Screen Space Fluid Rendering with Curvature Flow

Maarten Terpstra* and Bram Musters*

*Department of Computing Science
University of Groningen
Nijenborgh 9
Groningen 9747 AG
{m.l.terpstra, b.t.musters}@student.rug.nl

Abstract

Fluids can be simulated using Smoothed Particle Hydrodynamics(SPH). However, visualizing these fluids in a realistic way proves to be hard. The paper by van der Laan et al. [2009] proposes a method that visualizes this SPH simulation as a nature-like fluid using surface smoothing by screen-space curvature flow. The advantage of this method is that it can be rendered in real-time for a high amount of particles without a need for a finite grid. This paper analyzes the proposed method and describes how to implement it using C++, OpenGL and the nVidia Cg toolkit.

Keywords: Smoothed Particle Hydrodynamics, fluid, visualization, computer graphics

1 Introduction

Fluids are abundant in nature: whether close to our homes, falling from the sky, deep in the earth in the form of magma, or in large volumes as lakes or oceans, fluids are nearly guaranteed to be found everywhere. These fluids are not only necessary for most animals roaming the planet to survive, but are also used by humans in industrial processes to obtain power, starch, sawed wood or cooling in our engines.

For some applications it may be useful to simulate these fluids using computers in order to improve usage of the fluids in certain situations. By using visualizations of fluids bottlenecks or missed opportunities in systems may be identified and the model can modified to improve the system. This kind of visualizations are mainly used in engineering.

Other types of visualizations, which are mainly used in academia, includes fields of research as oceanography, volcanology and astrophysics. In these kind of visualizations, entire systems are modeled to identify breaking points or how different parameters may affect an entire system.

There are several methods available for simulating fluids, such as the Eulerian method or the Smoothed Particle Hydrodynamics (SPH). The Eulerian method is fairly straightforward, as it describes fluid motion as an integration of the surface over time. This simple technique comes with a downside in the fact that it requires a (finite) grid on which the fluid moves, which is computationally expensive and limits the fluid in that it cannot move everywhere. SPH proves to be a viable alternative. With this technique, a body of fluid is reduced to a number of particles, where each particle has a position, velocity, acceleration and density. These properties can be used to extract a smoothed surface which can then be visualized in an aesthetically pleasing way.

Simulating a fluid is only half the battle. The insight gained from a fluid rendering can increased by having a natural visualization of the system.

2 Problem Definition

Since the fluids are simulated using SPH, the fluid is discretized in different particles. A scene containing a fluid, simulated by SPH, can be seen in figure 1. Since this "fluid" does not look like a fluid that can be found in nature, a visualization technique has to be applied.
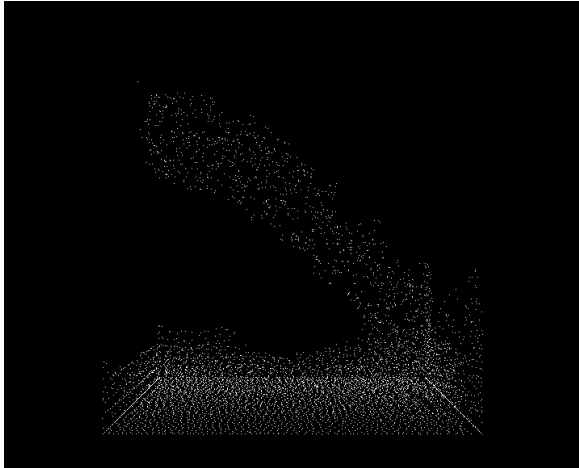

Figure 1: SPH simulation

It is our goal to create a nature-like rendering of fluids that can be rendered in real-time, in order to use it in games, for example. It is also desirable to make sure that the rendering can be customized according to the requirements of users. For example, fluids can have different kind of thicknesses, or the quality of the rendering can be altered in order to keep the rendering in real-time.

It is also import to distinct what makes a rendering realistic and in real-time. We define real-time to be a simulation that is smooth, stutter-free and runs in about the same as the real world. In practice, this is not always feasible and therefore we define a simulation to be real-time if it produces at least 20 frames per second.

A nature-like visualization of a fluid is a tangible concept as humans have a clear vision of what is a realistic visualization and what is not, but it is not quite as easy to formally define.

There are several components that contribute to a realistic visualization of a fluid. In order to be realistic a fluid surface needs to be smooth. That is, there are no sharp discontinuities between particles.

Moreover, the surface needs to be imperfect. This may sound counter-intuitive as it contradicts with the point that a surface needs to be smooth, but it turns out that in nature a fluid surface is rarely perfectly smooth due to wind and other natural effects.

Finally, a fluid should also act as a fluid in a simulation. This means that object attributes as color should be attenuated as that object is submerged in the fluid and the fluid should produce foam if it is mixed with gases in large quantities.

The combinations of these components are what can make a fluid surface visualization realistic and it these components we seek.

2.1 Related work

Various methods are developed that try to achieve a nature-like rendering of fluid. Some of these techniques require to use a mesh, which is not desirable. Other techniques have the drawback that they can not be rendered in real-time.

Zhang et al. [2008] developed a method that makes use of point-based rendering, therefore a grid is unnecessary. However, a drawback of this method is that it results in unreasonably thick surfaces.

3 Solution

The paper introduced by van der Laan et al. van der Laan et al. [2009] provides a fitting solution to our problem. The paper describes a method for visualizing fluids simulated using SPH in a natural way. The goal of the paper is to provide a solution to our aforementioned problem. According to the paper, they want to create:

1. Achieves real-time performance, with a configurable speed versus quality trade-off.

2. Does all the processing, rendering and shading steps directly on the graphics hardware.

3. Smooths the surface to prevent the fluid from looking blobby or jelly-like.

4. Is not based on polygonization, and thus does not suffer from the associated tessellation artifacts.

Items 1 and 3 have a direct connection with our goals and items 2 and 4 are ways to enable these goals.

The method described in the paper consists of multiple passes. It assumes that there is a functional SPH simulation where each fluid particle has a density. Then each frame the following steps are performed:

1. Splat points as spheres and determine depth values per fragment

2. Smooth the spheres based on curvature flow

3. Attenuate colors based on thickness

4. Add noise texture and advect throughout the simulation

5. Add foam

6. Render using Fresnel and Phong equations

3.1 Depth determination

It is desirable to determine depth of fragment in the simulation in order to know which fragments are part of the surface. In order to obtain these depth values, the points from the SPH simulation are splatted to discs and their fragments are subjected to a hardware depth test. This result is subsequently written to a depth buffer. Note that merely the depth value is splatted and the color and normal attributes are left unchanged as they will be changed in later steps.

3.2 Surface smoothing

Smoothing is a critical component in the paper. It is responsible for creating a smooth surface from a set of points so to avoid a blobby and jelly-like surface which is undesirable. The paper argues that this is a better approach than using Gaussian blurring because it performs better and ought to produce better results because there will be no silhouette blurring. This is achieved by translating points along the z-axis according the curvature flow.

Curvature flow is defined by the divergence of the normal by $2H = \nabla \cdot \hat{\mathbf{n}}$. Subsequently, depth values can be displaced every timestep based on the curvature in that point, as $\frac{\partial z}{\partial t} = H$.

To obtain normals, the point of which the normal needs to be determined is mapped back to a point in view by inverting the projection transformation. This point is called

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

where V_x and V_y are the viewport dimensions and F_x and F_y is the focal length in x and y component.

The normal is then obtained by

$$\mathbf{n}(x,y) = \frac{\partial \mathbf{P}}{\partial x} \times \frac{\partial \mathbf{P}}{\partial y}$$

Rigorous algebra shows that

$$\mathbf{n}(x,y) = \begin{pmatrix} -C_y \frac{\partial z}{\partial x} \\ -C_x \frac{\partial z}{\partial y} \\ C_x C_y z \end{pmatrix} z \qquad (1)$$

Here $C_x = \frac{2}{V_x F_x}$ and $C_y = \frac{2}{V_y F_y}$

Subsequently, the unit normal must be obtained. This is obtained by $\hat{\mathbf{n}}(x,y) = \frac{\mathbf{n}(x,y)}{|\mathbf{n}(x,y)|}$

Now that we have a unit normal we can define mean curvature as

$$2H = \frac{\partial \hat{\mathbf{n}}_x}{\partial x} + \frac{\partial \hat{\mathbf{n}}_y}{\partial y}$$

These values can be calculated fairly easy and can thus be used to modify the z-values in the visualization easily and fast using Euler integration in time.

3.3 Thickness

The thickness of a fluid is important because an object behind the fluid should change color according to the amount of fluid in front of it. This means that the distance from the camera towards the first opaque object has to be determined for each fragment.

3.4 Noise

In order to remove the appearance of an artificially smooth surface, noise has to be added. Since the simulation has moving particles in it, the noise has to advect with those particles. van der Laan et al. [2009] suggests to use Perlin noise Perlin [1985].

3

One octave of noise is added to each projected particle, this makes sure that the noise moves along with the fluid flow.

Also, foam can be added to the fluid by adding grey according to the magnitude of the noise.

3.5 Rendering

Finally, all intermediate results are merged together in a final texture that is rendered as fullscreen quad. The Fresnel equation is used to calculate the correct reflection and refraction vectors needed for the Phong Phong [1975] illumination model. This results in the following equation van der Laan et al. [2009]to determine the output color:

$$C_{out} = a(1 - F(\mathbf{n} \cdot \mathbf{v})) + bF(\mathbf{n} \cdot \mathbf{v}) + k_s(\mathbf{n} \cdot \mathbf{h})^{\alpha}. \quad (2)$$

F is the Fresnel equation, a is the refracted fluid color, b is the reflected scene color, k_s and α are both constant and represent the specular highlight. $\mathbf{n}$ is the normal along the surface, $\mathbf{h}$ is the halfangle between the camera and the light, and $\mathbf{v}$ is the direction pointing towards the camera.

3.6 Fit

When the initial requirements for the realistic water rendering are considered it seems that the curvature flow way is a good fit.

- Given enough iterations of the fluid surface, a surface will become smooth and will have no sharp discontinuities between particles.

- It is possible to generate an imperfect surface by adding a noise texture to the surface and attaching it to particles. This will distort the surface a little bit and will create an imperfect, and more natural surface

- It takes into account attenuation of colors based on the depth of the fluids and submerged objects. Moreover, foam can be added for an added realistic effect.

Besides that it can produce a natural rendering of a fluid surface, it is also fast. It can render a smooth surface from thousands of particles and still have an acceptable framerate. This is also an important quality for a simulation.

4 Implementation

The solution that has been proposed in the previous section is implemented using the C++ programming language in combination with the OpenGL API. To program on the GPU, CG is used as shading language.

The actual SPH fluid simulation is given and can be seen in figure 1, by implementing the multiple passes described in the previous section, a nature like fluid is expected. By creating Frame Buffer Objects (FBO), the results can be written to an off-screen rendering target. This is useful since we are using multiple passes. Also, multiple buffers can be attached to the FBO.

4.1 Depth determination

The depth at each pixel of each particle closest to the camera is determined using a combination of a vertex- and fragment shader. Each particle has a position in world space that is passed to the vertex shader. These particles are rendered as spheres to determine the correct depth values.

The vertex shader computes and passes the following properties of each particle to the fragment shader:

- Position of center in eye space.

- Position of center in screen space.

- Splat size.

- Splat radius.

The fragment shader uses this input to determine the depth at every fragment. To determine this depth, the splat is rendered as a sphere by discarding fragments that fall outside the sphere. From this, the normal from the center of the sphere towards its surface is determined by taking the difference of the current fragment position and the current particle center. Using this normal, the point is transformed to clip space, the z value of this position is the depth value. The depth values are then written to the depth buffer of the current FBO.

The depth component of each fragment can be visualized as grey value by setting the R, G and B components to the depth value. The results of this visualization can be seen in figure 2. It can be observed that the particles become darker when they appear closer to the camera.
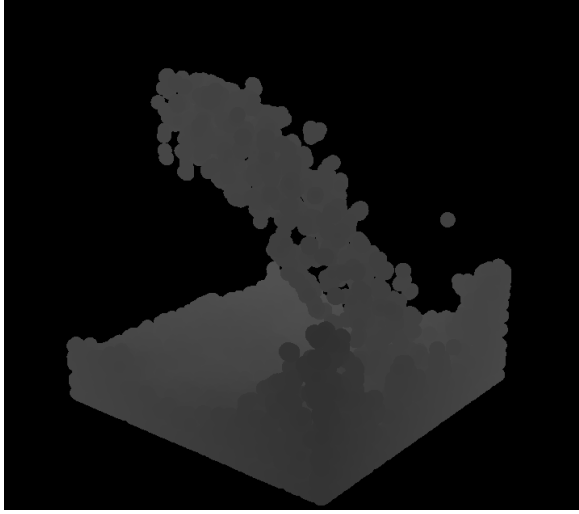
Figure 2: Depth component visualization

Algorithm 2 Surface smoothing pseudocode

```
calculateNormals();
for i := 1 to smoothSteps do
    smoothDepths();
    calculateNormals();
    flipTextures();
```

we can obtain the depth values of neighbours. Using these neighbours, the finite differences between depth values can be calculated in the x and y direction. Since we now have $C_x, C_y, \frac{\partial z}{\partial x}, \frac{\partial z}{\partial y}$ and z, we can calculate the normal according to equation 1. These normals are written to the normals texture. A visualization of the normals, done by assigning the normal to the color output, can be seen in figure 3.

4.2 Surface smoothing

Two fragment shaders are used for surface smoothing. The first fragment shader calculates the surface normals from the depth values and writes them in a texture. The second fragment shader smooths the depth values using curvature flow, using the surface normals computed in the first shader. These smoothed depth values are also written to a texture.

Since the smoothing happens in multiple steps, and textures can not be overwritten directly, multiple textures and FBOs are needed. The depth texture has to be updated, so an extra temporary depth texture is needed, the same holds for the surface normal texture. In each smooth step, one texture is used as input and the other is used as render target, the textures are switched after every step. The surface normals change when the depth values are changed, so these also need to be recomputed every smooth step. Listing 4.2 shows this process in pseudocode.

4.2.1 Normal calculation

The depth buffer is passed as input to the fragment shader, along with the uniform variables C_x and C_y. Since the depth buffer is in a texture,
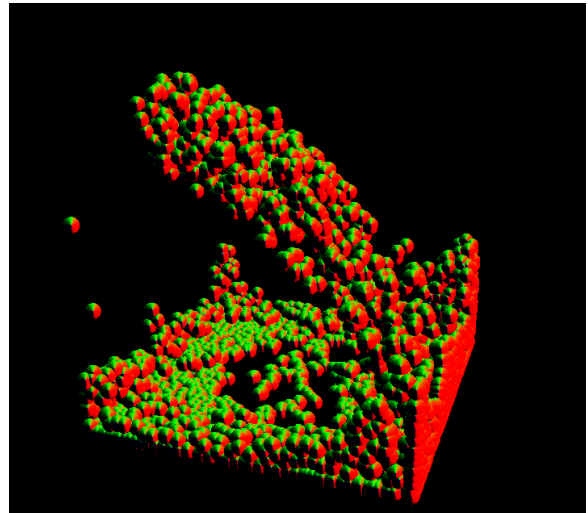

Figure 3: Visualization of the normals

4.2.2 Depth smoothing

Since we now have all surface normals in a texture, finite differencing can be used again to calculate the derivatives of the surface normals. The depth values are modified according to these derivatives and written to the depth texture.

4.3 Rendering

For the final rendering phase we implemented a fragment shader that produces a color for each fragment, which only has a depth component. We have a default color for water that is modified according to its surroundings based on shading. The end-results are projected onto a full-screen quad, which is the best option due to our framebuffer-objects.

The paper suggests to implement a combination of Phong shading and the Fresnel equations shading for displaying the surface.

Phong shading works by linearly interpolating the normals from the vertices across a surfacePhong [1975]. This results in a smooth surface of an object with correct specular highlights, as opposed to Gouraud shading which is not smooth and may not render specular highlights.

The Fresnel equations describe the the behaviour of light when it moves between media with different refractive indices. This helps visualizing water in a realistic way because the rendering can take refraction and reflection of a fluid surface into account in a realistic way.

Normally, computing the Fresnel equations would be a costly and difficult operation. To circumvent this, we have applied Schlick's approximation to determine the reflective Fresnel component.

5 Results

We have implemented the methods as described in the Implementation section. Our project is not quite done, but the first results are visible in figure 4.

Currently, splatting the points and determining the normals per fragment works. Also parts of the rendering implementation are complete, but refraction of the water is lacking. Moreover, thickness is currently not implemented, as well as noise and foam. Most notably lacking is the smoothing pass.

6 Workload division

All the programming was done in pairs so we'd value that the programming was 50-50 effort.

For the report, Bram focused more on problem definition, and implementation while Maarten fo-
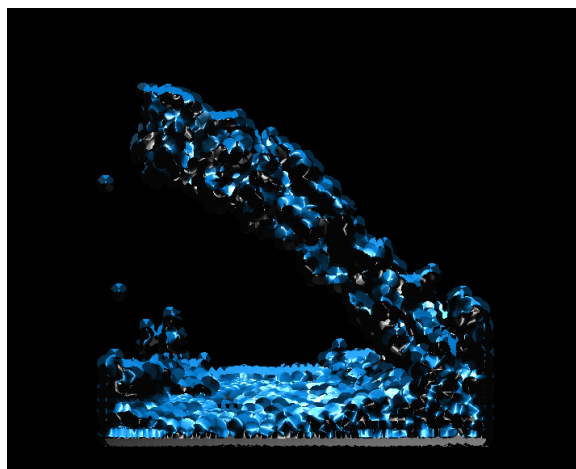

Figure 4: The SPH simulation with splatted point and Phong and Fresnel rendering

cused more on the solution as proposed by the paper and the introduction.

All in all, we feel that we both contributed equally to the entire final project.

References

Perlin, K. (1985). An image synthesizer. *ACM Siggraph Computer Graphics*, **19**(3), 287–296.

Phong, B. T. (1975). Illumination for computer generated pictures. *Communications of the ACM*, **18**(6), 311–317.

van der Laan, W. J., Green, S., and Sainz, M. (2009). Screen space fluid rendering with curvature flow. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, 91–98. ACM.

Zhang, Y., Solenthaler, B., and Pajarola, R. (2008). Adaptive sampling and rendering of fluids on the gpu. In *Proceedings of the Fifth Eurographics/IEEE VGTC conference on Point-Based Graphics*, 137–146. Eurographics Association.