Approaching Fluids – in Games

Jonas A. Johansson

Abstract.

Over the past years there has been an increasing interest in applying fluid simulation in video games. Before, the arithmetic requirements were too high for real-time applications, but with the more powerful hardware available today – along with algorithmically optimizations – games can now use fluids to improve the behaviours and visuals of effects such as smoke and water. The purpose of this document is to describe some of the more popular approaches – including Eulerian and SPH – to approach real-time fluids in games. It also describes some of the techniques available to visualize the data with realistic results.

Categories: *Fluids, 3D Particles, Particle System on GPU*
Keywords: *SPH, Eulerian, Langrangian, GPGPU, OpenCL*

1. Introduction.

Fluid-simulation has traditional mainly been used in areas where real-time execution has not been a requirement, such as movies and science projects. This is because of the expense of updating the fluids behaviours – which involves a lot of calculations of complex and expensive formulas – computations that mostly needs to be done several times each second to produce realistic results. The full-scale computational power to produce, for instance, a glass of water is actually tremendous. Thus, every step to reduce that cost is potentially a great difference between getting reasonable fps, frames per second, or not. However, it's not only updating the fluids behaviours from its physical characteristics that is a challenge – but also to use the calculated data to produce a fulfilling visual result.

2. Behaviour processing

There are different approaches – both in theoretical perspective and in implementation – in how to simulate the behaviours of fluids. However, all of them have in common that they rely on the fact that a "piece" of water is influenced by its neighbours. The problem is to find a way to make the computations of that piece's physical characteristics, while reducing the arithmetics throughput needed for the simulation. How this piece is represented in implementation depends on the used approach: the Eulerian one represents the fluids using grids (seeing the water as cells) – while the Lagrangian one considers the water as a set of particles. Both these have different kinds of pros and cons.

2.1 Navier-Strokes equations

In physics, the Navier-Strokes equations, describes the physical behaviours (motion) of fluids. They are named after Claude-Louis Navier (1785-1836) and George Gabriel Stokes (1819-1903) and consider work derived by Newton's second law of motion.

The equations are mainly used to calculate velocities rather than positions, thus results performed by these equations are mainly referred to as "velocity fields" or "flow fields".

The equations are widely used in scientist-related projects, such as weather simulation (modelling weather and oceans), water flow in pipes, and designing aircrafts and cars. Because of their complex nature, the Navier-Strokes equations aren't very suitable for video games, but could be used to pre-calculate data if desired.

2.2 Eulerian fluid simulation approach

The Eulerian approach, based on the Euler equation (fluid dynamics) by the mathematician Leonhard Euler (1707-1783), is commonly concretized as a grid-based approach. It is typically used when simulating larger areas of water, such as seas and oceans.

The equation considers mass, momentum and energy, corresponding to the Navier-Strokes equations, according to the following formula:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0$$

$$\frac{\partial \rho \mathbf{u}}{\partial t} + \nabla \cdot (\mathbf{u} \otimes (\rho \mathbf{u})) + \nabla p = \mathbf{0}$$

$$\frac{\partial E}{\partial t} + \nabla \cdot (\mathbf{u}(E + p)) = 0, \qquad (1)$$

Where ρ is the fluid mass density, u the fluid velocity vector, with components u, v, and w, E = ρ e + ½ ρ (u2 + v2 + w2) is the total energy per unit volume and p is the pressure.

The approach is to implementing a grid with a set of cells and then apply the Eulerian formulas to calculate how a change in height of a cell affects (and is affected by) its neighbours – and adjust them accordingly.

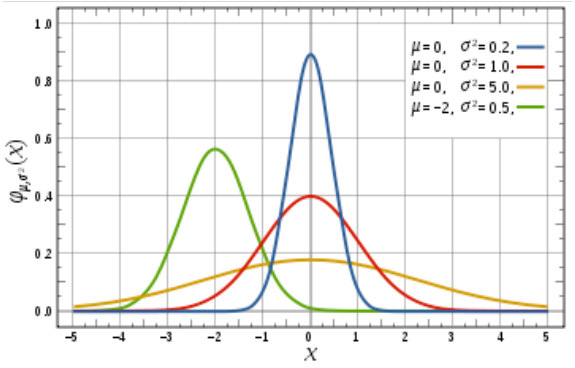
The problem with the Eulerian approach is the lack of possibility to have several height values – since the grid stores the height value for a given point (making it not fully 3D), thus making it unsuitable for sprays and splashes, and surfing/overturning waves.

In their paper on Real-Time Eulerian Water Simulation Using a Restricted Tall Cell [CM2011] Chantanez and Müller proposed a method to deal with the issues of grid-based water (such as lack of overturning waves). Through using a hybrid grid presentation composed of a regular cubic cells on top of a layer of tall cells, one can reduce the computational expenses by reducing the resolution on areas which aren't seen by the viewer while maintaining the appearance by the surface. The proposed approach is to represent the fluids in two different data structures in the same time: tall cells (the normal grid cells) below the surface and cubic cells on a top layer above the tall cells. The tall-cells represents the height of the cells, while the cubic cells are used to create additional detail and 3d appearances at the surface, by making several cells representing several layers of water. For the tall cells, the quantities like velocity, pressure etc, are stored at the center of the topmost and the bottommost sub cells, while the cubic cells store these quantities at the center.

The key is to parallelly solve the Euler equations on both the height field columns and the cubic grid cells, which makes the surface influenced by the lower water levels, but with additional calculations by the surface to improve visualizations.

2.3 Smoothed particle hydrodynamics (SPH)

SPH is a mesh-free Langrangian method (i.e. particle based). The particles have a spatial distance (referred to as "smoothing length") which properties are defined by a kernel function – a function of two variables that defines a integral transform – meaning that the physical property of a particle is defined by the properties of the particles within a specified range of the kernel. One of the most used kernel functions are the Gaussian function – which smoothes out the influence non-linearly over the given distance – as shown in the image:



For instance, when applied to temperature of liquids, a given particle at a given position depends on all particles within a radial distance, using the SPH formula:

$$A(\mathbf{r}) = \sum_j m_j \frac{A_j}{\rho_j} W(|\mathbf{r} - \mathbf{r}_j|, h), \qquad (2)$$

Where m_j is the mass of particle j, A_j is the value of the quantity A for particle j, ρ_j is the density associated with particle j, $\mathbf{r}$ represents the position and W is the kernel function mentioned above.

The density ρ_j can be computed with Eq. (2) giving:

$$\rho_i = \rho(\mathbf{r}_i) = \sum_j m_j W(\mathbf{r}_i - \mathbf{r}_j, h), \qquad (3)$$

An issue with SPH is its restriction to liquids with the same physical characteristics. With their paper, on particle based fluid-fluid interaction [MSKG2005] Müller, Solenthaler, Keiser, and Gross, proposed a set of extensions to the regular SPH method to solve this matter, allowing simulation of the phenomena of multiple fluids with different particle types. In the common SPH approach, many attributes which are identical for all particles are stored globally, e.g. the particle mass

m, the density p etc. In their approach each particle carries all those attributes individually, in addition to a few new. This also allows the introduction of air-based fluids – that can be used to simulate air bubbles in liquids.

The main advantage of SPH is its ability to collide with the surrounding terrain – which makes it, unlike Eulerian approaches, very suitable for sprays and splashes. However, it is less suitable for large water areas because of the particle resolution needed to maintain visual appearance.

2.4 SPH on GPU

The last few years it has become very popular to accelerate data computations by using the graphics processing unit (GPU) – a technique called "general-purpose computing on graphics processing units", GPGPU. The GPU is designed to process resembling data quickly by using parallelism, thus creating a huge advantage when working with particles – or in other ways resembling data.

In their paper [HKK2007], Harada, Koshizuka and Kawaguchi, investigated the SPH algorithm performance when fully implemented on the GPU, using OpenGL while storing data in textures. They noted a higher speed increased, of the proposed method compared to the CPU, central processing unit, implementation, the more particles used in the simulation. In the highest amount of simulated particles (about 4,194,304), the GPU implementation was roughly 28 times faster than the corresponding CPU implementation. However, since the particle data needs to be stored in the video memory, about 600 MB was needed, thus leaving less memory on the GPU for other kind of geometry, textures etc.

Traditionally, this field of computational power has only been disposed by using rendering APIs, application programming interfaces, such as OpenGL and DirectX. However, the last years a new range of techniques has become available to access that power more straightforwardly – for instance by using Direct Compute or OpenCL. However, even though there're new ways of manipulating the data on GPUs, it is mostly more complicated to implement algorithms on GPUs than CPUs – because without sufficient use of the parallelism, one can end up getting worse performance than using the CPU for the calculations.

3. Visualisation

When the physical calculations (behaviours) for the frame have been determined, additional management may be required to mimic the fluids visual appearance. Mostly, the fluids data are too complex, or in other ways unsuited, for rendering straightforwardly, thus adding additional requirements to prepare the data for visualization. Luckily, there're several techniques, both for Eulerian and Lagrangian fluids, to obtain more render-friendly data and present it on the screen with a desirable appearance.

3.1 Marching cubes

In their paper [LC1987], Lorensen and Cline, proposed a computer graphics algorithm, namely "marching cubes", for extracting a polygonal mesh of an isosurface from a three-dimimensional scalar field (essentially 3d points) – sometimes called voxels. The approach is basically to generate a full-scale 3d mesh from the point cloud.

The process can be made on the GPU, but it's considered expensive, compared more modern techniques, according to the presentation Screen Space Fluid Rendering for Games [SG2010].

Although the Marching Cube algorithm is one of the most popular approaches to generate meshes for iso-surfaces of scalar fields, there're some disadvantages of this approach – especially with real-time applications in mind. The algorithm itself is camera-independent, meaning that many invisible triangles and surface details are generated. Because of the algorithms way of operating in three dimensions, but producing results for rendable 2d surfaces, it has become more and more common to find other, more cheap ways of rendering fluids.

3.2 Screen space meshes

An approach which has become more popular the last few years is to generate a screen space mesh from the particle data [MSD2007]. Because this approach mainly operates in 2D space, it avoids a lot of the expenses that occurs in the Marching cubes algorithm, thus addressing its main inadequacies.

The first step is to generate a depth map and calculate the internal and external silhouettes of the surface, in screen space. Using this data it is possible to construct a 2D screen space triangle mesh with a technique that is derived from the marching squares approach (which is similar to the marching cubes). The result-mesh is transformed back to 3D space for rendering with desired visualizations techniques such as occlusions, reflections, refraction, and other shading effects.

The algorithm acts in the following steps:

1. Setup a regular depth map
2. Find internal and external silhouettes
3. Smooth depth values

4. Generate a mesh (in 2D) using Marching Squares
5. Smooth silhouettes
6. Transform the mesh back into world/3d space
7. Render the mesh with desired shading techniques

The main difficulty with this approach is the second step – to find the silhouettes to generate the desired mesh.

3.2.1 Silhouettes

The depth map, which stores depth values at each node of the grid, is generated from scratch at the beginning of each frame. First the depth values are initialized with infinite values. Then the algorithm iterates through the particles in the point cloud twice. In the first phase, depth values are set. In the second phase, additional depth values are generated where the silhouettes cut the grid. In both these phases the particles have to be transferred from world space to screen space.

During the detection of silhouettes, the algorithm iterates through the particles a second time, where only grid edges which connect depth values that are further apart than a given number, are considered to be “silhouette edges”. The key is to find an additional node on each silhouette edge. Each node is located between the adjacent nodes of its silhouette edge and stores the depth value of the front layer.

3.2.2 Mesh generation

The next step is to generate vertices and triangles for the screen space mesh, by using the nodes with an initialized depth value which is finite. Additional nodes are generated where silhouette edges have only one initialized node.

Lastly, the mesh is transformed back to world space for rendering, which can be done quickly using the depth map, in order to render it with reflections and refractions from the environment.

3.4 Screen space fluid rendering

In their paper on screen space fluid rendering with curvature flow [LGS2009], Wladimir J. Van der Laan, Simon Green and Miguel Sainz, proposed a method derived from the screen space meshes approach, but with the extension of using the generated depth buffer directly for rendering – without generating an intermediate mesh. The proposed method is targeting mainly two issues: firstly, it avoids polygonization which does not map to graphics hardware in a straightforward way, and secondly, it addresses the issue of jelly-like appearances of screen space meshes – through adding surface details on smaller scale than the particles themselves.

The first steps of the approach is very resembling of the one used in the screen space meshes approach. Firstly, the surface depth is determined by rendering the particles as spheres, and retaining the closest value – essentially a depth map. Secondly, the result is smoothed in screen-space to avoid jelly-like appearance. The problem with the second step is to find a suitable kernel for the operation – the most common ones all have different kinds of issues making them unsuited for the task. Gaussian filters causes blur over the silhouette edges; Bilateral filters preserve edges but are computationally expensive. The solution is to use something called “Curvature flow”, which is a method, invented to smooth out sudden changes in curves. Thirdly, a procedure to simulate the appearance of thickness is applied, by generating a new depth map – but with the difference that the fragment shader outputs the thickness of a particle instead of its depth value. The rendering is made with point spheres. Depth testing is still enabled though, so that only the particles closest to the camera are calculated. Lastly, all the intermediate results are combined in the rendering step, which is made by rendering a full-screen quad. The light characteristics on the fluid are determined by the normals given by finite differences of the surface depth from step one. During the rendering, Perlin noise is added to create additional detail to the fluid.

4. Future work

Due to the superior performance of the GPU implementations of the algorithms, it is likely that more algorithms will be disposed that emphasizes the GPUs incredible ability to make liquid calculations. Compared to the CPUs, GPUs have the past years increased the arithmetical throughput more rapidly, thus making them even more attractive for these kinds of simulations in the future.

Another area that will probably become more explored is the way to optimize the well-used approaches even further, by introducing new data structures, mathematical optimizations, and/or to use new hardware to accelerate the calculations even further.

It is also likely that more research will be made to improve the possibility of mixing several fluid hybrid simulation techniques (as in the Restricted Tall Cell example) – for instance to add splashes to grid-based fluids.

5. Conclusions

There are several ways – depending on the projects needs – to implement fluids in games. Eulerian is very useful to simulate larger areas of water; Langrangian is convenient for sprays and splashes. However, whatever approach one decides to use, it seems to be possible to gain a lot performance by accelerating the computations using the GPU.

Fluid simulation is a very hot topic with new and related research areas arising all the time, but fluids have yet only made a brief introduction in video games – but it will likely change in the close future. Thanks to continuous development of hardware, large-scale fluid simulation will become more available to games as every day passes by, thus making the future of fluids very interesting.

References

[CM2011] N. Chentanez, M. Müller, Real-Time Eulerian Water Simulation Using a Restricted Tall Cell Grid, ACM Transactions on Graphics (SIGGRAPH 2011), 30(4), pp 82:1-82:10

[MSKG2005] Matthias Müller, Barbara Solenthaler, Richard Keiser, Markus Gross. Eurographics/ACM SIGGRAPH Symposium on Computer Animation (2005), pp. 1–7K. Anjyo, P. Faloutsos (Editors)

[HKK2007] Takahiro Harada · Seiichi Koshizuka · Yoichiro Kawaguchi. Smoothed Particle Hydrodynamics on GPUs

[LC1987] William E. Lorensen, Harvey E. Cline: Marching Cubes: A high resolution 3D surface construction algorithm. In: Computer Graphics, Vol. 21, Nr. 4, July 1987

[SG2010] Simon Green. Screen Space Fluid Rendering for Games". GDC 2010.

[MSD2007] Matthias Müller Simon Schirm Stephan Duthaler. Eurographics/ ACM SIGGRAPH Symposium on Computer Animation (2007) D. Metaxas and J. Popovic (Editors)

[LGS2009] Screen Space Fluid Rendering with Curvature Flow. Wladimir J. van der Laan, NVIDIA, Rijksuniversiteit Groningen. Simon Green, NVIDIA. Miguel Sainz, NVIDIA