Simulating Fluids in Zero Gravity

Gabriel L. Somlo
Computer Science Department
Colorado State University
Fort Collins, CO, U.S.A.

Abstract *A lot of research has been conducted on achieving realistic fluid simulations: waves, flow, waterfalls, etc. This paper focuses on a less talked about aspect of fluid simulation: the behavior of liquids in zero gravity. Unlike in most other situations which are dominated by gravity, surface tension becomes the key phenomenon of the simulation. A spring-connected system of particles is used to model the surface tension of the fluid. The system is executed for a series of steps (at regular intervals of time), which then are rendered and become frames of an animation. Some details need further refinement, but overall the results are very encouraging.*

Keywords: fluid, gravity, particle, multithreading

1 Introduction

Most movies about adventures in space rely heavily on computer-generated special effects. Obviously, it is much more cost-effective to use a computer than to move equipment, actors, and crew into orbit in order to shoot a couple of takes. The absence of gravity is what causes the main difference in the behavior of objects and people in space. Realistic movement of people and hard objects can be achieved by filming underwater. Unfortunately, this method will not work with fluids. Weightless fluids are no longer forced by gravity to adopt the shape of a container, or to flow on a surface. Instead, surface tension becomes the main phenomenon that influences their behavior.

A lot of work has been done to simulate fluids in the presence of gravity. O'Brien and Hodgins [7], and Kass and Miller [6] have each obtained spectacular results in simulating flow, waves, and splashing of fluids in the presence of gravity, based mainly on solving the Navier-Stokes equations of fluid dynamics. Chen et.al. [4] proposed a *real-time* simulation of fluid flow, also based on a simplified version of the Navier-Stokes equations. However, little has been published about the behavior of liquids in the absence of gravity.

In this paper, I propose a method to realistically simulate surface tension in fluids. A grid of particles will delimit the fluid surface. Neighboring particles are connected by springs, to model surface tension. Unconnected particles reject each other, in order to preserve the object's volume.

The method is described in Section 2. Several considerations about the implementation of the algorithm are presented in Section 3. Section 4 describes the results I have obtained. Finally, Section 5 contains some concluding comments and several ideas for future improvements.

2 Force-Directed Placement

The method I propose is based on a graph layout algorithm developed by Fruchterman and Reingold [5]. In essence, they represent a graph as a system of particles (nodes) connected by springs (edges), and let it go through a relaxation process. Unconnected particles reject each other, whereas particles that share an edge attract each other. The desired result

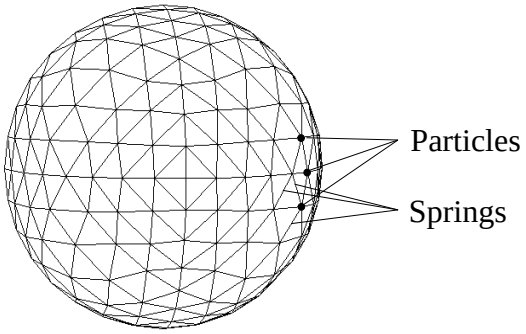

Figure 1: Representation of the fluid surface

is a graph that is readable, with few intersecting edges. An animation of the intermediate steps of the relaxation was quite spectacular, suggesting the simulation of surface tension in fluids presented in this paper.

The surface of a fluid in zero gravity can be represented by a polygon mesh, with *particles* as vertices, and *springs* as edges. Unconnected particles reject each other, in order to preserve the volume of the fluid. Surface tension is modeled by the springs connecting neighboring particles on the surface. A portion of such a surface is depicted in Figure 1. The force-directed placement algorithm is presented in Figure 2.

At each relaxation step, all particle positions are adjusted according to the cumulative effects of inter-particle interaction forces - rejection from all unconnected particles, and attraction towards all connected ones. These adjustments take place according to the formula

$$\Delta\overline{P}_i = \sum_{i=1}^{N} \overline{u}_{ji} \cdot \frac{k_g^2}{d_{ji}} - \sum_{k=1}^{M_i} \overline{u}_{ki} \cdot \frac{d_{ki}^2}{k_e}$$

where

$$\overline{u}_{ji} = \frac{\overline{P}_i - \overline{P}_j}{|\overline{P}_i - \overline{P}_j|}, \text{ and } d_{ji} = |\overline{P}_i - \overline{P}_j|$$

$\overline{P}_i$ denotes the position of the i^{th} vertex; $\overline{u}_{ji}$ is the unit vector pointing from vertex j to vertex i; d_{ji} is the scalar distance between vertices j and i; finally, k_g and k_e are the gravitational and elastic constants, respectively. All N particles contribute to the rejection force, but only the subset of M_i connected particles contributes to the attraction force.

Each step of the algorithm has a complexity of $\mathcal{O}(N^2)$, because the rejection forces are exerted between any two particles in the system. The connectivity of the polygon mesh is limited to a constant $M_{\max}$, and therefore the part of the algorithm that computes the attraction between connected particles will have a complexity of only $\mathcal{O}(N)$.

After every recomputation of the vertex positions, the polygon mesh is colored using one of the known shading methods: Gouraud, Phong, or Ray-Tracing. Thus, we obtain one frame for each step of the force-directed placement algorithm. These frames, when played in a sequence, accurately depict the behavior of a fluid object in the absence of gravity.

3 Implementation Issues

In this section I present several of the issues that had to be dealt with while implementing the simulation: parallelism and simulation speed, and properties of the polygon mesh.

3.1 Parallelism and Simulation Speed

For large masses of fluid (or for a very accurate model of the fluid surface), a large number of particles would be required. Due to the $\mathcal{O}(N^2)$ complexity of the simulation algorithm, this might lead to unacceptably long simulation times. I have considered two ways to speed up the simulation: a multithreaded implementation [2], and the use of a better algorithm to compute the rejection forces [3]. Currently, I am only using the first method - a multithreaded implementation of the simulation.

3.1.1 Multithreading

Multithreading takes advantage of the presence of multiple processors in a workstation, allowing for parallelism without the overhead of creating multiple processes. Multithreaded programming makes use of all the well-known syn-

```
volume = W · H · D;   // W, H, and D are the width, height, and depth of the volume
G = (V, E);           // V and E are the vertices and edges of the graph

function f_a(x) { return x^2/k_e; };   // elastic attraction force

function f_r(x) { return k_g^2/x; };   // (negative) gravitational rejection force


for i = 0 to number_of_frames do
    // calculate repulsive forces
    for v ∈ V do
        // each vertex has position (.pos), and displacement (.disp)
        v.disp = 0;
        for u ∈ V, u ≠ v do
            // Δ is the difference vector between nodes
            Δ = v.pos − u.pos;
            v.disp+ = (Δ/|Δ|) · f_r(|Δ|);
        end for u
    end for v

    // calculate attractive forces
    for e ∈ E do
        // each edge is a pair of vertices .v and .u
        Δ = e.v.pos − e.u.pos;
        e.v.disp− = (Δ/|Δ|) · f_a(|Δ|);
        e.u.disp+ = (Δ/|Δ|) · f_a(|Δ|);
    end for e

    // limit maximum displacement; also, prevent displacement outside the volume
    for v ∈ V do
        v.pos+ = (v.disp/|v.disp|) · min(v.disp, maxdisp);
        v.pos.x = min(W/2, max(−W/2, v.pos.x));
        v.pos.y = min(H/2, max(−H/2, v.pos.y));
        v.pos.z = min(D/2, max(−D/2, v.pos.z));
    end for v
end for i
```

Figure 2: Pseudocode for force-directed placement algorithm

chronization mechanisms available in a shared memory multiprocessor model: mutex locks, conditional locking, semaphores, etc.

The force-directed placement algorithm is very much suitable for parallelism in general, and multithreading in particular. Practically, during the computation of the rejection forces, each particle can be assigned a thread. Similarly, during the computation of the attraction forces, each edge can also be assigned a thread. For practical purposes however, the number of threads should equal the number of processors in the workstation (P). Therefore, P worker threads are started at the beginning of the pro-

gram, each of them processing as many particles or edges as necessary, until all the work has been finished.

As far as inter-thread synchronization is concerned, no thread will ever need to read from a location being written to by another thread. Threads processing particles will never attempt to write to the same location. The only problem arises when multiple threads processing edges attempt to update the position of a shared vertex. One possibility to solve this problem would be to assign a mutex lock to each node, forcing edge-processing threads to acquire locks for both of their ends before updating their positions. This approach is similar to the well-known "Dining Philosophers Problem". Another solution would be to allow only one edge-processing thread to update its ends at any given time, using only one mutex lock. This is based on the realistic assumption that these threads will spend a much longer time computing the position updates than writing them to memory, and therefore none of them will be significantly delayed in their activity. Since the main focus of this paper was a realistic simulation of fluid behavior, I postponed a test for the relative efficiency of the two above methods for future work. Because the first method seems more deadlock prone, I preferred the second one, which seems safer and simpler.

3.1.2 The N-Body Problem

Another way of dealing with a large number of particles in the simulation is to use a faster algorithm to compute the rejection forces. Computing gravitational interaction between N particles is known as "the N-body problem". Each particle interacts with all other particles, and therefore, the most straightforward way of computing all the interactions would have a complexity of $\mathcal{O}(N^2)$. However, Barnes and Hut [3] developed an improved algorithm of complexity $\mathcal{O}(N \log N)$. Their approach is based on a hierarchical subdivision of space into cubic cells, each of which is recursively divided into eight subcells whenever more than one particle is found to occupy the same cell.

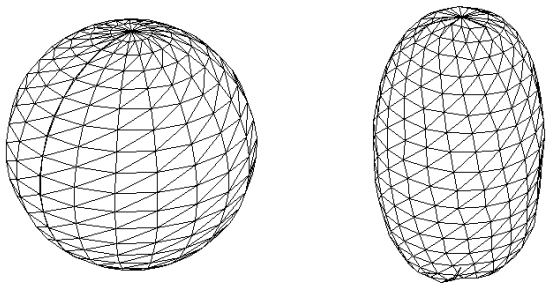

Figure 3: Relaxation of a spherical polygon mesh

Interactions between distant *clusters* of particles can thus be computed with high accuracy, significantly reducing the cost of computation for a large number of particles.

I have not implemented this algorithm in the current version of my simulator, since currently the low number of particles did not justify the effort. However, should the need arise, using this algorithm would reduce the *overall* complexity of the simulation to $\mathcal{O}(N \log N)$.

3.2 Properties of The Polygon Mesh

The topology of the polygon mesh modeling the fluid surface has a great influence on the equilibrium shape of the fluid mass. In nature, surface tension would cause a mass of fluid to adopt a spherical shape in the absence of gravity. Therefore, I started out with a parametrically generated sphere (see Figure 3, left). However, due to the fashion I implemented the force-directed placement algorithm, all of the springs tend to end up having the same, user-specified target length. This caused the parametric sphere to end up looking like a football (see Figure 3, right) - with the longer edges from around the "equator" shortened, and the shorter edges from around the "poles" elongated.

This problem can be solved either by specifying a distinct target length for each edge, or by using an equilateral polyhedron to approximate the spherical rest shape of the fluid mass.

The first solution is easy to implement, but less efficient and elegant. Moreover, the mass of fluid will display non-uniform behavior across its surface, due to the non-uniform concentration of the sampling points (particles). The second solution is simple, efficient and elegant. Its difficulty consists in building a polyhedron with facets that are equilateral triangles, to approximate a sphere as close as possible.

I decided to adopt the second solution. For the results presented in this paper, I used a polyhedron like the one depicted in Figure 1, which turned out to be a good enough approximation of the "perfect" polyhedron I was looking for.

4 Results

I have used a mesh consisting of 512 triangles and 258 vertices to simulate a fluid object. The structure of this mesh has been depicted in Figure 1. The force-directed placement simulation was executed for 100 steps, each step resulting in a still frame. The size of each of these frames was 300×300 pixels.

An approximate comparison on a dual-processor UltraSparc workstation which already displayed a 50% CPU usage, showed that the multithreaded simulation of the 100 steps took 15 seconds, whereas the single-threaded simulation took 20 seconds. Considering that a large fraction of the time is spent writing output files (an operation that could be eliminated in the future), the actual speedup should be much better.

Some of the intermediate frames are presented in Figure 4. The MPEG animation of the simulation can be found at *http://www.cs.colostate.edu/~somlo*

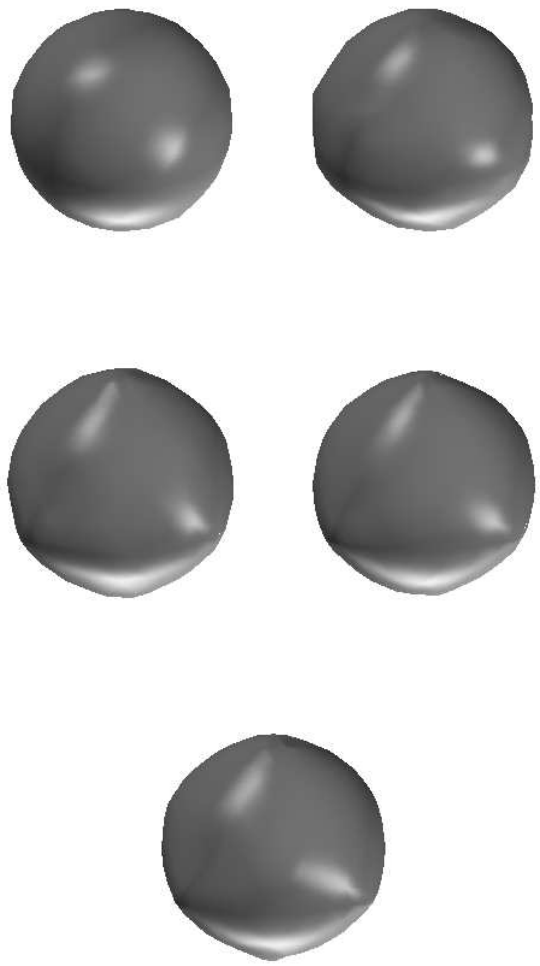

Figure 4: Intermediate frames from the fluid simulation

5 Conclusions and Future Improvements

In this paper I have presented a method for realistic simulation of fluids in zero gravity. The method is based on a system of particles connected by springs (modeling surface tension), and the experiments I have run so far are very encouraging.

The simulation could not be executed in real time, because of the delays introduced by the shader. For a more realistic simulation, where the transparency and refractive properties of the fluid would be taken into account, the expected delays introduced by the shader would be even greater.

In the future, I plan to use a raytracing-based shader, in order to add refraction and transparency to the images. Also, I will look into ways to speed up the shading process,

by using parallelism (multithreading) and algorithm optimizations. The capabilities of the simulator could also be extended to allow for merging and splitting of multiple fluid droplets, and for deformation due to contact with a hard surface. Last, but not least, a more accurate measure of the speedup of the multithreaded implementation should be obtained.

References

[1] ***. *Berkeley MPEG-1 Video Encoder: User's Guide.* Plateau Research Group, Computer Science Division, U.C. Berkeley, 1995.

[2] ***. *Multithreaded Programming Guide.* Sun Microsystems Inc., 1994.

[3] J. Barnes, P. Hut. A Hierarchical $\mathcal{O}(N \log N)$ Force-Calculation Algorithm. *Nature*, 326:446-449, 1986.

[4] J.X. Chen, N. da Vitoria Lobo, C.E. Hughes, J.M. Moshell. Real-Time Fluid Simulation in a Dynamic Virtual Environment. *IEEE Computer Graphics and Applications*, 17(3):52-61, 1997.

[5] T.M.J. Fruchterman, E.M. Reingold. Graph Drawing by Force-Directed Placement. *Software - Practice and Experience*, 21(11):1129-1164, 1991.

[6] M. Kass, G. Miller. Rapid, Stable Fluid Dynamics for Computer Graphics. *Computer Graphics*, 24(4):49-55, 1990.

[7] J.F. O'Brien, J.K. Hodgins. Dynamic Simulation of Splashing Fluids. *Proceedings of Computer Animation '95*, Geneva, Switzerland, 1995.