Title of your thesis

LaTeX

Piotr Janisz

Faculty of Electrical Engineering, Automatics, Computer Science and Electronics

University

A thesis submitted for the degree of

PhilosophiæDoctor (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. Emphais 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 comparision of those two rendering techniques.

Contents

1

Introduction

2

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.

2.1 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 2.1 and consists of following steps (2): rendering depth texture (section 2.1.1) and thickness textures (section 2.1.3), depth smoothing (section 2.1.2) and assembling fluid surface with rest of the scene (section 2.1.4).

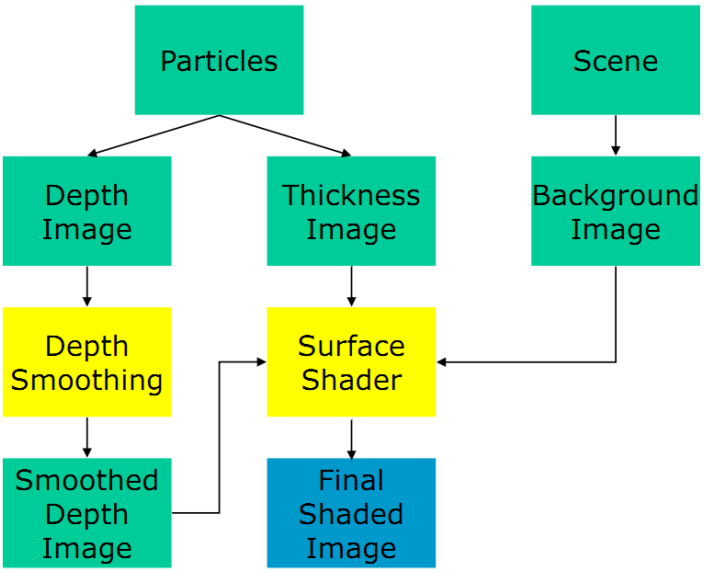

Figure 2.1: Screen Space Fluid Rendering - The figure shows high level overview of the method, taken from (1)

2.1.1 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 2.2) 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 2.1).

Table 2.1: Comparision of sphere rendering methods

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

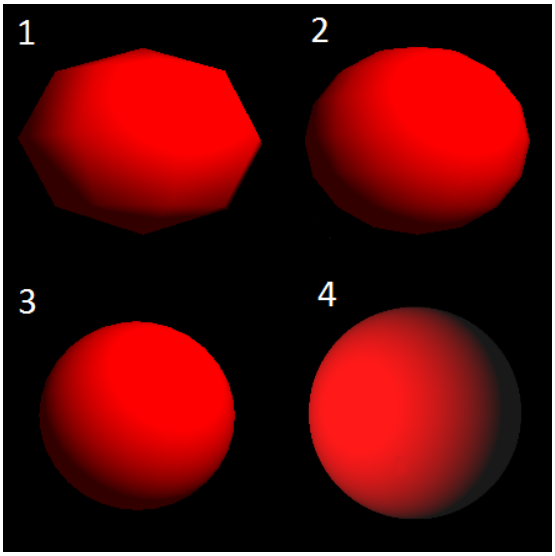

Figure 2.2: Spheres rendered with different methods - 1 - mesh with 128 triangles, 2 - mesh with 512 triangles, 3 - mesh with 2048 triangles, 4 - point sprite with normals generated in fragment shader

2.1.2 Smoothing

Although previous step produces surface it's quality is not sufficient. As can be seen on figure 2.3 individual spheres can be seen giving fluid surface gelly-like appearance. To remove this artifact some kind of smoothing has to be applied. Section 2.1.2.1 will describe gaussian smoothing and section 2.1.2.2 curvature flow smoothing.

2.1.2.1 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 2.4). 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 (3)):

$$O(s_0) = \frac{\sum_{s \in S} f(s, s_0) I(s)}{\sum_{s \in S} f(s, s_0)} \tag{2.1}$$

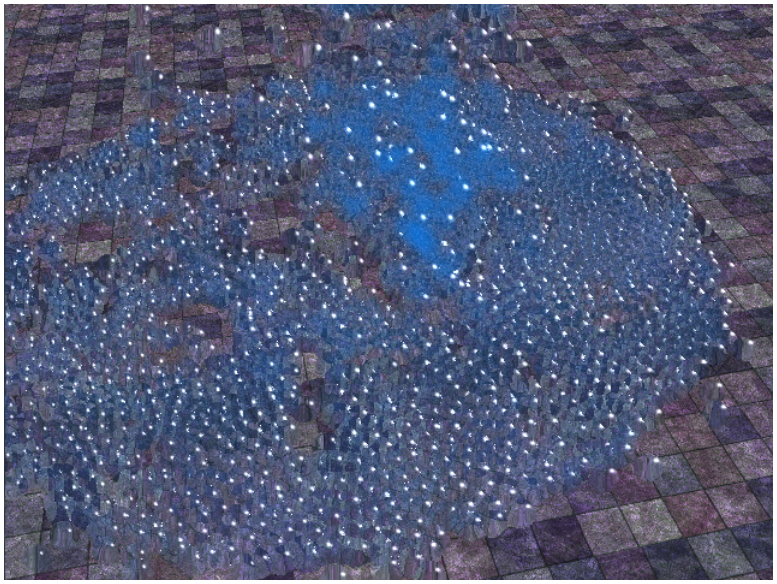

Figure 2.3: Fluid surface rendered without smoothing phase -

TODO

Figure 2.4: Fluid surface rendered with Gaussian smoothing - From left: depth image, depth image after applying Gaussian filter, diffuse shaded surface

where

$$f(s, s_0) = g_s(s - s_0)g_t(I(s) - I(s_0)) \tag{2.2}$$

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:

$$g_s(s) = g(x, \sigma_S)g(y, \sigma_S) \quad g_t(I) = g(I, \sigma_t) \tag{2.3}$$

As can be seen on equation 2.2 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 2.3). Computional 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 2.5). Approximation gives some artifacts but they are not visible when fluid is moving and other effects (reflection and refraction) are applied.

TODO

Figure 2.5: Comparision of normal bilateral filter with it's separable approximation - Upper row presents images for bilateral filter and bottom row presents separable approximation.

[TODO mention about using w-buffer instead of z-buffer for bilateral gauss work correctly] [TODO mention about changind kernel size with pixel distance]

2.1.2.2 Curvature flow smoothing

This technique was described in [odnonik].

2.1.3 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 linearly interpolated. [TODO obrazek takiej tekstury]

2.1.4 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]

2.2 Isosurface extraction

There will be several preliminary scientific targets to be accomplished on the way...

Bibliography

[1] Simon Green. **Screen Space Fluid Rendering for Games**. Game Developers Conference, 2010. 4

[2] 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. 3

[3] 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. 5