GPU-driven Real-Time Lagrangian Fluid Simulation using Smoothed Particle Hydrodynamics TNM085 Group 7

Oskar Ankarberg
Rickard Lindstedt
Max Jonsson
Michael Novén
Michael Sjöström

Linköping University

March 29, 2015

1 Objective

To create a real-time fluid simulation in three dimensions using Navier-Stokes equations for fluids in combination with Smoothed Particle Hydrodynamics, rendered and calculated on the GPU using metaballs with the marching cubes algorithm.

2 Definitions

Smoothed Particle Hydrodynamics (SPH) A way of simulating fluids in terms of particles and how they affect each other.

GPU The Graphics Processing Unit of a computer.

CPU The Central Processing Unit of a computer.

Metaballs Organic looking objects defined as a function in N dimensions.

Marching cubes A bitwise volume rendering technique.

3 Background

This project is part of a course held at Linköpings University in Modelling and Simulation. The group members are all M.Sc students in Media Technology at Linköpings Institute of Technology with the ambition of creating a memorable project and to deepen their knowledge in fluid simulation, volume rendering and modern GPU programming.

4 Mathematical model

4.1 Navier-Stokes equations

The most commonly used algorithm for describing the properties of a moving fluid is the Navier-Stokes Equations (1).

$$\underbrace{\frac{\partial \rho\phi}{\partial t}}_{\text{Accumulation}} + \underbrace{\nabla \cdot (\rho \mathbf{u}\phi)}_{\text{Convection}} = \underbrace{\nabla \cdot (\Gamma \nabla \phi)}_{\text{Diffusion}} + \underbrace{S_\phi}_{\text{Source}} \tag{1}$$

This approach is based on the conservation of forces where each term respresents a physical mechanism of the fluid. The total conserved quantity will thus evolve based on the contribution of each separate term in the equation and will move with the flow.

The orignal Navier-Stokes equations can be specialized to represent different attributes of a fluid such as heat, momentum and mass. In this case, the conservation of momentum and mass are governing for an incompressible fluid and are described in Equation (2) and (3).

$$\frac{\partial \mathbf{v}}{\partial t} = -\frac{1}{\rho}\nabla p + \upsilon \nabla^2 \mathbf{v} + \mathbf{g} \tag{2}$$

$$\frac{\partial \rho}{\partial t} = 0 \tag{3}$$

The parameters $p, \rho, \upsilon, \mathbf{v}$ stands for pressure, density, viscosity and velocity. Pressure differences implies changes in velocity, this means that the pressure gradient ∇p will determine what way the particles will go. The viscosity term $\nabla^2 \mathbf{v}$ represents the loss of energy due to internal friction and $\mathbf{g}$ is the gravity force.

5 Physical model

5.1 Eulerian fluid

There are two different ways of approaching the fluidsimulation problem. One approach is where the density of predefined voxels is checked and the fluid is simulated in terms of pressure at the sides of each voxel. The result of this method of simulation is known as a *Eulerian fluid*.

5.2 Lagrangian fluid

The other approach, which is the one used in this project, is to use a particle-based model where ideally each and every particle affects the others. This is known as a *Langrangian fluid* and is most commonly implemented with Smoothed Particle Hydrodynamics, hereinafter referred to as SPH.

5.2.1 Smoothed Particle Hydrodynamics

The particle hydrodynamics part of SPH is where the different particles interact with each other according to some physical model of a fluid. A *smoothing kernel* makes the fluid act properly by measuring how much surrounding particles will affect the current particle , this is explained in section 6.2. For each term in the Navier-Stokes equations a unique smoothing kernel is used for approximating the given contribution.

6 Implementation

6.1 Materials

The simulation is implemented in C++ using OpenGL[1] for simple rendering, GLFW[2] and GLEW[3] for window handling and management and lastly OpenCL[4] for implementation on the GPU.

6.2 Calculation of forces

In order to be able to implement each term in the momentum conservation equation, the gradient and laplace operators will be used to determine the direction of the pressure and the viscosity. These directions can be seen as forces that is added together to calculate the new acceleration of each particle with the help of Newtons second law.

$$\mathbf{F}_{tot} = \mathbf{F}_{pressure} + \mathbf{F}_{viscosity} + \mathbf{F}_{gravity} \tag{4}$$

These separate forces of every particle p_i depends on the weighted sum of physical values for every neighbouring particle p_j. Equations (5) and (6) show how to calculate the forces.

$$\mathbf{F}^i_{pressure} = -\sum_j m_j \frac{p_i + p_j}{2\rho_j} \nabla W_{pressure}\left(\mathbf{r}_{ij}\right) \tag{5}$$

$$\mathbf{F}^i_{viscosity} = v \sum_j m_j \frac{\mathbf{v}_j - \mathbf{v}_i}{\rho_j} \nabla W_{viscosity}\left(\mathbf{r}_{ij}\right) \tag{6}$$

[1] https://www.opengl.org/
[2] http://www.glfw.org/
[3] http://glew.sourceforge.net/
[4] https://www.khronos.org/opencl/

The weighting functions $\nabla W_{pressure}$ and $\nabla W_{viscosity}$ depends on the effective distance h to each neighbouring particle p_j [2]. The weighting functions for the pressure and viscosity forces are shown in Equations (7) and (8).

$$\nabla W_{pressure}(\mathbf{r}) = \frac{45}{\pi h^6}(h - |\mathbf{r}|)^3 \frac{\mathbf{r}}{|\mathbf{r}|} \tag{7}$$

$$\nabla W_{viscosity}(\mathbf{r}) = \frac{45}{\pi h^6}(h - |\mathbf{r}|) \tag{8}$$

Here $\mathbf{r}$ represents the euclidian distance between the current particle and one of its neighbours. The pressure term p can be calculated with the constitutive Equation (9)

$$p_i = p_0 + k(\rho_i - \rho_0) \tag{9}$$

where p_0 and ρ_0 are the rest pressure and density. The computation of the density of each particle is shown in Equation (10)

$$\rho_i = \sum_j m_j W(\mathbf{r}) \tag{10}$$

where the corresponding weight function of the density is described in Equation (11).

$$W(\mathbf{r}) = \frac{315}{64\pi h^9}(h^2 - |\mathbf{r}|^2)^3 \tag{11}$$

6.3 Simulation

When the forces of each particle have been calculated, the final step is to use these forces to simulate the moving particles. Euler's implicit method can be used to calculate the new velocity and position of each particle with a fixed timestep. Newton's second law $F_{tot} = ma$ is applied to get the acceleration, the Equations can be seen in (12) and (13).

$$\mathbf{p}(t)' = \mathbf{p}(t) + \mathbf{v}(t)t + \frac{at^2}{2} \tag{12}$$

$$\mathbf{v}(t)' = \frac{\Delta \mathbf{p}(t)}{t} \tag{13}$$

6.4 Pseudo Code

This pseudo code is representing each step of the Navier Stokes equations in the SPH. It gives a better understanding of how the equations are implemented in C++.

```
for each Particle p do
    set corresponding cell index to p;
end
for each Particle p do
    get corresponding cell index to p;
    get every neighbour cell containing particles n;
    if n distance to p ≤ effective distance h then
        calculate density;
        calculate pressure;
    end
end
for each Particle p do
    get corresponding cell index to p;
    get every neighbour cell containing particles n;
    if n distance to p ≤ effective distance h then
        calculate pressure force;
        calculate viscosity force;
    end
    add gravity force;
end
for each Particle p do
    move particle p;
end
```

Algorithm 1: *SPH calculation using neighbouring cells*

6.5 Optimisation

In order to optimise the implementation, a neighbour-finding algorithm should be introduced. As only the closest surrounding particles affect each other, the area containing particles taken into consideration is limited. This area, or space, will hereinafter be referred to as a *bucket*.

6.5.1 Bucket of neighbours

The space where the particles can move is divided into a grid of cells (two dimensions) or voxels (three dimensions). The bucket, or the space containing each neighbour, for each particle consists of the particles current cell or voxel along with all surrounding cells or voxels. This means that for each particle only the closest ones, up to a distance of

$$d_{2d} = \sqrt{(2c_w)^2 + (2c_h)^2} \tag{14}$$

will be taken into account when calculating the different terms of the Navier-Stokes equations, where c_w is the width and c_h the height of each cell.

6.5.2 GPU implementation

Both the CPU and the GPU have a limited number of threads to use and both can use all their threads at the same time. A modern graphics card has about 100 to 1000 cores and each core can run a couple of threads each, which makes it theoretically possible to run several thousand threads simultaneously. A modern CPU on the other hand has 2 to 8 cores and most often only run one thread per core. The CPU cores are more powerful which makes the CPU superior in making complex manipulations on a small sets of data, meanwhile the GPU is better on doing simple manipulations to large sets of data. Addition, subtraction, multiplication and division are examples of simple manipulations. However, to fully benefit from the large number of threads on the GPU, the computations must be parallelized.

Generally the GPU handle all graphics related computations in a computer. In computer games the GPU must compute a large amount of calculations to be able to supply the user with many frames each second. To do this the GPU distributes the computations to be done on all available cores and each core executes instructions on pieces of data. This is the parallelization method aimed for in the GPU implementation of this project.

The implementation is done using OpenCL. OpenCL is a framework for cross-platform, parallel programming which allows the user to access for example the GPU.

6.6 Rendering

The most basic way of rendering is to draw the particles as squares, circles, boxes or spheres. This however resembles the look of a ball pool. To further make the SPH look like a fluid and the particles to act more realistic, a different rendering algorithm must be applied. This is where the method called metaballs is introduced.

6.6.1 Metaballs

In metaballs, each particle is defined as a function depending on the square of the radius of the particle divided by the distance between the sphere and each visible point as can be seen in Equation (15) below.

$$intensity_{2d} = \frac{r^2}{(x - x_0)^2 + (y - y_0)^2} \tag{15}$$

The resulting function is shown in Figure 1.

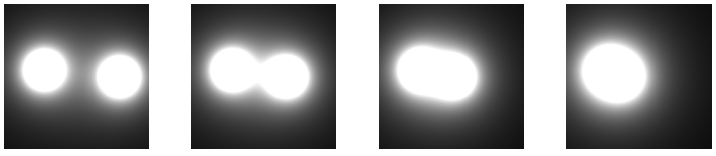

Figure 1: *Metaballs function*

Each solid part will be determined by a predefined threshold. Due to the nature of the function, two particles close to each other will attract the other one and they will upon connecting merge to one bigger sphere, giving an organic look to it, as can be seen in Figure 2.

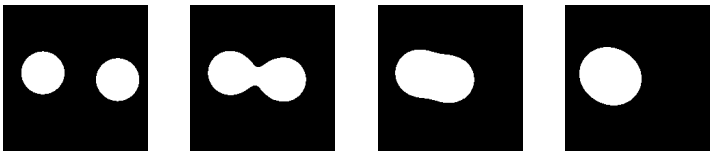

Figure 2: *Metaballs with thresholding*

Upon plotting the intensities of the metaballs function, seeing the intensity as a function of X and Y, and by thresholding, it is easier to visualise what the function looks like. The result can be seen in Figure 3.

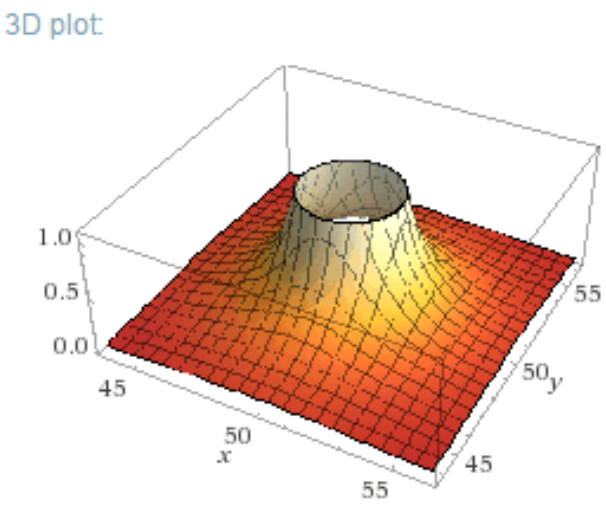

Figure 3: *Metaballs intensities plotted as function of x and y, clipped at given threshold.*

However, while the metaballs method is the underlying principle for making the particles look more fluid-like, what makes it look good is the rendering technique known as marching squares (two dimensions) or marching cubes (three dimensions).

6.6.2 Marching squares

In two dimensions, as mentioned earlier, we speak of marching squares which is a bitwise method for determining how each part of a given area should look. Firstly, the total area is divided into smaller squares. For the position of each corner of each square, the metaballs function for each particle will be calculated. The value of each corner is then compared to the predefined threshold and if the threshold is exceeded, that corner will contribute to the look of the current square.

6.6.3 The bitwise sum

The number of corners will reveal how many different cases a square or cube can have. Number of possible cases relates to 2^n where n is the number of corners. In our marching squares case, the top left corner will contribute with 2^0, the top right corner with 2^1, the bottom right corner with 2^2 and lastly the bottom left corner with 2^3. This bitwise sum results in 16 different cases of each square as can be seen in Figure 4. An example of how the cases are calculated is provided in Figure 5.

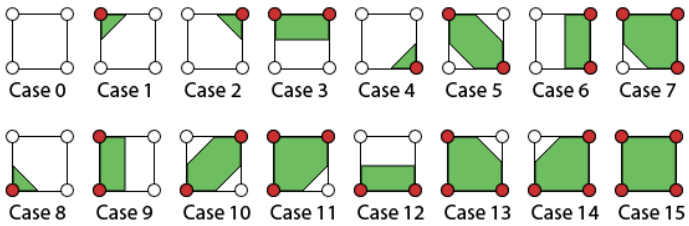

Figure 4: *The different cases of marching squares.*

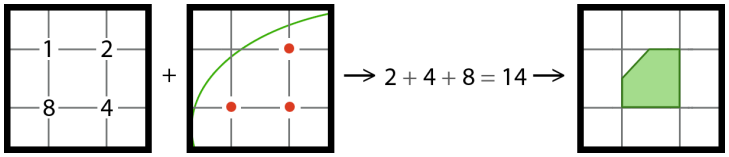

Figure 5: *How the look of a square is computed.*

6.6.4 Interpolating the triangles

Once the look of the square is decided the corners of each triangle will be interpolated with regards to the value in the corner of the square. The value is used as a weight for interpolating each point of the triangle to properly blend with surrounding squares.

The different steps of the marching squares algorithm are visualised in Figure 6.

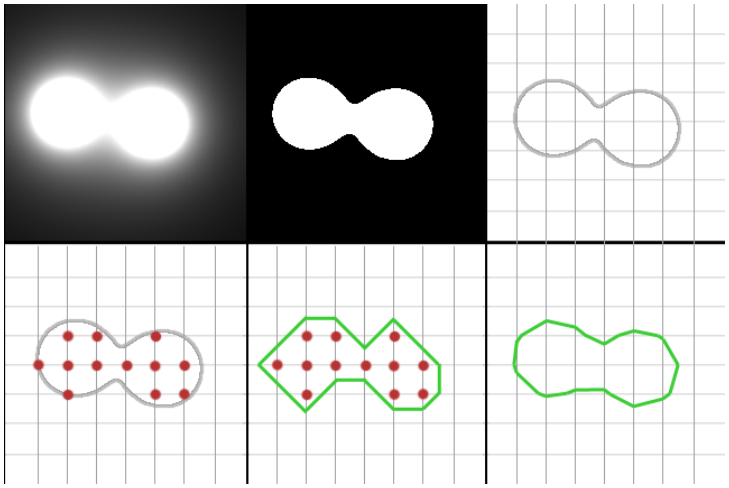

Figure 6: *From top to bottom, left to right: heightmapping, thresholding, the contours of the thresholding, corners within the metaballs, mapping of triangles, interpolation.*

6.6.5 Marching cubes

The marching cubes algorithm acts basically the same way as its little brother [1], marching squares, only in three dimensions instead of two. The main difference is that instead of cells we look at voxels which have eight corners. This gives us $2^8 = 256$ different possible cases. To make this bearable from a computational point of view a lookup table must be defined, containing the look of each of the 256 different cases. Each triangle making up the overall look of the voxel will be

defined by vertices which in turn is set as the three dimensional position of each corner of the current voxel. To map the vertices beneath the surface to the intersecting edges a predefined binary table should be introduced. If one intersection is found, the linear interpolation method will be applied to estimate where on the edge the intersection occurs.

7 How to run the simulation

7.1 Windows

To run the simulation on a windows computer several programs and libraries are needed. In this project Visual Studio[5] was used as development environment. OpenCL for Intel integrated graphics or NVIDIA CUDA[6] needs to be installed as well as the GLFW and GLEW library. To run the project a couple of settings must be made in project properties. Under "C/C++" and "General" include the GLFW and GLEW include folders under "Additional Include Directories". Under "Linker" and "General" add the GLFW and GLEW library under "Additional Library Directories". The last step is to add "opengl32.lib", "glu32.lib", "glew32.lib" and "glfw3.lib" under "Input" and "Additional Dependencies".

7.2 OS X

Running the simulation on OS X is easier. As development environment Xcode[7] was used. Under "Build Phases" and "Link Binary With Libraries" add the frameworks "IOKit", "CoreVideo", "Cocoa", "OpenGL" and "OpenCL" and the library "libglfw3.3.0.dylib".

8 Results and Conclusions

The final project consists of two softwares - one in two dimensions using marching squares as rendering which can be seen in Figure 7. The other in three dimensions rendered using marching cubes and basic Phong shading, see Figure 8.

The result is a fluid that acts like real water. The metaballs and marching squares/cubes give it a correct look and it runs fairily well with up to about 1000 particles. In the three dimensional case it is possible to flip and turn the container, making the fluid move around.

Between 30 and 60 frames per second is reached during simulation with up to 1000 particles which is enough to make it run smooth and look good.

[5]https://www.visualstudio.com/
[6]http://www.nvidia.com/object/cuda_home_new.html
[7]https://developer.apple.com/xcode/

Figure 7: *Marching squares*

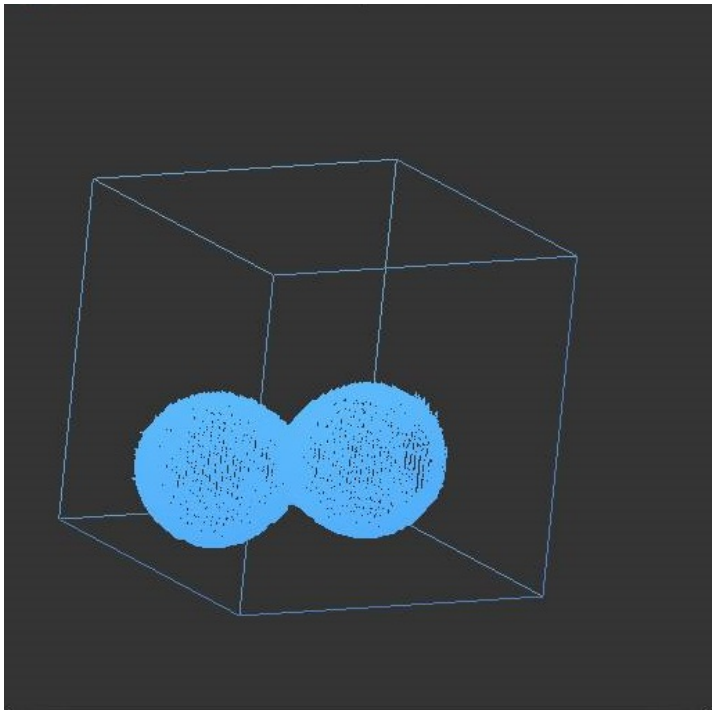

Figure 8: *Marching cubes*

9 Discussion

By adding the interpolation step in the marching squares/cubes algorithm, the resolution (number of voxels for the entire space) is the only thing limiting the resolu-

tion of the metaballs. Of course, the more voxels the more computational power is required and thus when it comes to real-time simulation a lower resolution is preferred. This is, however, compensated by the interpolation step of the Phong shading making the edges the only thing that may look a bit odd.

The group encountered a problem that was never solved. The calculations done in the software seems to differ in speed depending on which operating system that is being used. On a MacOSX machine the software runs smoothly but the exact same software is almost ten times slower on a Windows machine. Since this problem is out of the groups knowledge it was accepted the way it was.

References

[1] Watt, A. 3D Computer Graphics - third edition. Harlow. Pearson Education; 2000.

[2] Harada, T. Koshizuka, S. Kawaguchi , Y. Smoothed Particle Hydrodynamics on GPUs [Internet]. 2007 [updated 2007 Feb 10; cited March 29, 2015]. Available from: http://inf.ufrgs.br/cgi2007/cd_cgi/papers/harada.pdf