

# Title of your thesis

L<sup>A</sup>T<sub>E</sub>X

Piotr Janisz

Faculty of Electrical Engineering, Automatics, Computer Science and  
Electronics  
University

A thesis submitted for the degree of

*Philosophiæ Doctor (PhD), DPhil,..*

year month

## **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 [reference] and is running on cpu. Authors presented performance of this algorithm running on one CPU core and I will present multithread implementation and it's performance analysis. At the end of this paper I will also present comparison of those two rendering techniques.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	State of the art . . . . .	1
<b>2</b>	<b>SPH</b>	<b>3</b>
2.1	Introduction . . . . .	3
2.2	Fundamentals of SPH . . . . .	5
2.3	SPH for fluids . . . . .	6
<b>3</b>	<b>Tools</b>	<b>7</b>
3.1	Opengl . . . . .	7
3.1.1	Vertex Shader . . . . .	8
3.1.2	Geometry Shader . . . . .	10
3.1.3	Fragment Shader . . . . .	10
3.1.4	Perspective projection and homogeneous space . . . . .	10
3.1.5	Depth buffer . . . . .	13
3.1.6	Framebuffer Object . . . . .	13
3.2	boost . . . . .	15
3.3	PhysX . . . . .	16
3.3.1	Rigid bodies . . . . .	17
3.3.2	Fluids . . . . .	17
<b>4</b>	<b>Rendering techniques</b>	<b>19</b>
4.1	Screen space . . . . .	19
4.1.1	Surface depth . . . . .	20
4.1.2	Smoothing . . . . .	21

## CONTENTS

---

4.1.2.1	Gaussian smoothing . . . . .	21
4.1.2.2	Curvature flow smoothing . . . . .	24
4.1.3	Thickness . . . . .	25
4.1.4	Rendering . . . . .	25
4.2	Isosurface extraction . . . . .	25
4.2.1	Overview . . . . .	26
4.2.2	Block subdivision . . . . .	26
4.2.3	Block processing . . . . .	27
4.2.4	Lookup cache . . . . .	28
4.2.5	Normals generation . . . . .	29
4.2.6	Multithreading . . . . .	31
	<b>Bibliography</b>	<b>33</b>

# 1

## Introduction

### 1.1 Motivation

### 1.2 State of the art

## 1. INTRODUCTION

---

## 2

# SPH

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

### 2.1 Introduction

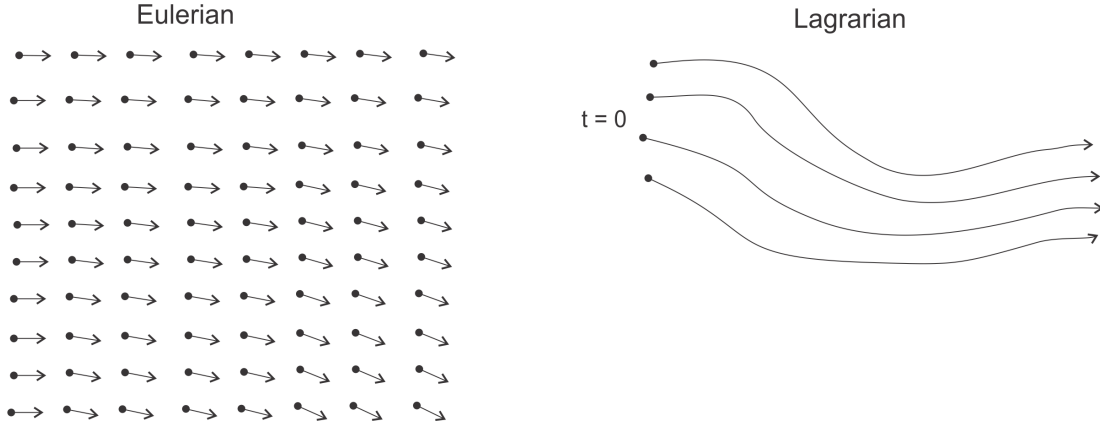
To perform numerical simulation of natural phenomena several steps have to be performed (see (1, section 1.1.2)). First the mathematical model has to be obtained from observation of physical phenomena. The mathematical model consists of a set of governing equations, boundary conditions and initial conditions. Next governing equations have to be discretized in order to solve them with a numerical algorithm. For describing physical governing equations two basic techniques exist: Eulerian description and Lagrangian description.

For describing governing equations of fluid flows two basic techniques exist: Lagrangian and Eulerian. In the first one we look at the fluid as a set of parcels which move through space and time. Each fluid parcel has its own mass which is fixed, and is identified by its 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 (?)). More on fluid field description can be found in (2, section 2.1) and (3)

For discretization there are several approaches: grid based (Lagrangian grid and Eulerian grid) and meshfree methods (SPH, particle in cell). Lagrangian grid is attached

## 2. SPH

---



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

to material and moves with it. Each mesh cell represents one fluid parcel from Lagrangian description. The advantages of this method are simpler equations given by Lagrangian 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 over time (like for instance water splashes) well. In such cases mesh must be recomputed 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 memory required is also large as whole domain must be covered with grid. Third way of discretization are mesh free methods. The idea of them is to combine strong points of both Eulerian and Lagrangian 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. Lagrangian and Eulerian grids as well as meshfree methods are described in more details in (1, 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):

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

Where  $W$  is a smoothing kernel and has following properties:

$$\int W(r - r', h)dr' = 1 \quad (2.2)$$

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

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

Integral interpolant 2.1 can be further approximated by sumation interpolant:

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

where  $\frac{m_b}{\rho_b}$  is a volume of particle  $b$ . What SPH does is 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 (1, section 2.2.1)). On the other hand accuracy of sumation interpolant 2.5 is hard to predict (see (4, section 12.1)).

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  $W$  ((1, section 2.2.2)):

$$\nabla A_I(r) = \int A(r') \nabla W(r - r', h) dr' \quad (2.6)$$

## 2. SPH

---

As earlier we can approximate integral with summation:

$$\nabla A_s(r) = \sum_b m_b \frac{A_b}{\rho_b} \nabla W(r - r_b, h) \quad (2.7)$$

SPH is a interpolation method - does not uses a grid (MESHFREE), approximates fluid or other materials as a set of particles. Requires less resources comparing to grid methods when it comes to such phenomenas as splashes or breaking waves.

SPH first used by Leon Lucy and R. A. Gingold and J. J. Monaghan

Different kernels can be used for different computations.

paper on astrophysics from 1992 (Monaghan1992) - (4)

book about sph (Liu) - (1)

Particle based simulations for interactive applications (Muller2003) - (5)

### 2.3 SPH for fluids

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

$$\frac{DM}{Dt} = 0 \quad (2.8)$$

second one is Navier-Stokes conservation of momentum equation:

$$\rho \frac{Dv}{Dt} = -\nabla p + \mu \nabla^2 v + f \quad (2.9)$$

Both equations are presented in Lagrarian form.  $\frac{D}{Dt}$  is a material derivative (TODO opisa co to jest, referencja).  $p$  is the pressure,  $\mu$  is fluid viscosity and  $f$  represents external forces acting on the fluid.

Navier-Stokes equations: - continuity, - momentum conservation, - energy conservation For SPH first one is not necessary and for fluid simulation in games energy conservation equation can be discarded as we only interested in fluid movement, not temperature (we can assume the temperature is constant).

TODO SPH equations needs to be symmetrized.

## 3

# Tools

This chapter will present brief introduction of tools used in building visualization system. This is not 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 (6)).

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 changing 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-

### 3. TOOLS

---

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 (7, 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 for 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. Most frequently used and most useful functionality of OpenGL 2.1 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.

#### 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 (8, pages 412-419)).

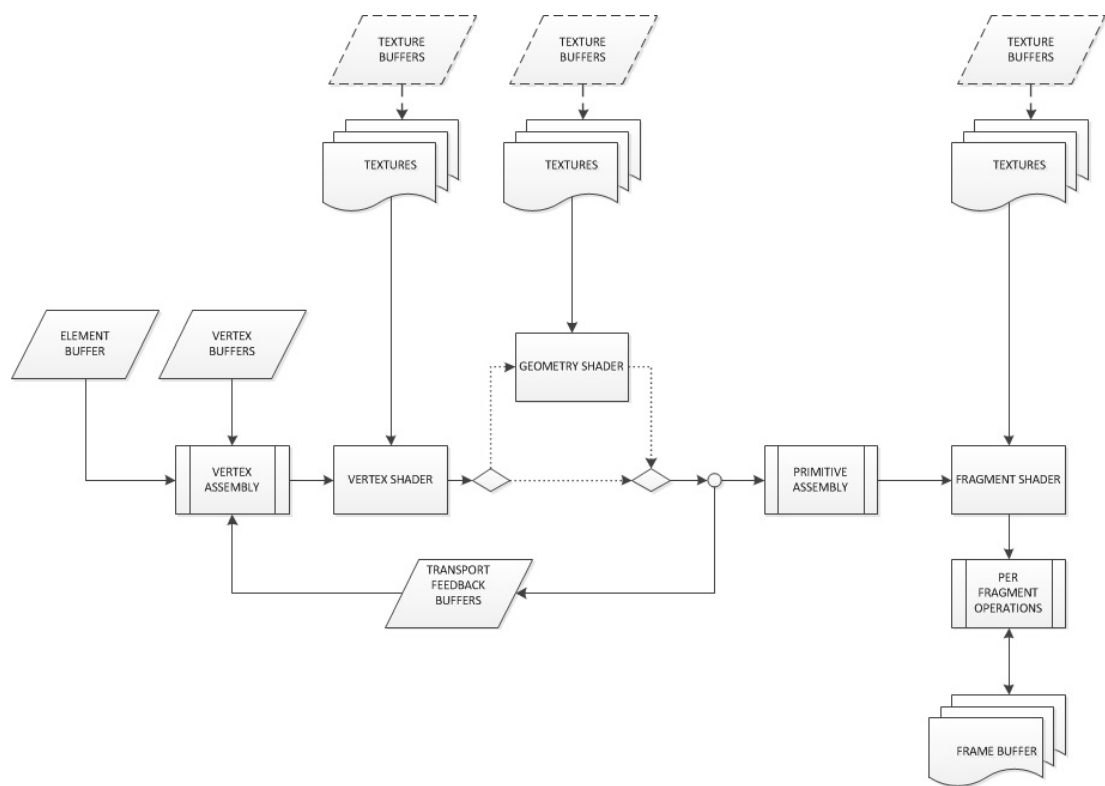


Figure 3.1: OpenGL 3.3 pipeline - taken from (8, chapter 12)

## 3. TOOLS

---

### 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 (9), drawing vertices normals (8, pages 434-437) to more advanced as shadow volume extrusion (9, section 10.3.3.1), isosurface extraction (10) 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 one several fragments may contribute to one pixel. The fragment shader 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.

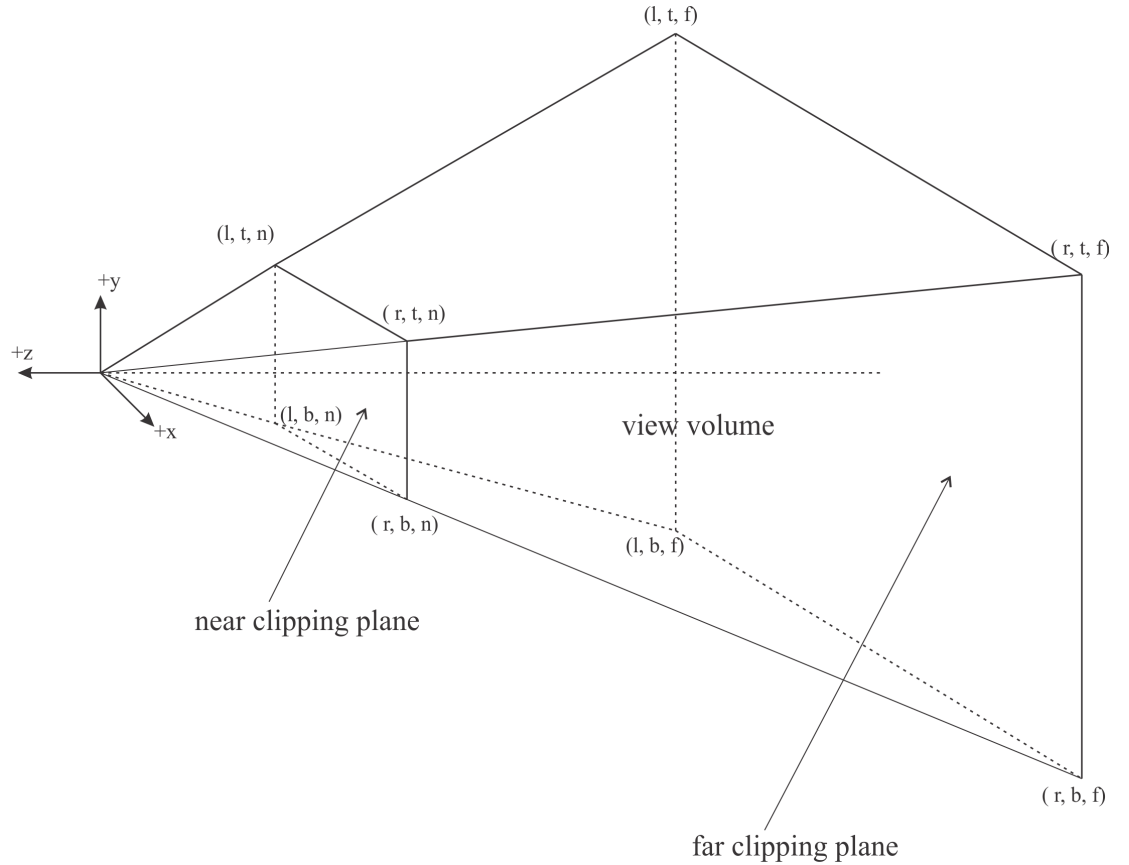
### 3.1.4 Perspective projection and homogeneous space

Perspective projection mimics the kind of image produced by a typical camera by providing effect known as perspective foreshortening. With this effect the farther objects lies from camera the smaller it appears on the screen. Perspective projection transforms points from 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 which part that lies inside four side planes is a virtual screen. Virtual screen left, right, bottom and top edges lies at  $x = l, x = r, y = b, \text{ and } y = t$ . It's z coordinate is equal to -n. Those parameters are used to computed projection matrix  $M_{V \rightarrow H}$  (equation 3.1) ( $V \rightarrow H$  means transformation from view space into homogeneous space):

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

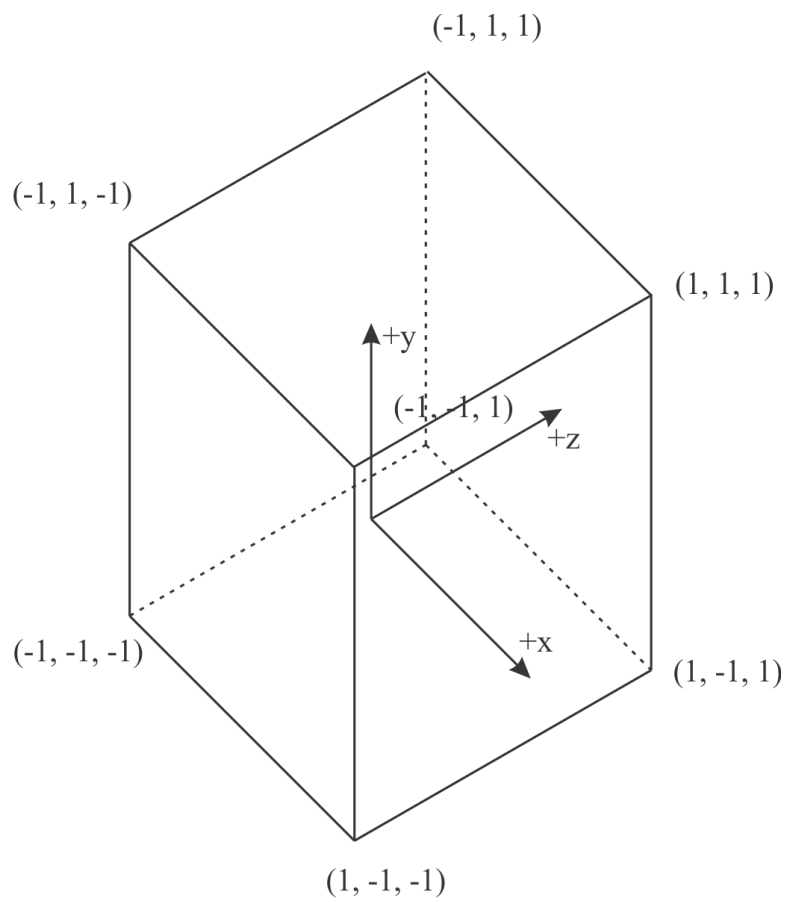
where  $w = r - l$  and  $h = t - b$ .



**Figure 3.2:** A perspective view volume (frustum)

### 3. TOOLS

---



**Figure 3.3:** The canonical view volume in homogeneous clip space



### 3.1.5 Depth buffer

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

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

thus we get:

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

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

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

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  $d(z)$  is following:

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

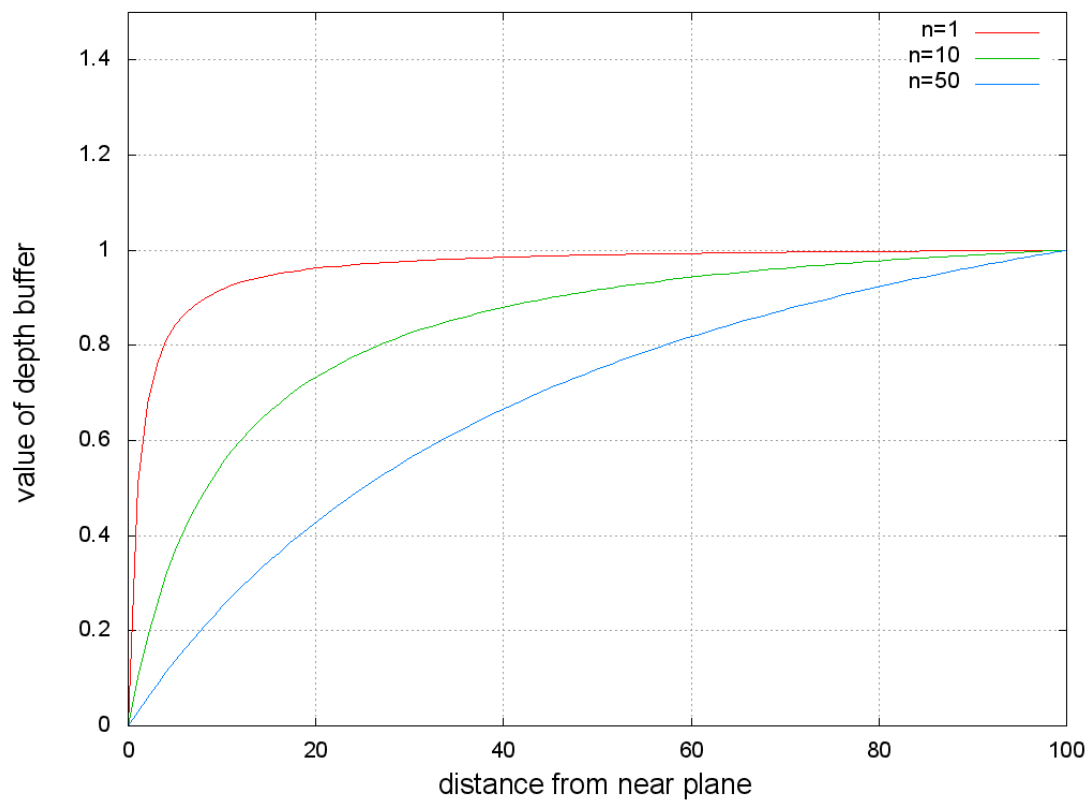
Equation 3.5 shows that value of depth buffer is a nonlinear function of z component of the point. Figure 3.4 shows how this value changes with changing near plane distance from the viewer. It can also be seen that precision is not evenly distributed and it decreases with the distance from near plane.

### 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 then used as an input to the pipeline again but this requires additional copy

### 3. TOOLS

---



**Figure 3.4:** Value of z buffer as a function of distance from near plane for different values of  $n$ .  $f - n$  is kept constant at 100

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 to (8, 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 implementing 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 then can be found in (8, chapter 8) and (11, chapter 10).

## 3.2 boost

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 - like regular expressions (FIXME).

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 over again.

Boost libraries used in my project are boost regex and boost threads. The first one is used to process configuration files. Boost trheads library is used to parallelize isosurface extraction algorithm presented in (TODO reference).

TODO reference to documentation or main page

### 3. TOOLS

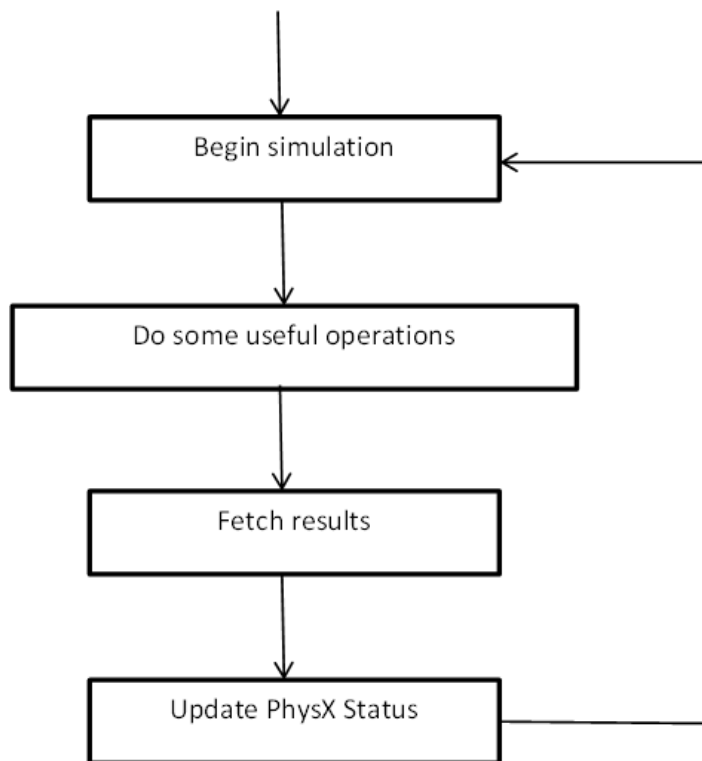
---

#### 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 simulations.

The main drawback is that hardware acceleration can only be performed on Nvidia 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 in figure 3.5.



**Figure 3.5:** 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

#### 3.3.2 Fluids

PhysX provides two methods for simulating fluids SPH and simple method. First one was described in chapter (TODO reference to chapter). Second one does not take interparticle forces into consideration thus simplifies computations. This simplified method is good for simulating phenomenas such as smoke or fog, but for realistic water SPH must be used.

TODO more details about fluids in PhysX - simulation methods, parameters, compare with SPH chapter, emitters, drains, limitations (64K particles per fluid)

TODO *ftp : //download.nvidia.com/developer/cuda/seminar/TDCI\_PphysX.pdf*

TODO prezentacja

### 3. TOOLS

---

## 4

# Rendering techniques

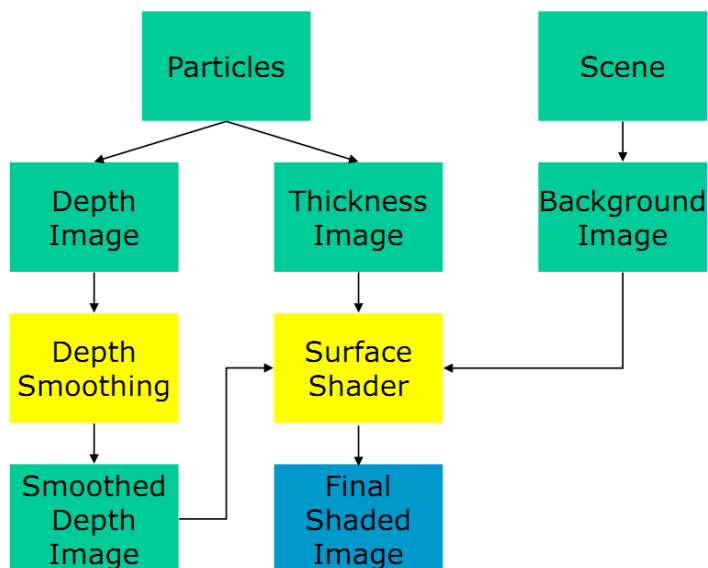
Output from particle base simulations is list of particles containing positions. It can also contain additional parameters - for example PhysX fluid simulation returns velocity, lifetime and density in addition to position for each particle. Traditionally triangle meshes are used for rendering objects. Using this approach requires constructing fluid triangle mesh for given particle positions. This method is called isosurface extraction and will be described in second section of this chapter. Another possibility is to render each particle as a point (or a billboard in general) with opacity so that 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 prevents 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 first section.

### 4.1 Screen space

As mentioned before this approach extracts visible surface by rendering each particle into depth buffer. High level overview of this method is presented on figure 4.1 and consists of following steps (13): 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).

## 4. RENDERING TECHNIQUES

---



**Figure 4.1: Screen Space Fluid Rendering** - The figure shows high level overview of the method, taken from (12)

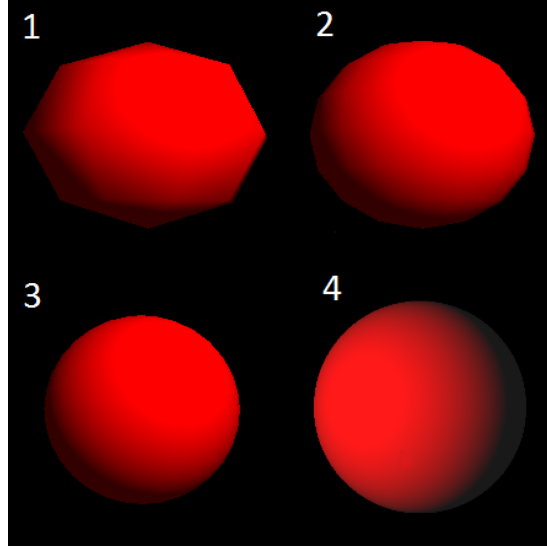
### 4.1.1 Surface depth

To obtain fluid surface visible from the viewpoint of camera each particle is rendered as a sphere into depth texture [rysunek]. At each pixel only closest value is kept using hardware depth test. To avoid rendering large amount of geometry each 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 4.2) because depth values are generated for each pixel. Rendering spheres as point sprites is 10 to 100 times faster comparing to rendering them as triangle meshes (see table 4.1).

**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





**Figure 4.2: 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

### 4.1.2 Smoothing

Although previous step produces surface it's quality is not sufficient. As can be seen on figure 4.3 individual spheres can be seen giving fluid surface gelly-like 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.

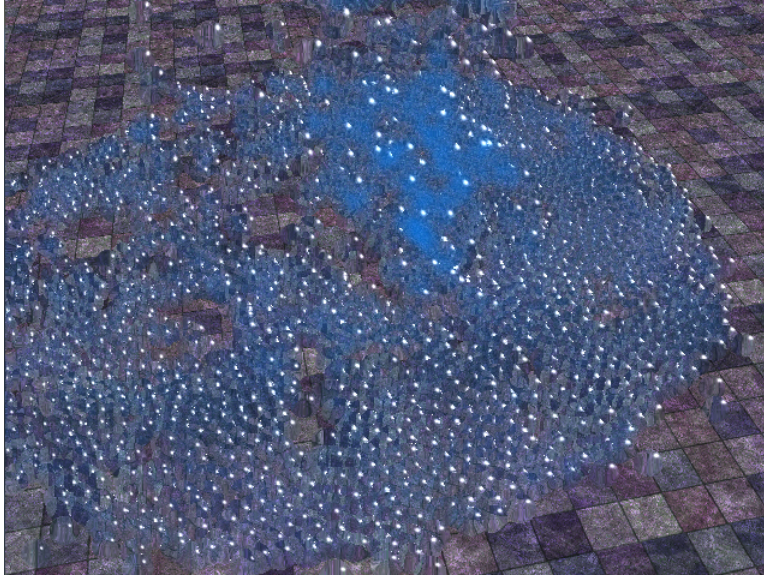
#### 4.1.2.1 Gaussian smoothing

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 it's linear separability. However this filter produces undesired effect of blending drops of fluid with background surfaces (see figure 4.4). Thus edge-preserving filters needs to be used which are also called bilateral filters. 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 (14)):

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

#### 4. RENDERING TECHNIQUES

---



**Figure 4.3: Fluid surface rendered without smoothing phase** - Individual spheres can be seen, giving surface unnatural appearance

TODO

**Figure 4.4: Fluid surface rendered with Gaussian smoothing** - From left: depth image, depth image after applying Gaussian filter, diffuse shaded surface

where

$$f(s, s_0) = g_s(s - s_0)g_t(I(s) - I(s_0)) \quad (4.2)$$

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:

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

As can be seen on 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 gives for 2 dimensional image about 400 operations for each pixel when using bilateral filtering. In comparison separable implementation requires 40 operations. It turns out however that computing bilateral filter as it was a space-invariant gives good approximation for real time applications (see figure 4.5). Approximation gives some artifacts but they are not visible when fluid is moving and other effects (reflection and refraction) are applied.

TODO

**Figure 4.5: Comparison of normal bilateral filter with it's separable approximation** - Upper row presents images for bilateral filter and bottom row presents separable approximation.

Bilateral filter has some undesired effect when applied on z-buffer output. The problem is that depth values are not evenly distributed. This mean that difference

## 4. RENDERING TECHNIQUES

---

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. In result the further surface lies from camera the less edges are preserved (see figure 4.6). Alternative technique to Z-buffering is W-buffering which offers better distribution of depth values (9). Some hardware supports W-buffering but most of it doesn't and it doesn't seems to be supported in future. A workaround to this problem was presented in (15). It customizes projection transformation in vertex shader in way that depth buffer values have linear distribution at the end.

TODO

**Figure 4.6: 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

Another issue with smoothing using fixed kernel size filter is that further surfaces are more smoothed than closer surfaces (see figure 4.6). Solving that problem would require using filter with variable kernel size, depending on fragment distance from viewer. As implementing this efficiently on graphic hardware is hard and the effect is not so irritating this won't be considered any more. 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 (13). It uses curvature flow to smooth surface. Curvature is a .... TODO.

So instead of smoothing surface itself we can smooth out sudden changes in curvature. Such changes occur between particles as shown on figure 4.7

TODO

**Figure 4.7:** Some title - Some description

#### 4.1.3 Thickness

This step is computed to determine how opaque is surface in given point. It is achieved 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 fluid depth texture, thickness texture and texture with rest of the scene rendered. [TODO normal reconstruction, refraction, reflection, thickness]

### 4.2 Isosurface extraction

Classic algorithm for isosurface extraction is marching cubes (see (16)). 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).

## 4. RENDERING TECHNIQUES

---

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 (17). 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 radius of influence ( $R_c$ ), isosurface threshold and produces list of triangles. Particle 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:

$$F(p) = \sum_{s \in D_{R_c}(p)} f(dst(s, p)) \quad (4.4)$$

where  $s$  is a center of particle,  $D_{R_c}(p)$  is a disc with  $p$  as a center and radius  $R_c$ .  $f(r)$  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 with a few operations:

$$f(r) = \begin{cases} (\frac{r}{\sqrt{2}R_c})^4 - (\frac{r}{\sqrt{2}R_c})^2 + 0.25 & \text{if } r < R_c \\ 0 & \text{otherwise} \end{cases} \quad (4.5)$$

When  $R_c > R$  neighboring particles combines into one smooth surface (see 4.8). This is a common technique known as metaballs (see TODO cytat). The biggest problem here is to efficiently compute field values at corners and to traverse space in most optimal way. Those topics will be described in following subsections.

### 4.2.2 Block subdivision

Space is divided into blocks containing cubes. Such a division aims for:

- Reduce memory consumption - smaller blocks contains less particles and cubes.

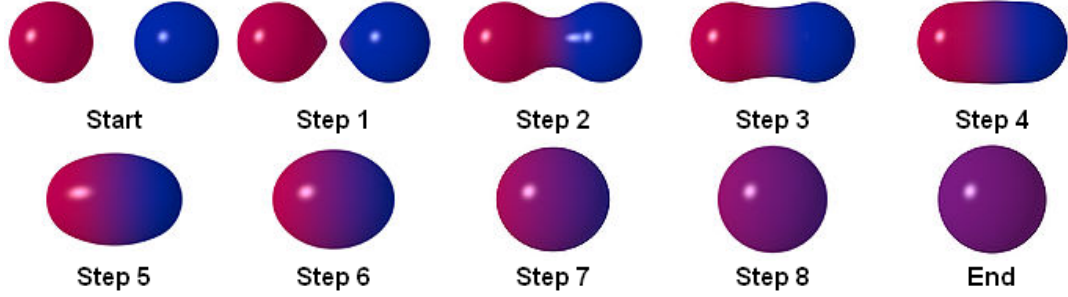


Figure 4.8: Illustration of combining particles - SOMETHING

- Allowing for parallelize algorithm in an easy way - because processing one block is an 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 there are no dependencies between them.

TODO imgae of block

Block subdivision is a first step of the algorithm. For every particle containing blocks are computed. 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.

### 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 (17). Block containing  $n \times n \times n$  cubes is divided to  $n$  slabs and  $n+1$  slices (TODO rysunek). 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.

To speed up a marching cubes algorithm we would like to traverse only cubes that contains surface. This requires finding so called seeding cubes - initial cubes at which

## 4. RENDERING TECHNIQUES

---

algorithm will start traversing block. We then choose one slab 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 todo lists of [odpowiadajacy] slabs.

If we pick up a random slabs and cubes each time then we have to keep all cached values in slices while processing a cube. The idea of marching slices is to decrease memory used by cache 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 would be to find surfaces all local minimas - then we would have to store only cache in slices below and above current slab. However this approach would be time consuming. (17) 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  $F(c) > \text{threshold}$  and  $|c - c_{\text{start}}| < 2$  do
         $c = \text{corner below } c$ 
    if  $F(c) < \text{threshold}$ 
        add cube to it's slab todo list
```

This method will not find local minimas every time as it always goes down from the centre of a particle. Figure 4.9 shows the case when algorithm (TODO ref) fails to find a local minimum.

### 4.2.4 Lookup cache

To compute field value for a given corner all particles that lies within  $r_c$  from it have to be found. This task can be accomplished by building space partitioning data structure like octtree or kdtree (TODO moe odnoniki). However costs of building spatial data structures are relatively high, and the nearest neighbor finding algorithm requires floating point computations. (17) presented another to accomplish this task. Presented method does use 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 all corners containing all particles lying within  $r_c$  from it. So for every particle



TODO

**Figure 4.9: Finding local minima** - In this example algorithm fails to find local minimum

we would traverse all corners lying inside a sphere of radius  $r_c$  and update their particles list by inserting the current one. TODO mention about. The main disadvantage of this approach is that we have to store a information for each corner, and for every corner we have to store OSOBNALIST. 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  $(x, y)$  and stores a linked list of particles that have influence in that column. TODO image. 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.

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 have an influence.

Looking up particle

#### 4.2.5 Normals generation

In order to properly shade extracted surface normal vevtors have to be computed for every 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

## 4. RENDERING TECHNIQUES

---

that vertices are not shared between triangles thus it's not applicable with surface extraction method used.

Another approach is to compute normal at each vertex as a gradient (FIXME) of scalar field. (TODO wzor). As values of field are given only in corners and mostly 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.

TODO

**Figure 4.10: Different approaches to normal generation** - gradient, average, weighted average based on triangle area, weighted average based on incident angle

Yet another approach is to compute normal for every triangle that includes given vertex and use a 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. In my algorithm weight are the angles because with area long and narrow triangles with will still can have much impact on normal.

#### 4.2.6 Multithreading

(17) presented algorithm that run in one thread. However algorithm has a potential of parallelization due to block processing. Parallelization of this algorithm is even more desired as nowadays CPU manufacturers increase performance by multiplying CPU cores instead of increasing clock frequency of 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 (TODO). 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 current frame. This increases latency but it is not visible when frame rate is high.

TODO diagram

#### 4. RENDERING TECHNIQUES

---

# Bibliography

- [1] M. B. LIU G. R. LIU. *Smoothed Particle Hydrodynamics a meshfree particle method*. Worlds Scientific Publishing Co. Pte. Ltd., 2003. 3, 4, 5, 6
- [2] G. HAUKE. *An Introduction to Fluid Mechanics and Transport Phenomena*. Springer, 2008. 3
- [3] JAMES F. PRICE. **Lagrangian and Eulerian Representations of Fluid Flow: Kinematics and the Equations of Motion**. <http://www.whoi.edu/science/PO/people/jprice/ELreps.pdf>, 2006. 3
- [4] J. J. MONAGHAN. **Smoothed particle hydrodynamics**. *Annual review of astronomy and astrophysics*, 1992. 5, 6
- [5] 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. 6
- [6] **OpenGL Language Bindings**. [http://www.opengl.org/wiki/Language\\_bindings](http://www.opengl.org/wiki/Language_bindings), 2011. 7
- [7] HUBERT NGUYEN, editor. *GPU Gems 3*. Addison-Wesley Professional, 2007. 8
- [8] RICHARD S. WRIGHT ET AL. *OpenGL SuperBible: Comprehensive Tutorial and Reference (5th Edition)*. Addison-Wesley Professional, 2010. 8, 9, 10, 15
- [9] JASON GREGORY. *Game Engine Architecture*. A K Peters, Ltd., 2009. 10, 24
- [10] CHRISTOPHER DECORO NATALYA TATARCHUK, JEREMY SHOPF. *Advanced Real-Time Rendering in 3D Graphics and Games Course SIGGRAPH 2007*, chapter 9. SIGGRAPH, 2007. 10
- [11] DAVE SHREINER. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1 (7th Edition)*. Addison-Wesley Professional, 2009. 15
- [12] SIMON GREEN. **Screen Space Fluid Rendering for Games**. Game Developers Conference, 2010. 20
- [13] 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. 19, 24
- [14] 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. 21
- [15] ROBERT DUNLOP. **Linearized Depth using Vertex Shaders**. [http://www.mvps.org/directx/articles/linear\\_z/linearz.htm](http://www.mvps.org/directx/articles/linear_z/linearz.htm), 2006. 24
- [16] HARVEY E. CLINE WILLIAM E. LORENSSEN. **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. 25
- [17] KEN BIRDWELL ILYA D. ROSENBERG. **Real-time particle isosurface extraction**. *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, 2008. 26, 27, 28, 31