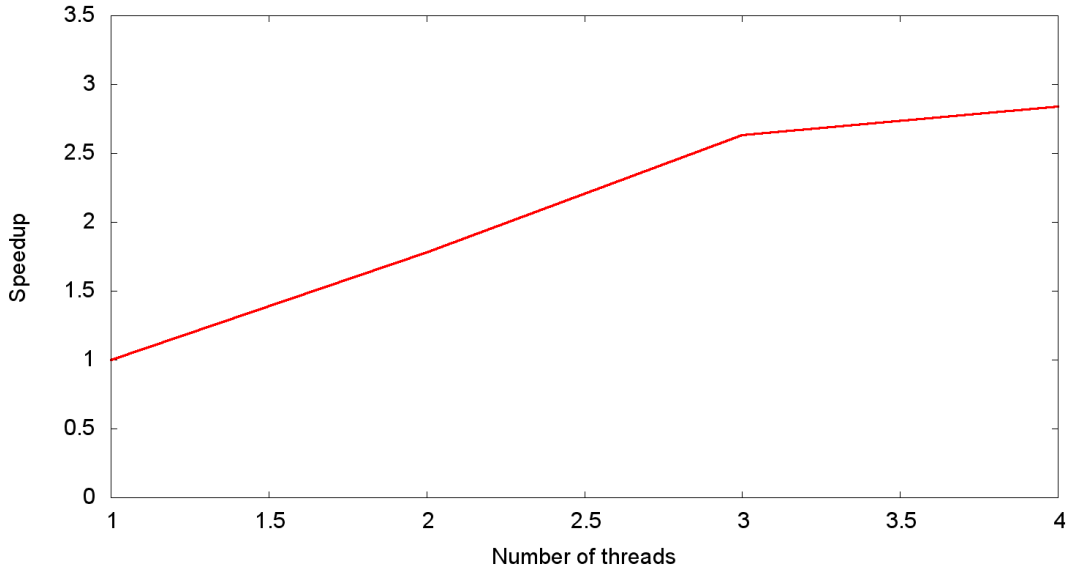


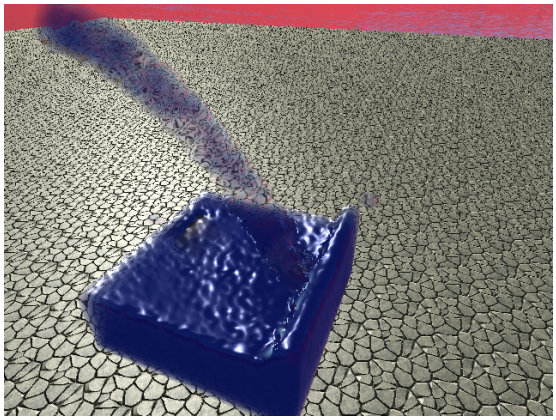
Figure 5.5: Speedup of multithreaded version of isosurface extraction - Tests performed on machine 1 for 60000 particles. Only rendering and surface extraction time were taken into account.

5.1.1 Visual appearance

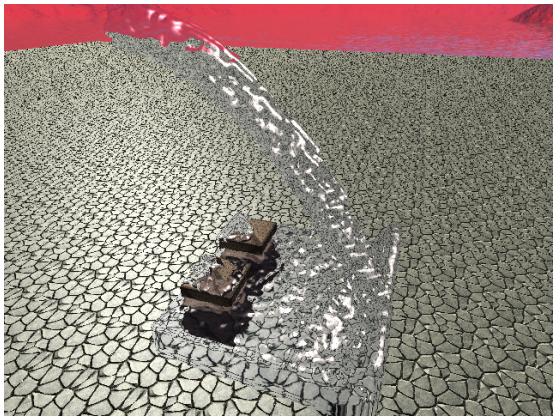
Figure 5.6 shows comparison of visual appearance of waterfall scene rendered with different algorithms. Screen space algorithms include reflection, refraction based on fluid depth and color based on fluid depth. Isosurface extraction does not use depth based refraction and fluid color. Best quality is offered by curvature smoothing with 100 iterations. Bilateral Gaussian smoothing yields similar results as curvature flow with 60 iterations. Fluid rendered using surface extraction is also realistic and it doesn't have artifacts on edges like screen space methods. It's advantage is that it produces surface which can be rendered by game's engine using existing materials. Screen space methods requires custom shaders which can be harder to integrate with game engine.

5.1.2 Conclusions

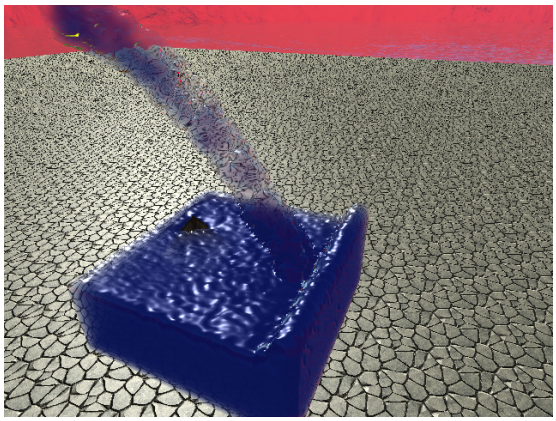
I have presented 3 algorithms for rendering particle fluid. All of them can be used in real time simulations in computer games. However each of them has different application. Isosurface extraction is good for rendering small amount of particles - less than 10000. It can be used to render small fountains, water flowing from a tap or blood. To speedup isosurface extraction it can be implemented on GPU. When higher number of particles are simulated screen space techniques can be used. Bilateral Gaussian smoothing can be used on lower performance and curvature flow smoothing on higher performance hardware.



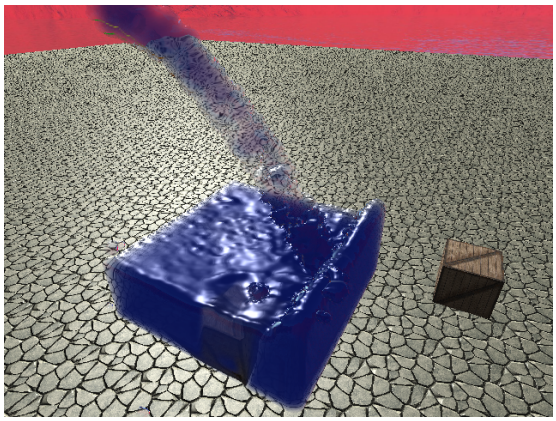
(a) Bilateral Gaussian



(b) Isosurface extraction



(c) Curvature flow, 60 iterations



(d) Curvature flow 100 iterations

Figure 5.6: Comparison of visual appearance of different algorithms.

Bibliography

[1] Ken Birdwell Ilya D. Rosenberg. **Real-time particle isosurface extraction**. *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, 2008. ii, 37, 39, 40, 41, 42, 43, 45, 47

[2] **Ocean surface simulation**. http://developer.nvidia.com/nvidia-graphics-sdk-11-direct3d, 2010. 2

[3] M. B. Liu G. R. Liu. *Smoothed Particle Hydrodynamics a meshfree particle method.* Worlds Scientific Publishing Co. Pte. Ltd., 2003. 5, 6, 7, 8, 9

[4] G. Hauke. *An Introduction to Fluid Mechanics and Transport Phenomena.* Springer, 2008. 5, 8

[5] James F. Price. **Lagrarian and Eulerian Representations of Fluid Flow: Kinematics and the Equations of Motion**. http://www.whoi.edu/science/PO/people/jprice/ELreps.pdf, 2006. 5

[6] J. J. Monaghan. **Smoothed particle hydrodynamics**. *Annual review of astronomy and astrophysics*, 1992. 7, 8, 9, 10

[7] M. Gross M. Müller, D. Charypar. **Particle-based fluid simulation for interactive applications**. *SCA '03 Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, 2003. 8, 9, 10

[8] **OpenGL Language Bindings**. `http://www.opengl.org/wiki/Language_bindings`, 2011. 11

[9] Hubert Nguyen, editor. *GPU Gems 3.* Addison-Wesley Professional, 2007. 12

[10] Richard S. Wright et al. *OpenGL SuperBible: Comprehensive Tutorial and Reference (5th Edition).* Addison-Wesley Professional, 2010. 12, 13, 14, 20

[11] Jason Gregory. *Game Engine Architecture.* A K Peters, Ltd., 2009. 14, 17

[12] Christopher DeCoro Natalya Tatarchuk, Jeremy Shopf. *Advanced Real-Time Rendering in 3D Graphics and Games Course – SIGGRAPH 2007*, chapter 9. SIGGRAPH, 2007. 14

[13] Robert Dunlop. **Linearized Depth using Vertex Shaders**. `http://www.mvps.org/directx/articles/linear_z/linearz.htm"`, 2006. 18

[14] Dave Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1 (7th Edition).* Addison-Wesley Professional, 2009. 20

[15] **Working Draft, Standard for Programming Language C++**. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf, 2011. 20

[16] **PhysX Documentation**. NVIDIA Corporation, 2008. 23

[17] Mark Harris. **CUDA Fluid Simulation in NVIDIA PhysX**. presentation at SGIGRAPHASIA2008, 2008. 23

[18] Simon Green. **Screen Space Fluid Rendering for Games**. Game Developers Conference, 2010. 28

[19] Miguel Sainz Wladimir J. van der Laan, Simon Green. **Screen space fluid rendering with curvature flow**. *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, 2009. 27, 32, 34

[20] Lucas J. van Vliet Tuan Q. Pham. **Separable Bilateral Filtering for Fast Video Preprocessing**. *Multimedia and Expo, 2005. ICME 2005. IEEE International Conference on*, 2005. 30

[21] Harvey E. Cline William E. Lorensen. **Marching cubes: A high resolution 3D surface construction algorithm**. *Proceedings of the 14th annual conference on Computer graphics and interactive techniques, Vol. 21, No. 4.*, 1987. 37

BIBLIOGRAPHY

[22] J. F. Blinn. **A Generalization of Algebraic Surface Drawing**. *ACM Transactions on Graphics*, **1**, 1982. 37

[23] J. L. Bentley. **Multidimensional binary search trees used for associative searching**. *Communications of the ACM*, **18**, 1975. 41