An Object-Oriented Particle System for Simulation and Visualization

Jun Zhang
Edward Angel
Department of Computer Science and
Albuquerque High Performance Computing Center
University of New Mexico
Albuquerque, NM 87131

Paul Alsing
David Munich
Albuquerque High Performance Computing Center
University of New Mexico
Albuquerque, NM 87131

Particle systems have been used successfully in applications ranging from simulations of physical phenomena, to creation of special effects in the movie industry, to scientific visualization. One problem encountered in practice is that application programmers usually derive a new code from scratch for each application even though particle systems have a basic object-oriented flavor. In this paper, we demonstrate a simple object-oriented particle system that has been implemented in C++ and applied to a variety of applications including graphical special effects, Lennard-Jones fluid simulation, isosurface extraction, and generation of streamlines from flow simulations. In each case, the code was developed from the basic particle system in a very short time. We shall also show comparisons with non-object-oriented codes that demonstrate that the object-orientation has only a small performance penalty. We shall also demonstrate parallelization of our simulations.

Keywords: particle systems, object-oriented, parallel programming, simulation, visualization

Introduction

Particle systems have a long history. Within the simulation community, particle systems have been used to simulate a variety of physical phenomena, providing an alternate to traditional methods. Within the graphics community, particle systems were introduced by Reeves[9] to model explosions. Later particle systems were used to model natural phenomena such as fire and water, flocking of birds, and the behavior of crowds[3, 6, 10, 11, 12]. In scientific visualization, particle systems have been used to find points on isosurfaces defined by implicit functions.

What is common in all these is the use of a system of particles whose behavior is controlled by a set of rules. For physical simulations, the rules are the physical laws. For

isosurfaces, the rules are such that they keep particles on a surface, even if that surface changes in time. In computer graphics, the rules are behavorial. Note that in these examples, the behavior of the particles is independent of how they are viewed or rendered.

It has been argued [1] that particle systems are inherently object-oriented. In object-oriented terms, particles are objects with a set of attributes and methods. Nevertheless, in applications of particle systems, each application usually has involved completely independent code development. Furthermore, particle systems have many features that make them well suited for solution on high-performance computers. For example, we can usually allocate groups of particles to processors in a straight-forward manner. We usually distribute the force (behavior) calculation among processors.

We decided to build a simple base particle system using the object-oriented features of C++ and then use it to develop code for some of the standard applications that have been used successfully with particle systems. We had two basics motivations. We wanted to demonstrate that starting with the basic system we could develop the code for specific applications very quickly by subclassing functions in our basic systems. Second, we wanted to measure the performance penalties of using an object-oriented system.

This paper is organized as follows. In the next section, we shall give a brief introduction to particle systems consisting of Newtonian particles. We shall then describe the base classes of our software system. Then we shall describe four applications that we used to test the system. The first applications are standard graphics applications: a simulated waterfall and a spring-mass system. The second application is a Lennard-Jones fluid simulation. We compare the processing time with implementations in C and C++ using a direct conversion from the original non object-oriented code. The third application is an implementation of Heckbert and Witkins's isosurface sampling algorithm. This system has much more complex dynamics and is a good test of how easy it is to use the object-oriented system for development of new systems. Finally, we use our system for generation of streamlines for a global ocean current visualization. In this application, we replace differential equation solvers on a single processor by parallel solvers.

Particle Systems

A particle system is a collection of discrete entities called particles. Each particle is described by its state and a collection of attributes that determine its appearance and other factors such as its lifetime. In this work, we will consider only Newtonian particles, each of whose state is entirely determined by its position and velocity. Thus, in three dimensions, a particle has six degrees of freedom and a system of n particles has 6n degrees of freedom. In terms of the differential equations that determine the evolution of the particle system, particle i is has a position $\mathbf{p}_i(t)$ and a velocity $\mathbf{v}_i(t)$ and satisfies the equations

$$\frac{\mathbf{f}_i(t)}{m_i} = \frac{d\mathbf{v}_i(t)}{dt}; \qquad \mathbf{v}_i(t) = \frac{d\mathbf{x}_i(t)}{dt}$$

where m_i is the mass of the particle *i* and $\mathbf{f}_i$ is the total force on the particle *i*. The entire complexity of the particle system resides in the force term which may include terms from forces due to interactions with all other particles in the system.

In an ideal Newtonian system each particle is a point mass and thus could be rendered for visualization purposes as a point. However, much of the flexibility of particle systems comes from our ability to render particles in a manner decoupled from their state. Thus, in an animation, the state determines the location of each particle, but we can render each particle as an animated character centered at that location. Thus, the attributes of the particle include this kind of information.

We can thus implement a particle system by looping through the steps

1. Accumulate the forces on each particle
2. Step forward one time step using a standard differential equation solver
3. If desired, render the particle system using the particles' attributes.
4. Account for any creation (birth) of new particles or deletion (death) of old particles

The force accumulation step is the most time consuming part of the process and what characterizes a particular application. Generally, there are three types of forces (1) unary forces in which the force on each particle does not depend on other particles (2) k-ary forces in which the force on a particle depends on a small set of up to k other particles, and (3) n-ary forces where each the force on each particle can depend on all other particles. For a system of n particles, the first two types require an O(n) force calculation while the third is $O(n^2)$, unless we make some type of approximation.

Once we have done the force accumulation, we compute the state of all the particles at the next time step. This is usually done by using a standard ordinary differential equation solver. Depending on the order of the solver, multiple force accumulations may be required for each time step.

The first two steps are routine in any simulation using models whose evolution can be described by differential equations. The third step is key to the graphical and visualization applications of particle systems. Here we use the state of the particle system to determine an image. The basic graphical objects in the image need not have any relationship to the particles other than their locations. For example, in animation of crowd scenes the particle system governs the locations of the characters, keeping them moving and avoiding collisions through repulsion forces. However, once the state is known at a given time, we can place an entire character at the location in the state.

The challenge is to build a flexible software system that can support a multitude of applications by allowing for different force accumulators, different numerical methods, including the ability to parallelize the code, different methods of rendering, and different

birth and death rules. Heckbert and Witkin [15] suggested an object-oriented particle system but did not implement it within an OOP framework, such as C++. We have three basic objects in our system: particles, forces, and particle systems. We start with the base class for a particle:

```
class Particle {
public:
  Particle();

  float    _mas;   //mass
  float    _age;   //age
  vector_3 _pos;   //position vector
  vector_3 _vel;   //velocity vector
  vector_3 _frc;   //force accumulator
  vector_3 _col;   //RGB color
  float    _rad;   //repulsive radius
};
```

The members mass, position, force, and velocity have obvious meaning. The age allows us to eliminate older particles. The repulsive radius [15] allows us to speed up the calculation by considering only forces from neighboring particles.

```
class Force{
public:
  Force(ParticleSystem* psys);

  virtual void applyForce() = 0;

  Force* _next;

protected:
  ParticleSystem* _my_psys;
};
```

The force class contains a pointer to a particle system on which it acts and a pure virtual function that calculates the forces.

```
class ParticleSystem {
public:
  ParticleSystem();
  ParticleSystem(int max);
  virtual ~ParticleSystem();

  virtual void simulate() = 0;
  virtual void render() = 0;
```

```
  int  numOfParticles();
  void addParticle(Particle p);
  void killParticle(int pn);
  void addForce(Force* pf);
  void computeForce();
  void EulerIntegrate(float deltatime);
  void print();

protected:
  Particle* _particles;  //particle array
  int       _num;
  int       _max;
  Force*    _forces;     //force linked list
```

The constructor, `ParticleSystem` allocates memory for an array of `_max` `Particle` structures (pointed to by `Particle* _particles`). The `numOfParticles` function returns the number of particles in the system. Functions `addParticle` and `killParticle` can be used to add or remove particles. The `addForce` function adds forces into the system. The `computeForce` function sets the forces of all particles to zero and then traverses the force linked list and applies the forces. `EulerIntegrate` uses the Euler method to calculate and update position and velocity of all particles in the system (and can be overridden by another integrator). The print function can be used to output attributes of all particles.

Function simulate does the simulation and render draws the particle system to the screen. As these methods are pure virtual methods, they must be implemented by derived classes. The class hierarchy of our object-oriented particle system is shown in Figure 1.

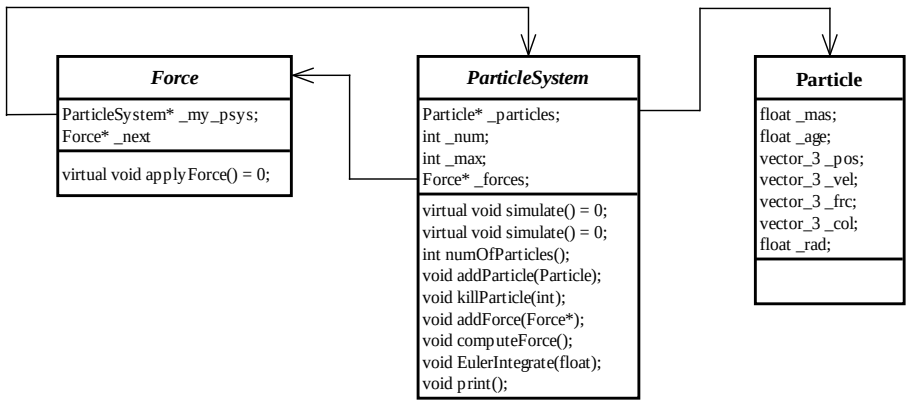

Figure 1. Particle System Class Diagram

Two Simple Examples

Our base implementation contains an Euler integrator and does not make any attempt to increase efficiency by keeping track of the neighbors of each particle. However, for simple examples we can create particle systems with very small user programs.

Figure 2 shows one time step of waterfall simulation. The only force here is gravity so there is no interaction among the particles. Each particle is rendered as a small dot. Figure 3 shows one frame of a spring mass system of two particles. In the rendering, a line is drawn connecting the particles.

Figure 2. Particle Waterfall

Figure3. Spring-Mass System

Although these examples are almost trivial, note that the dynamics and rendering are completely different. Each requires only about 30 lines of user code with our system.

Lennard-Jones Particle System

Our next example is the Lennard-Jones fluid simulation [5, 14]. Here the force between atoms is given by

$$\mathbf{f}_{ij} = \frac{24\varepsilon}{r_{ij}^2}\left[\left(\frac{\sigma}{r_{ij}}\right)^6 - \left(\frac{\sigma}{r_{ij}}\right)^{12}\right]\mathbf{r}_{ij}$$

where $\mathbf{f}_{ij}$ is the force on particle i due to particle j separated by distance $\mathbf{r}_{ij}$, and σ and ϵ are constants. The interparticle force consists of both an long range attractive and a short range repulsive term, both of which decrease rapidly with the distance between particles. The usual way of working with this system is to solve for particles within a three-dimensional box and assume periodicity to obtain terms when needed from neighboring boxes.

We solved this problem initially in three ways. First, we used Plimton's F77 code [8]. We then converted this code to C code so that we could check for performance differences due to the language change. We then reimplemented the code with our particle system. We had to add class members for pressure and energy and implement the forces, all of which were relatively straightforward tasks. We also changed the differential equation solver to use the velocity-Verlet method[13]. The time results in Table 1 show a small performance penalty for the object-oriented particle system.

Data	Fortran77 code	C code	C++ code
Total time (secs)	3002.58	3570.61	3837.36
Force time (secs)	2940.09	3486.62	3722.10
Temperature	0.58583	0.58656	0.58674
Pressure	-0.49264	-0.49933	-0.49837
Potential energy	-5.08588	-5.08692	-5.08710
Total energy	-4.20714	-4.20708	-4.20700

Table 1: Timing comparison and simulation results for 3375 atoms and 5000 time steps.

For large numbers of particles, the $O(n^2)$ force calculation dominates. Although we do not perform the force calculations for particles separated by more than a cutoff distance, the computation is still $O(n^2)$ if we have to compute the radius for all pairs of particles. As an alternative we used a link-cell method [5]. Here particles are placed in cells of a physical dimension such that a particle can only interact with particles its own or neighboring cells. At the cost of maintaining cell lists, the $O(n^2)$ force calculation is reduced to O(n m) where $m = n\, n_{cells}\, \rho$. Here n_{cells} is the number of neighboring cells

searched (27 in 3D, reduced to 13 if Newton's third law is incoporated into the force computation) and ρ is the ratio of a single cell volume to the total computational volume, with n_{cells} ρ << 1. This change required only about 100 lines of code being added to the basic particle system code. Further reduction in the time to calculate interparticle forces could be obtained if one additionally incorporated the use of nearest-neighbor lists [5] for each particles. Though such structures could easily be added to objected-oriented code, nearest-neighbor list were not utilized in this present work. Table 2 summarizes the results.

Number of atoms	512	1000		3375	
Number of cells	3	3	4	5	6
Total time (secs)	110.56	396.00	184.07	1059.43	694.24
Force time (secs)	102.92	376.62	164.32	926.39	561.94
Temperature	0.58612	0.58574	0.58673	0.58612	0.58651
Pressure	-0.50665	-0.50305	-0.49832	-0.49368	-0.49358
Potential energy	-5.08634	-5.08595	-5.08730	-5.08626	-5.08690
Total energy	-4.20716	-4.20733	-4.20721	-4.20708	-4.20713

Table 2: Timing and simulation results for Lennard-Jones system with link-cell structure for 5000 time steps.

Sampling Implicit Surfaces

Our next example is an implementation of Witkin and Heckbert's particle system [15] for sampling implicit surfaces of the form

$$F(x,y,z,t) = 0,$$

where the function F is known analytically. Note that the surface may be changing in time. What is desired is a system that will place particles on the surface and keep them there as the surface changes in time. In addition, the particles should spread them themselves uniformly over the surface so that their locations can be used to determine a triangular mesh that approximates the surface.

In this system there are two types of forces. One keeps a particle that is known to be on the surface initially on the surface as the surface moves. The second force is a repulsive

force among particles that distributes the particles over the surface. The first force can be expressed in terms of F and its partial derivatives. The second is based on Gaussian repulsive force whose width is based on a radius of repulsion. In Witkin and Hekbert's scheme each particle can have its own radius of repulsion which aids in convergence and in the birth-death process which creates new particles in regions where they are underpopulated and removes particles in regions of high particle density. With these adjustments, the particle system can start with a single particle on the isosurface and converge with a user determined density of particles uniformly distributed over the surface. Figure 3 shows a sequence of images of particles populating a sphere. The size of the particles in the rendering is proportional to their radii of convergence. Figure 4 shows a blobby sphere determined by 500 particles.

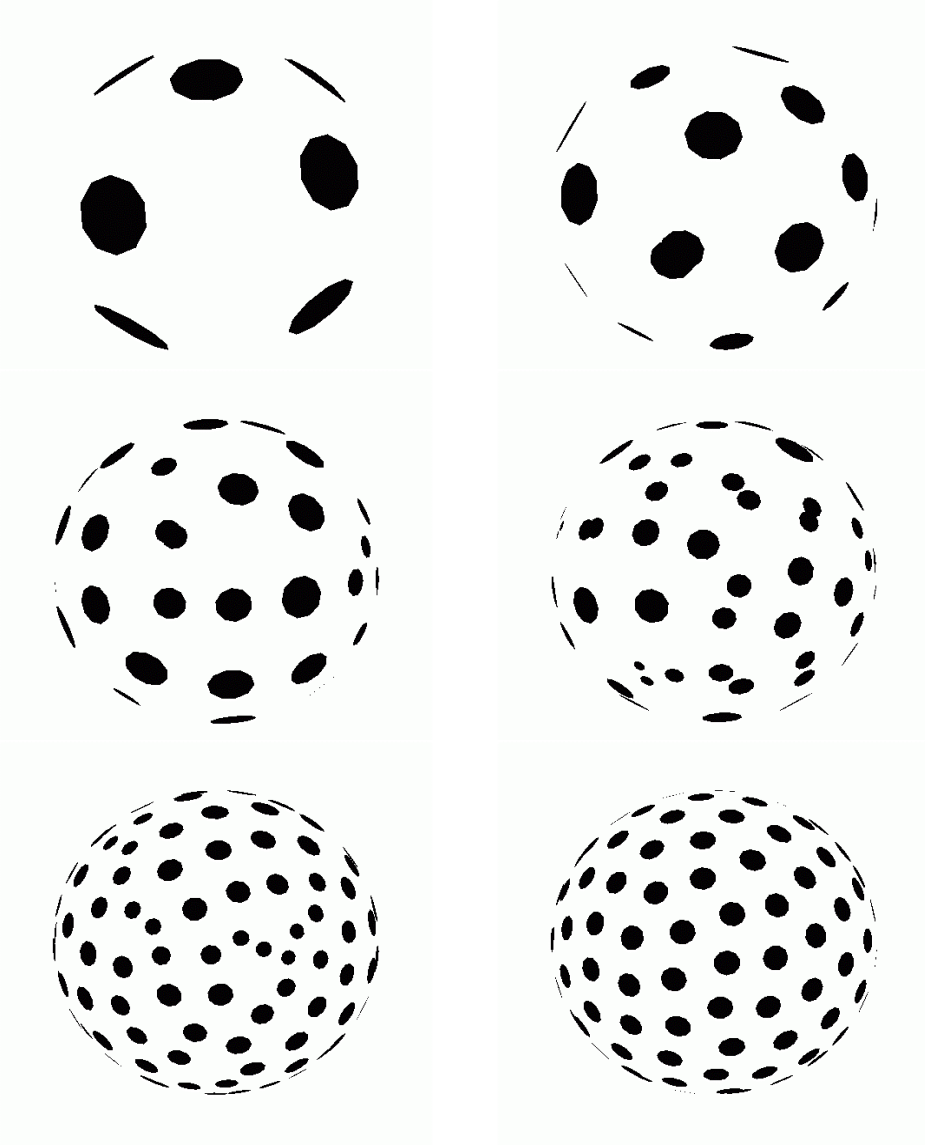

Figure 3. Particles populating surface of sphere

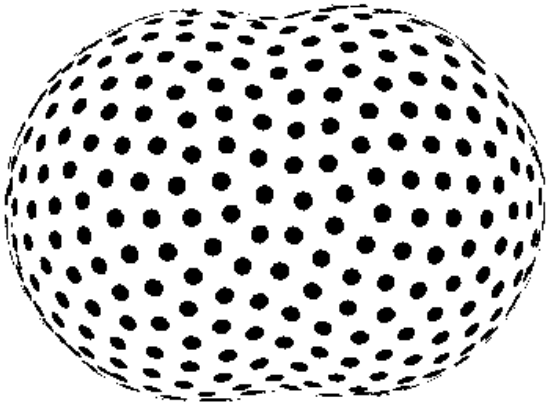

Figure 4. Blobby sphere rendered with 500 particles.

Although this example required extensive changes to the basic code, all the changes were in subclasses. More impressive was the fact that the entire system was created in a few days.

Flow Visualization

The final example is flow visualization using streamlines. Flow visualization uses weightless particles whose evolution is determined by the values of the velocity given in the data set. Thus, although there are no forces in the sense of our previous examples, we can use the object-oriented particles system by a simple modification of the differential equation solver. Each streamline is the trace of a particle. Particles are created where we want to start a streamline and terminated (killed) when the velocity goes below a threshold or some other criteria is met. For example, Figure 5 shows streamlines from a global ocean current simulation. In this visualization, we are interested in finding eddies in the flow. We do so by counting sign reversals in the cross product of flow vectors as we move along the streamlines.

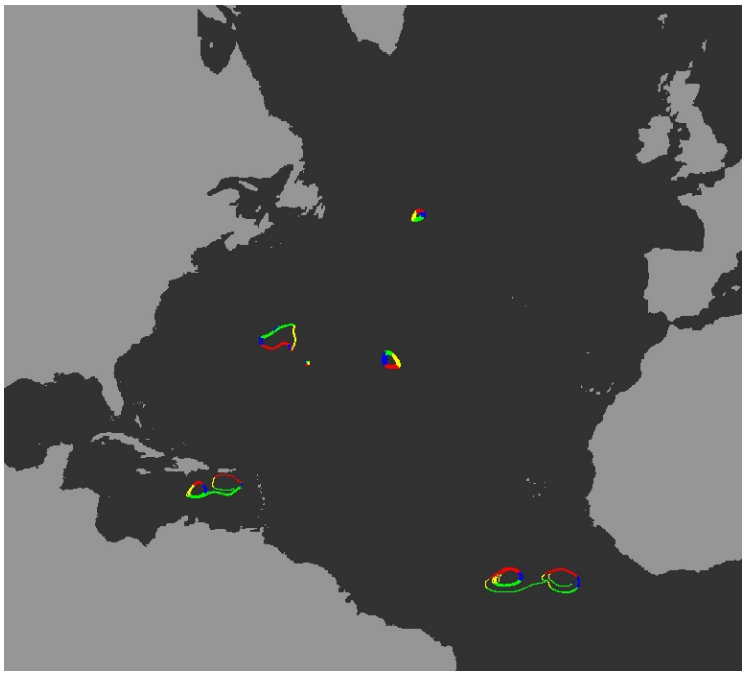

Figure 5. Eddies in ocean current simulation.

Parallelization

We have implemented a preliminary parallel version of the Lennard-Jones system without link-cell structure. We parallelized the force computation using the atom-decomposition algorithm [7]. In this algorithm, each of the P processors is assigned a group of N/P atoms at the beginning of the simulation. Atoms in a group need not have any spatial relationship to each other. A processor computes forces on only its N/P atoms and will update their positions and velocities. This algorithm requires all-to-all communication so that each processor knows about the positions of all the atoms in the system. We implemented this algorithm using MPI library. Table 3 shows the timing results on up to 10 processors.

Number of Processors	2	4	5	10
Total Time (s)	345.06	180.34	139.30	80.87
Force Time(s)	331.52	166.63	126.45	67.14
Communication Time (s)	2.36	3.6	3.14	4.55

Table 3: Timing results for the parallel Lennard-Jones System for 1000 particles.

Atom-decoposition or replicated data is the simplest, yet crudest form of molecular dynamics parallelization. As a measure of scalability, a parallel algorithm is said to be *isoefficiently scalable* [7] if one can increase the number of processors P and retain its parallel efficiency by increasing the size of the problem (i.e. larger problems can utilize larger number of processors). The appropriate metric for isoefficienctly scalability (*IP*) is the ratio of communication costs to computations costs, which one desires to keep near unity. The atom-decomposition algortithm has an *IP* metric of P. In this sense, the atom-decomposition is not a scalable algorithm since its *IP*-value increases with the number of processors. Other competing algortihms [7] such as force-decomposition and spatial decomposition have *IP* metrics of $\sqrt{P}$ and 1 respectively, but involve a substantial overhead in coding. We chose the atom-decomposition algorithm for this work for its ease of coding and independence of geometry. The other algorithms could be straight-forwardly incoporated into our objected-orient code, and will be investigated in the future.

Conclusions

We set out to test the proposition that an object-oriented particle system can be used for both simulation and visualization purposes. The test of success of such a system rests on both the effort required for code development and the performance of the resulting code. We believe we have been successful on both counts. We developed code for five very different applications, all in a very short time, once we had the basic system working. Timing tests show a very small performance penalty compared with code developed specifically for one application. Most of the performance difference appears to be due to the conversion from F77 to C/C++ rather than to the imposition of the object-oriented system. We were also successful in parallelizing the system.

There is still much work to be done. First, we want to run our system with some much larger particle systems, systems with at least $O(10^4)$ particles. One possibility is to use our previous use of particle systems for isosurface extraction from scalar fields [2]. We would also like to run applications on systems with hundreds or thousands of processors. This would most likely entail using either a parallel spatial decomposition or force-decomposition algorithm. The former takes advantage of the decreasing surface to volume ratio of the spatial domains a processor is responsible for relative to the number of particles that need to be passed amongst neighboring processors, and is valid in the regime of 10^6-10^8 particles. The latter maintains the geometry indepence of the atom-decomposition method, but entails a much more efficient algorithm for computing interparticle forces without the use of an all-to-all communication, and is more efficient than spatial decomposition in the range of 10^3-10^5 particles. We would also like to implement other strategies to reduce the complexity of force calculations, epsecially serial and/or parallel fast-multipole methods.

Acknowledgements

The work was supported in part by the Albuquerque High Performance Computing Center and by Sandia National Laboratories. The global climate data were provided by the Advanced Computing Laboratory, Los Alamos National Laboratory.

References

[1] Baraff, D. and Witkin A. (1997), “Physically Based Modeling: Principles and Practice”, *SIGGRAHP’97 Course Notes*.

[2] Crossno, P. J. (1997), “Isosurface Extraction Using Particle Systems”, IEEE Visualization, 1997.

[3] Fournier, A. and Reeves W. T. (1986), “A Simple Model of Ocean Waves” *Computer Graphics (Proceedings of SIGGRAPH ’86)* **20**(4): 75-84.

[4] Heckbert, P. S. (1997), “Note from Course 14: New Frontiers in Modeling and Texturing” *SIGGRAPH ’97*, Los Angeles, California.

[5] Hockney, R. W. and Eastwood, J. W. (1988), “Computer Simulation using Particles” McGraw-Hill Inc., New York; Allen, M.P. and Tildesly, D.J. (1987), “Computer Simulations of Liquids,” Oxford Science Publications, New York.

[6] Peachey, D. R. (1986), “Modeling Waves and Surf” *Computer Graphics (Proceedings of SIGGRAPH ’86)* **20**(4): 65-74.

[7] Plimpton, S. (1995), “Fast Parallel Algorithms for Short-range Molecular Dynamics” *Journal of Computational Physics* **117**: 1-19.

[8] Plimpton, S (2000), “Home Page for Steve Plimpton,” *http://www.cs.sandai.gov/~sjplimp/main.html.*

[9] Reeves, W. T. (1983), "Particle Systems- A Technique for Modeling a Class of Fuzzy Objects" *ACM Transactions on Graphics* **2**(2): 91-108.

[10] Reeves, W. T. and Blau, R. (1985), "Approximate and Probabilistic Algorithms for Shading and Rendering Structured Particle Systems" *Computer Graphics (Proceedings of SIGGRAPH ’85)* **19**(3): 313-322.

[11] Reynolds, C. W. (1987), “Flocks, Herds, and Schools: A Distributed Behavioral Model” *Computer Graphics (Proceedings of SIGGRAPH ’87)* **21**: 25-34.

[12] Sims, K. (1990), “Particle Animation and Rendering Using Data Parallel Computation” *Computer Graphics (Proceedings of SIGGRAPH ’90)* **24**(4): 405-413.

[13] Swope, W.C., Anderson, H.C., Berens P.H., Wilson, K.R. (1982), “A Computer-Simulation Method for the Calculation of Equilibrium-Constants for the Formation of Physical Clusters of Molecules: Application to Small Water Clusters”, *J. Chem. Phys.* **76**:637-649

[14] Verlet, L. (1967), “Computer experiments on classical fluids: I. Thermodynamical properties of Lennard-Jones molecules” *Phys. Rev*. **159**: 98-103

[15] Witkin, A. P. and Heckbert, P. S. (1994), “Using Particles to Sample and Control Implicit Surfaces” *Computer Graphics Proceedings*, Annual Conference Series, 1994: 269-277.