SPH Fluid Simulation with Metaballs

DD2323 Project Report

Alexander Hjelm
alhjelm@kth.se

Tsz Kin Chan
tkch@kth.se

May 23, 2019

1 Summary

Hippety hoppety, Women are property! ¿:)

This report will present the theory behind the metaballs rendering technique and show how it can be implemented as a real time simulation.

2 Introduction

Metaballs are soft, organic-looking 3D objects that appear to blob together when they are very close, and can be used to simulate dynamic fluids if using many particles on a large simulation domain. The metaballs rendering technique was invented by Jim Blinn in the early 1980s, and has been a very common demo effect since the 1990s.

The metaballs model is defined as a 3D isosurface, and can be rendered using the same methods that are common to isosurfaces.

The most common methods for rendering metaballs are ray-tracing for still image and animations, and the marching cubes algorithm for real time. https://steve.hollasch.net/cgindex/misc/metaballs.html

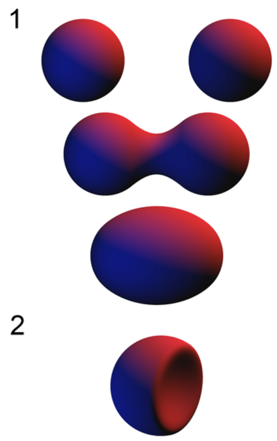

Figure 1: A conceptual visualisation of how metaballs work. 1 shows two metaballs gradually merging, and 2 shows the influence of a negative metaball on a positive metaball.
https://en.wikipedia.org/wiki/File:Metaballs.png

Using low-level graphics programming, we have programmed and rendered a real time, dynamic fluid using the metaballs with the marching cubes technique. The fluid particles use a basic physics model to collide with each other and the environment.

3 Theory

3.1 Metaball isosurface model

A metaball is an isosurface in 3D space. Define a function $f(x, y, z)$, which takes as input a set of coordinates in 3D space, and returns a floating poing value that represents the influence of the function on that point. When we have such a function, we can sample it at even or random intervals to determine which points belong inside the surface and which belong outside it. This is simply a matter of comparing wether the influence at a point is greater than a fixed threshold or not. We can then use a number of different rendering techniques to create a presentation of this 3D surface, as we shall see in the next section.

For now, let us look at the metaball surface model in particular. The most common isosurface fucntion for the metaball model is inverse quadratic:

$$[f(x, y, z) = 1.0/(x^2 + y^2 + z^2)]$$

This function describes a field where each ball has an influence point with quadratic falloff. For any point $\vec{p}$ in 3D, and for any ball's position $\vec{b}$, the influence of that ball on the point $\vec{p}$ will be proporitonal to $|\vec{p} - \vec{b}|^2$. This function is also used to model the strength of electrical fields in electromagnetics, which is why we choose to label it as the metaball potetial field function. [http://www.geisswerks.com/ryan/BLOBS/blobs.html]

If one were to draw the resulting field of a single metaball around the origin, it might look like figure 2, where the brightness of the color indicates the influence of the field at that point.

A similar model for two metaballs, with their potential fields partially overlapping in different stages, is shown in figure 3.

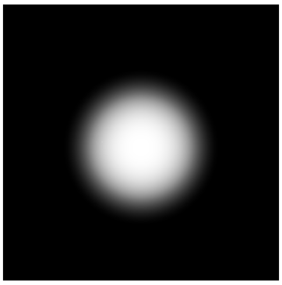

Figure 2: A concept of the inverse square function in 2D

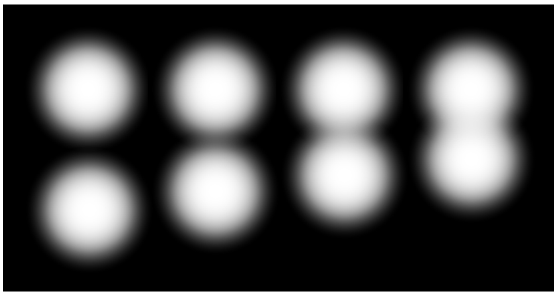

Figure 3: A concept of 2 metaballs in 2D. Every frame shows the two balls moving closer to each other.

As you can see, the area between the two balls becomes gradually lit up. The intersection between two potential fields becomes brighter as the two balls move closer, and at some point the center point between the balls will have a strength that is above our threshold. This effect will propagate outwards as the balls move closer, and we will use this later to make the balls appear as if they blob together in 3D.

3.2 Marching cubes

Marching cubes is a well-known algorithm for constructing a polygonal mesh of an isosurface form a given scalar field. It was first published in the 1987 SIGGRAPH by William E. Lorensen and Harvey E. Cline, and was highly adopted in medical visualization.

This algorithm takes a scalar field as an input, and output a list of triangles representing the isosurface. It iterates through the scalar field and calculate all corresponding value for each neighboring vertices in a voxel. Then depending on whether the value of neighbor vertices fall in or out of the isolevel, a list of triangle are generated according to the configuration. Since there are 8 vertices in a voxel, there are a total of 256 configurations of polygon placement as shown in figure 4. By combining all triangles generated within the scalar field, we can obtain the approximated polygonal mesh as a whole.

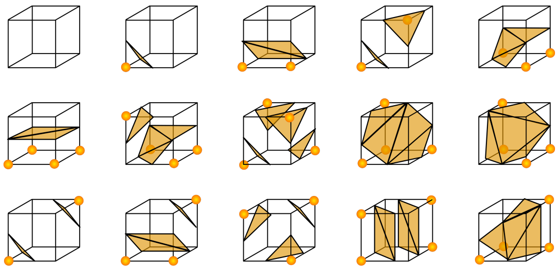

Figure 4: Marching cubes configurations

3.3 Smoothed Particle Hydrodynamic

Smoothed Particle Hydrodynamic (SPH) is a common computational method for simulating fluid flow on a machine. It was first published by Monaghan, Gingold and Lucy in 1977. In a SPH simulation, each fluid particle is interacting with its neighboring particle depending on its pressure, density, and velocity given a finite boundary around the particle. A particle's density is denoted as

$$p_i = \sum_j m_j W_{ij}$$

where W_{ij} can be any smoothing kernel. The pressure is then calculated with the ideal gas law in thermodynamics, written as $P = K(p - p_0)$. With the found density and pressure of a particle, we can find its acceleration by the following equation,

$$a_i = -\sum_j \frac{m_j}{m_i} \frac{P_i + P_j}{2p_i p_j} \nabla W_{ij} \hat{r}_{ij}$$

where ∇W_{ij} is another smoothing kernel and $\hat{r}_{ij}$ is the normalized vector of a surrounding particle to the current particle. Finally, the particle position is being updated according to the computed acceleration.

4 Implementation

We have used GLFW and GLSL. GLFW is a cross-platform utility library for creating window and OpenGL context, and we will use it for handling both window and input events. While GLFW handles all the high-level program logic, our project focuses on the low-level rendering. Mainly we will use GLFW to manage the OpenGL context. GLSL is the shading language for OpenGL which enables developers to control over the rendering pipeline. We will use GLSL to write our shader programs.

4.1 Marching cubes

```
private void Test(){
int a = 1;
}
```

4.2 Smoothed Particles Hydrodynamic

```
private void Test(){
int a = 1;
}
```

4.3 Misc

For the purpose of creating a nice demo, we created a fragment shader with basic Phong illumination, in order to give the animation more depth. We also added a physics model to the balls which consists of outer forces due to gravity and a repelling force between each pair of balls when they are close to each other, to prevent them from accumulating in one place. We also modeled the bounciness of the floor and walls of our simulation domain, in order to keep the balls confined and in motion for as long as possible.

We will not go into the details of how we implemented physics and the illumination model, as they are out of the focus of this report.

5 Result

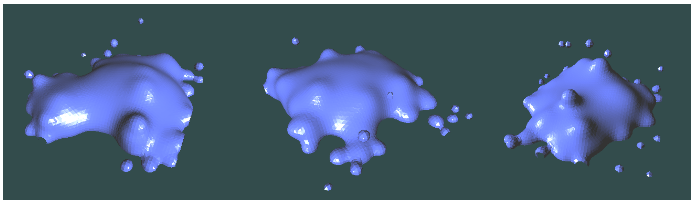

Figure 5: A concept of 2 metaballs in 2D. Every frame shows the two balls moving closer to each other.

An animated video can be found here: `https://youtu.be/jmScNqchXs0`

6 Conclusions

6.1 Rendering model

The voxel grid size is the performance bottleneck.

Comparison. Number of metaballs vs the maximum voxel grid size before the framerate drops below 30:

50	120
20	160
5	200

Disabling the metaball filter completely, and just running the voxel shader yields a maximum voxel grid size of 310 units before the framrate drops below 30.

The test was run on a NVIDIA Geforce GTX 1050 Ti.

Conclusion: The geometry shader itself can run with a voxel grid size in the order of 100-200 units at 60 fps, and up to 300 units at 30 fps. For our purpose, this can be used to create a nice looking dynamic fluid on a small simulation domain, or a very chunky fluid on a medium-large sized simulation domain. To render an impressive fluid in a medium sized voxel grid, it would be nice if we could optimize the particle model so that we could run in the order of 100 particles at a voxel grid size of 200. The following section will discuss some ideas for how this could be achieved in theory.

6.2 Particle model

It seems that the industry standard for fluid simulation in high quality games and video is the Smoothed Particle Hydrodynamics model with Ellipsoid Splatting rendering. Right now we are brute forcing the potential field over all particles (Possible conslusion: in line with SPH: spatial subdivision of metaball positions will allow for more balls, but not yield the neccessary performance increase that is needed to simulate particles in a high-resolution domain)

Yet another performance problem comes from the fact that the division step in our metaball potential function is computationally expensive. Ryan Geiss (whose tutorial on the metaball isosurface model we used) came with a suggestion for an approximate polynomial function which should in theory be faster on the GPU. His suggested function was:

$[g(r) = r^4 - r^2 + 0.25]$

This potential field function also has the neat property that it evaluates to 0 at $r = \frac{1}{\sqrt{2}}$, which means that you could do spatial optimization by only evaulating the points that fall within the radius of $r = \frac{1}{\sqrt{2}}$ of a singe metaball's center.

[http://www.geisswerks.com/ryan/BLOBS/blobs.html]

Our prediction is that this would yield a similar performance increase to implementing a naive spacial subdivision, but we are still not solving the problem that the geometry shader itself is expensive.