Fast Surface Particle Repulsion

Paul S. Heckbert

30 April 1997

CMU-CS-97-130

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

To appear: New Frontiers in Modeling and Texturing course,
SIGGRAPH '97, Los Angeles, Aug. 1997

ph@cs.cmu.edu,   http://www.cs.cmu.edu/~ph

Abstract

A fast algorithm for simulating particle repulsion on surfaces is described. Particles are distributed across a surface and made to repel. Calculating pairwise repulsion between n particles costs $O(n^2)$ in general, but if the energy potential drops off very rapidly, repulsion can be simulated in $O(n)$ time. This is accomplished using a simple grid data structure with spherical range queries. Particles change in number and position, necessitating a dynamic data structure. The method is used to accelerate repulsion in an algorithm for uniformly sampling implicit surfaces during interactive surface design. Although this method is straightforward, it is quite effective: speedups of 160 times were realized for simulations with 10,000 particles. We also give references to related work outside the field of computer graphics.

Keywords: physically based modeling, spatial data structure, grid, spherical range search, bucketing.

1 Introduction

To display surfaces rapidly during interactive surface design, particles can be distributed across the surface and displayed using tangent disks. In late 1993, Andy Witkin and I developed such an algorithm, and worked out a method to quickly achieve an approximately uniform sampling of particles, and to maintain it during interactive shape change [13]. Our method employed particle repulsion, birth, and death to achieve maximum speed and interactivity. The algorithm described in that paper employed a brute force pairwise repulsion method requiring $O(n^2)$ time for n particles. Because of this high repulsion cost, the largest simulations we had run at that time used 500 particles. The present paper summarizes work done in July 1994, and shown in a videotape at SIGGRAPH '94, that allows these repulsion calculations to be done in $O(n)$ time. This fast repulsion algorithm allows us to now simulate up to 10,000 particles with one second iteration-and-redisplay times on a 100 MHz graphics workstation.

1.1 Related Work

The problem of distributing particles uniformly on surfaces has been studied in a more simplified form in recreational mathematics and combinatorics. The problem here is to pack n circles on a sphere as densely as possible [4]. Possible optimization criteria include: (a) maximizing the radius of non-overlapping circles, and (b) minimizing a specified potential function. The focus here is typically to find the *optimal* packing for a given n; finding approximately optimal solutions quickly, or generalizing the problem to other surfaces besides the sphere, have not received as much attention. Packing circles on a sphere is is of practical interest because it relates to multidimensional quantization schemes with minimum error, and also to information theory channel coding schemes that are most tolerant to noise [3]. Billiards-like simulations of colliding balls have been proposed as a method for solving such packing problems [8]. Some good information on circle packing and related problems is available on the World Wide Web:

Dave Rusin's sphere FAQ
http://www.math.niu.edu:80/~rusin/papers/known-math/spheres/

Spherical Codes **book in development by N. J. A. Sloane *et al.***
http://www.math.niu.edu:80/~rusin/papers/known-math/spheres/spherepack.sloan
Has best known solutions for $n \leq 130$.

John Leech's Tcl program to simulate repelling particles on the sphere
http://www.math.niu.edu:80/~rusin/papers/known-math/spheres/repulsion

David Eppstein's links on Covering and Packing, part of his "Geometry Junkyard"
http://www.ics.uci.edu/~eppstein/junkyard/cover.html

In physics and chemistry, particle methods have been in use for decades to simulate phenomena at a wide range of scales, from molecules to liquids to galaxies [6]. The forces simulated range from long range forces such as electrostatic attraction and repulsion, or gravitational attraction, to short range forces such as molecular bonds. Simulating n particles governed by long range forces is the celebrated "n body problem". A straightforward implementation would require $O(n^2)$ time, but fairly recently, fast "multipole" algorithms that operate in $O(n)$ time have been found [5]. They employ quadtrees for clustering. The short range forces simulated in the present paper could be simulated in linear time with such an algorithm, but such an algorithm would be unnecessarily complex.

In computational geometry, the problem of finding nearby points is called *range searching* [9]. Specifically, finding the points among a fixed set of points that lie within a sphere is called *bounded distance search*. Disappointingly, most of the computational geometry literature on this problem has (until recently) focused on static point sets, and has pursued algorithms with optimal asymptotic worst case cost as $n \to \infty$. In practice, we are usually interested in bounded n, expected case cost, and actual execution times. Thus, the constant factors in the time cost that are traditionally ignored in computational geometry are critical to a practical implementation. So most of the computational geometry literature is of little help for our practical problem. (Fortunately, it appears that the field of computational geometry is slowly shifting more of its attention to dynamic data structures and expected case cost.)

Closer to the field of computer graphics, particles exhibiting bubble-like attraction/repulsion characteristics have been used for parametric surface mesh generation and volume mesh generation (triangulating and tetrahedrizing a surface or a volume with control over spacing) [10, 11]. Our particle repulsion method has been generalized to nonuniform, anisotropic mesh generation in two dimensions, for which the triangles are allowed to vary in size and shape as a function of position [2]. Additional references on the use of particles in implicit surface display and interactive surface design are given in our previous paper [13]. Somewhat related to the polka dot aesthetic employed by our method is the work of Knowlton, in which arbitrary shapes are modeled out of spheres [7].

2 Sampling Implicit Surfaces

We review briefly the surface sampling method from [13], which motivates the development of the fast repulsion algorithm. The problem is to sample and control an implicit surface $F(x, y, z) = 0$ using particles distributed across the surface. That paper describes a method that can quickly achieve an approximately uniform particle density and maintain it in the face of rapid surface deformations. Figure 1 shows an example of a picture generated with this algorithm.

Particles are constrained to lie on the surface using a first order differential equation. This differential equation prescribes each particle's velocity as a function of the value of the function F at the particle's current position, the gradient of F there, and the particle's "desired velocity" due to repulsion. The repulsion velocity vector can point in any direction; the differential equation projects it to the surface tangent plane so that the point will stay (approximately) on the surface. A feedback term pulls the particle back to the surface if it strays off. The details of this formula are presented elsewhere [13]; here we focus on repulsion.

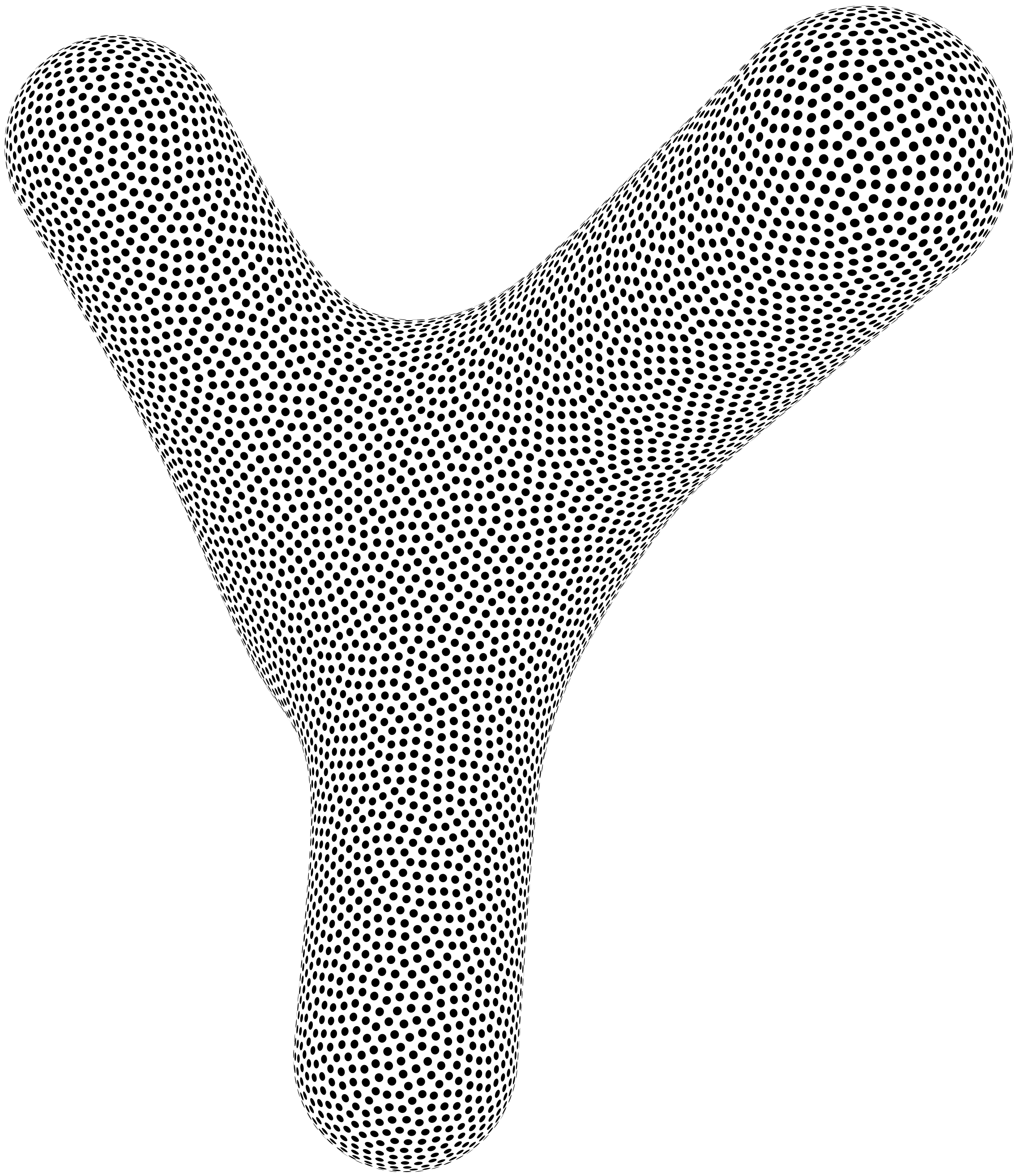

Figure 1: Implicit surface modeled using three blobby cylinder primitives, with 10,000 particles. Repulsion calculations for all particles take about .5 second on a 100 MHz SGI Crimson on this model, and redisplay using GL takes an approximately equal time.

2.1 Potential Energy

The repulsion velocity is derived by first defining the potential "energy" of one particle with respect to another to be proportional to a Gaussian function of the distance r between them. The energy of a particle is the sum of its pairwise energies with all other particles, and the repulsion velocity is taken to be proportional to the gradient of energy. We could use geodesic distance (shortest path on the surface) as the distance measure, but for simplicity and speed, we used Euclidean distance.

In a simulation of electric charges or masses under gravity, the potential energy would go as $1/r$, accelerations would be proportional to the gradient of energy ($-1/r^2$), and the differential equation would be second order. For such a potential function, and also for the the Lennard-Jones potential $r^{-12} - r^{-6}$ often used for liquid simulations [1], there is a singularity at $r = 0$. When Euler's method and constant time steps are used to integrate the differential equation, this singularity causes severe numerical problems. If two repelling particles approach each other too closely, they will shoot apart very fast (much faster than they would in real life, i.e. if infinitesimal time steps were used).

Since this is physically-*based* modeling, we are free to "play God" and make up almost any physical laws we please. In our simulation, we use a different potential and a first order differential equation, in which velocity (not acceleration) is proportional to the gradient of energy [12]. To avoid singularity difficulties, we choose a potential function that peaks at $r = 0$, is smooth, radially symmetric, and goes to zero in the distance. We chose the Gaussian function $e^{-r^2/2\sigma^2}$, since it is simple and meets all of these conditions.

2.2 Locality of Potential

Another benefit of the Gaussian is that it decays very rapidly; it drops to 1% of its peak value at about three standard deviations ($r_{\text{max}} = 3\sigma$), and contributions outside that range can in practice be neglected with little error. If 1% error is felt to be too large, then to achieve an error of 10^{-k}, the range can be set to $r_{\text{max}} = 2.15\sigma\sqrt{k}$. We define the *neighborhood* of a particle to be the sphere of radius r_{max} around it, and the *neighbors* of a particle to be all other particles within its neighborhood.

We should note, however, that we do not know if the Gaussian is optimal in some sense for this simulation. A slight problem with it is that when two particles are coincident ($r = 0$), the repulsion velocity is zero, so in the absence of any other particles, two coincident particles will stay coincident at this unstable equilibrium.

The Gaussian potential and its locality potential stand in contrast with the $1/r$ potential of gravity and electromagnetism, for which significant attraction and repulsion take place over great distances.

2.3 Adaptive Repulsion, Birth, and Death

A possible problem with this potential law is that if there are no particles in the neighborhood, the potential energy of the particle will be zero and it will not move. We have chosen rules for adaptive repulsion, birth and death of particles such as to make this possibility extremely unlikely or short-lived.

Each particle is given independent control over its "repulsion radius" σ. This allows some particles to "stake out" large pieces of surface, while others claim only small areas. The repulsion radius of a particle increases if its total repulsion from neighbors is too small, and it decreases if its total repulsion is too large. This is accomplished with a simple feedback mechanism, and it has the effect of keeping each particle "in communication with" (repelled by) an approximately constant number of other neighbors. Since each particle has its own σ, the neighborhood of each particle can differ: particles with large repulsion radii are influenced over greater distances than those with small repulsion radii.

To make the sampling process adaptive to interactive surface shape changes, we do not keep the number of particles fixed. Instead new particles are born where the local density is too low and existing particles die if the local density is too high. The desired density is a user-controlled parameter.

Before arriving at our method for adaptive birth and death, simpler schemes using a fixed number of particles were tried. These schemes relied on repulsion alone to redistribute the particles after a surface shape change. We found that these methods were orders of magnitude slower than adaptive birth and death. Whereas simple repulsion ran at a glacial pace, adaptive birth and death operated at the exponential pace of biological reproduction. New regions of surface created during interactive surface design were resampled in a small number of iterations much as plants or animals might migrate to and populate a new island in only a few generations.

We thus stress that the most important ingredients to fast sampling with particles on a dynamic surface are good rules for adaptive birth and death. We have elsewhere described one effective method for that [13] (it is certainly not the only method). Adaptive birth and death take care of gross redistribution of particles. Repulsion (or attraction/repulsion) seems to be the best method to fine tune the uniformity of the distribution.

2.4 Computing Repulsion

In this report, we set aside the issues of adaptive birth and death and adaptive control of repulsion radius, and focus on the more specific problem of calculating repulsion quickly for a given set of particles. We thus assume that we have n particles and that each particle is possibly repelled by its neighbors within a sphere of radius s_i. For this application, we sought uniformly distributed particles on an implicit surface using a Gaussian potential, but our repulsion method could easily be generalized to parametric surfaces, or to particles distributed throughout a 3-D volume. Generalizing the method to nonuniform particle distributions could also be done, but would be more difficult.

The naive algorithm for a first order simulation would be:

for $i \leftarrow 1$ to n do
    $velocity[i] \leftarrow 0$
    for $j \leftarrow 1$ to n do
        if $j \neq i$ then
            $velocity[i] \leftarrow velocity[i]$ + REPULSION(i, j)

where REPULSION(i, j) returns the repulsion velocity on particle i due to particle j. This algorithm would obviously have cost $O(n^2)$.

3 Fast Algorithm

Since we assume a short-range potential whose effect is negligible outside some range, we could possibly speed up the algorithm if, for each particle, we do not visit all of the other particles, but only the particle's neighbors, that is, only those that exert non-negligible repulsion on it. Furthermore, we expect, because of the design of the rules for adaptive repulsion, birth, and death summarized above, that when particles have reached equilibrium, they will be approximately uniformly distributed across the surface, that the repulsion radius of each particle will be approximately equal, and therefore the number of neighbors of all particles will be roughly equal, and bounded.

We thus seek a method for answering spherical range queries of the form

PARTICLES_IN_SPHERE(*Point p, Real r*):
    return the set of all particles within radius r of point p

as quickly as possible. Queries are made in repetitive mode [9], that is, the database is static during a string of queries. But the database is dynamic between batches of queries: particles move, appear, and disappear. The number of particles and their typical radii could change quickly over time. We assume, however, that particles lie within a bounded volume of space and that they are approximately uniformly distributed.

For our application, if the spherical range query operates in time proportional to its output size, and the average number of particles repelling a given particle is constant, then we could calculate mutual repulsion in time $O(n)$. That is indeed what we achieve.

While adaptive data structures such as k-d trees, octrees, and BSP trees could be used for this purpose, we tried, and had good success with a simpler data structure: a regular grid. Such approaches are sometimes called *bucketing*. We expected this to work well because of the approximately uniformly distribution of points.

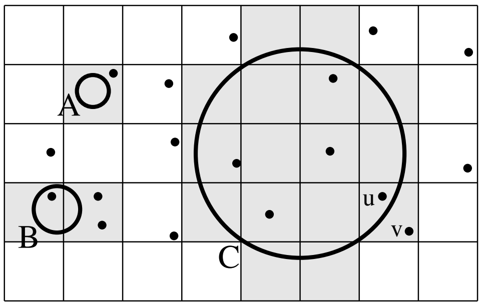

Figure 2: Circular range queries. The query circle A involves a single grid cell (in gray), query B involves two cells, and query C involves 16 cells. Within cells intersected by the circle, some particles such as u will be inside, while others such as v will be outside.

3.1 Fast Repulsion with a Grid

The basic idea is to divide space into a grid of parallelepiped cells, each of which consists of a linked list of the particles within it, if any. Since queries are spherical, cells should probably be cubical, and not elongated. Each particle is listed in a single cell's list. Queries can be answered quickly by finding all cells that intersect or lie interior to the sphere of radius r centered at p. The union of those cell's particle lists is a superset of the particles we want. For cells that lie entirely inside the sphere, all of their particles are within range, but for cells that are intersected by the sphere surface, some of the particles may be within range while others may be out of range. Figure 2 shows the 2-D analogue.

In our application, particles are moving over time, so they will be entering and leaving cells. Therefore, inserting and deleting particles from the cell lists must be as fast as possible. We also have a changing population of particles: cells are being born and dying, and the number of particles can go from 1 to thousands in a few seconds. To handle a fast-changing population, we employ a dynamic grid.

For range queries with fixed radius, it can be best to choose the grid size just bigger than the radius (this is often used in liquid simulations [1, p. 150]). Thus, the neighbors will all lie within a $3 \times 3 \times 3$ grid neighborhood. For our purposes, the queries can vary in radius, so the cell size should not be tied to the query radius, and scan conversion of a sphere is required to find neighboring cells.

Our grid will perform reasonably efficiently as long as the queries touch a handful of cells – not too many, not too few. If the cell size is very large, then the cell lists will be long and we will spend too much time traversing the lists to discard those particles that are out of range. The query might only involve one or two cells, however, so the time spend scan converting the cells intersected by the sphere will be negligible. If the cell size is very small, the scan conversion cost will be very high (many cells must be visited). Many cell lists will be empty or very short, however, and the number of particles that we test that are actually out of range will be small, so the time spent discarding out-of-range particles will be negligible. The best performance will occur for intermediate cell sizes, when the list traversal and scan conversion costs are in reasonable balance.

Our grid method therefore monitors the queries and it periodically (every redisplay) re-evaluates the grid resolution. Since redistributing several thousand particles to a new grid is not a cheap operation, we do not want to "re-grid" too often. The philosophy is much like that of garbage collection: you want to keep the system operating efficiently, but you don't want to be so aggressive with "optimization" that you slow things down overall. Our algorithm therefore leaves the grid unchanged if the average query radius is not excessively big or small. If the grid is deemed inefficient then a new grid resolution is chosen such that the average query radius will fall near the empirical optimum multiple of grid cell size. Memory for the new grid is allocated, the particles from the old grid are redistributed among the cells of the new grid, and the old grid is discarded.

3.2 Tuning for Our Application

For our application, particles lie on a bounded surface, and we tend to be editing single surfaces at a time or a small number of closely-spaced surfaces. So for our purposes, a grid with equal numbers of rows in x, y, and z is satisfactory. For speed of indexing, the number of cells along each axis was taken to be a power of two. Thus, the grid will have $2^k \times 2^k \times 2^k$ cells, for some small k. We have used $1 \leq k \leq 6$ for up to 10,000 particles, with good results.

The surface and the particles on it can move over time, but rather than have the grid cells translate to follow the surface, which would probably cause unnecessary cell-to-cell particle migration, we freeze the position of each cell in space and we tile space periodically. Thus, the grid does not cover just a bounded cube within 3-D space, it actually covers all of 3-D space with a periodic cubical tiling; the grid wraps around in x, y, and z. The grid can thus be thought of as a hashing scheme for 3-D point location in which, unlike pseudorandom hashing schemes, we want to preserve spatial coherence.

Through extensive empirical tuning, we have found that the following parameter settings work fastest: The ratio of the average query radius to the cube size should be between .7 and 1. The ratio of the diameter of the particle cloud (imagine a bounding sphere around all of the particles) to the size of the grid in world space distance, should be between .5 and 1.2. When the query radii go out of this range, the cubesize is adjusted upward or downward, while keeping the grid resolution fixed. That is, if the entire particle cloud is less than half the domain of the grid (domain size = cubesize times number of cells along that dimension), then the domain is too big, and if it's more than 20% bigger than the domain of the grid, then too much wraparound is probably occurring, because the domain is too small. In the first case, the resolution along each dimension is halved and the size of the domain is halved, while in the second case, the resolution is doubled and the size of the domain is doubled In both cases the cubesize remains unchanged. If the queries are inside the desirable ranges, then the grid is left unchanged.

We found that when the system was operating near optimal performance, there were, on average: 10-20 cells within the neighborhood, 40% of them were empty, there were 27 particles in the cells examined, but that only 7 of them were within range (inside the query sphere). Thus, the average number of particles repelling a given particle was 7. If the statistics of the particles differed from ours (different average number of repelling neighbors, etc.) then the optimal parameter values would probably change.

4 Code

We include the C++ code for the grid data structure, since it is relatively short. The code shows the precise rules used for indexing grid cells, and for regridding.

——————- particle.h

```
class particle {
    public:
        vec3 p;                 // position
        particle *next;         // next particle in linked list (for pds)
};
```

——————- pds.h

```
#ifndef PDS_HDR
#define PDS_HDR

// pds: particle data structure, for fast neighbor finding in 3-D
// Paul Heckbert        28 June 1994

#include "particle.h"

class pds {                     // data structure for a set of particles
        particle **bucket;      // array of cubical hash buckets;
                                // each contains a linked list of particles

        int nbit;               // number of bits per dimension for hash table
        int mask;               // for hashing, = 2^nbit-1
        float cubesize;         // size of a cube

        // statistics for testing
        int nmove_attempt;      // number of calls to move()
        int nmove_actual;       // number of particles actually moved
        int nqsingle, nqwrap, ncrange, ncempty; // visit_neighbor stats

    public:
        int npart;              // number of particles
        int debug;              // level of debugging (0=off)
```

```
int manualoverride;      // for testing query_advisory

inline int nb()          // number of bits
    {return nbit;};
inline float csize()     // cube size
    {return cubesize;};
inline int num()         // # of buckets in x, y, or z
    {return 1<<nbit;};
inline int nbucket()     // total number of buckets
    {return 1<<nbit*3;};
inline float size()      // size of entire cubical space
    {return num()*cubesize;};

pds(int nbit, float size);       // constructor

void query_advisory(float diam, float ravg);
                         // client is telling us stats about his queries
                         // call regrid if cubes are wrong size/number
void regrid(int nbit, float size);
                         // change nbit & cubesize, rebuild datastruc

inline particle **bucketi(int ix, int iy, int iz)
    {return &bucket[(ix<<nbit|iy)<<nbit|iz];}
                         // return bucket given bounded integer indices
                         // (each index in the range [0..mask])

inline particle **bucketf(particle *p)
    {return bucketi((int)floor(p->p.x/cubesize)&mask,
                    (int)floor(p->p.y/cubesize)&mask,
                    (int)floor(p->p.z/cubesize)&mask);}
                         // return bucket given unbounded float coords
                         // note: many cubes map to the same bucket

void insert(particle *p);        // add particle to set
void del(particle *p);           // delete particle from set
void address_change(particle *pold, particle *pnew);
                                 // particle's address changed
void move(particle *p, vec3 &d);// update particle's position to p->p+d

void visit_neighbors(particle *p, float r,
    void (*proc)(particle *pi, particle *pj));
                         // visit all particles within radius r of
                         // particle p, calling proc with arguments
                         // pi=p, pj=neighbor.
                         // note: might visit a few with distance > r
```

```
        void visit_stats_init();
        void visit_stats_get(int &nsingle, int &nwrap, int &ncube, int &nempty);
                                        // get stats about visit_neighbors
        void ds_stats();                // print data structure statistics
};

#endif
```

———————- pds.C

```
// pds: particle data structure, for fast neighbor finding in 3-D
// Paul Heckbert        28 June 1994

#include <bstring.h>
#include <assert.h>
#include <clock.h>
#include "pds.h"

pds::pds(int nbit, float size) {
    if (nbit<0) nbit = 0;
    pds::nbit = nbit;
    mask = num()-1;
    bucket = new particle *[nbucket()];
    bzero(bucket, nbucket()*sizeof(bucket[0])); // mark buckets as empty
    cubesize = size/num();
    npart = 0;
    debug = 0;
    manualoverride = 0;
    nmove_attempt = 0;
    nmove_actual = 0;
}

void pds::query_advisory(float diam, float ravg) {
    // client is telling us stats about his queries:
    //   diam is diameter of particle cloud, in world space
    //   ravg is average radius of queries, in world space
    // call regrid if cubes are wrong size/number
    if (manualoverride || diam<=0) return;
    float s = size();
    int n = nbit;
    ravg /= cubesize;           // now ravg is in cube units
    if (ravg<.7 || ravg>1.) s *= ravg/sqrt(.7*1.);
        // if queries too big or too small, then grow or shrink cubesize,
```

```
        // respectively
        // sqrt(a*b) is the geometric mean of the numbers a and b
    if (diam<.5*s && n>0) {n--; s /= 2;}
        // if points only take up a fraction of the space, use fewer buckets
    else if (diam>1.2*s && n<6) {n++; s *= 2;}
        // if points wrap around, use more buckets

    static double tprev = 0;
    double t = clock_cpu_read();
    if (n!=nbit || s!=size() && t-tprev>1.) {
        // if there's need for a change, and at least one second has gone by
        if (debug)
            printf("    query_advisory: diam=%g ravg=%g\n", diam, ravg);
        regrid(n, s);
        tprev = t;
    }
}

void pds::regrid(int nbit, float size) {
    if (nbit<0) return;
    if (debug) printf("REGRID from size=%g,%d-bit to %g,%d-bit\n",
        pds::size(), pds::nbit, size, nbit);
    if (cubesize==size/num() && pds::nbit==nbit) return;

    // create a linked list of all particles in set, empty the buckets
    particle *first = 0, *p, *q;
    int i;
    for (i=0; i<nbucket(); i++) {
        p = bucket[i];
        if (p) {
            for (q=p; q->next; q=q->next);      // find end of bucket's list
            q->next = first;
            first = p;
        }
    }

    if (pds::nbit!=nbit) {              // remake bucket array, if necessary
        delete [nbucket()] bucket;
        pds::nbit = nbit;               // NOTE: this changes num(), nbucket()
        mask = num()-1;
        if (debug) printf("now %d buckets\n", nbucket());
        bucket = new particle *[nbucket()];
    }
    bzero(bucket, nbucket()*sizeof(bucket[0])); // mark buckets as empty
    npart = 0;
```

```
    cubesize = size/num();

    // redistribute particles to new cells
    for (p=first; p; p=q) {
        q = p->next;
        insert(p);
    }
}

void pds::insert(particle *p) {          // add particle to set
    particle **b;
    b = bucketf(p);
    p->next = *b;
    *b = p;
    npart++;
}

void pds::del(particle *p) {             // delete particle from set
    particle **b, *o;
    b = bucketf(p);
    if (*b==p)
        *b = p->next;
    else {
        for (o=*b; o && o->next!=p; o=o->next); // find predecessor to p
        assert(o);                       // particle p wasn't in the list!
        o->next = p->next;               // link around p
    }
    npart--;
}

void pds::address_change(particle *pold, particle *p) {
    // particle is currently at p, but our data structure thinks it's at pold
    particle **b, *o;
    b = bucketf(p);
    if (*b==pold)
        *b = p;                          // fix pointer
    else {
        for (o=*b; o && o->next!=pold; o=o->next);      // find predecessor
        assert(o);                       // particle pold wasn't in the list!
        o->next = p;                     // fix pointer
    }
}

void pds::move(particle *p, vec3 &d) {  // update particle's position to p->p+d
    nmove_attempt++;
```

```
    if (d.x==0 && d.y==0 && d.z==0) return;
    particle **bo, **bn;
    bo = bucketf(p);                    // old bucket
    p->p += d;                          // move particle
    bn = bucketf(p);                    // new bucket
    if (bn==bo) return;

    // delete p from bucket bo's list
    if (*bo==p) *bo = p->next;          // link around p
    else {
        particle *o;
        for (o=*bo; o && o->next!=p; o=o->next);  // find predecessor to p
        assert(o);                      // particle p wasn't in the list!
        o->next = p->next;              // link around p
    }

    // insert p into bucket bn's list
    p->next = *bn;
    *bn = p;
    nmove_actual++;
}

void pds::visit_neighbors(particle *p, float r,
    void (*proc)(particle *pi, particle *pj)) {
                            // visit all particles within radius r of
                            // particle p, calling proc with arguments
                            // pi=p, pj=neighbor

    // indices of cube containing point p->p
    int xc = (int)floor(p->p.x/cubesize);
    int yc = (int)floor(p->p.y/cubesize);
    int zc = (int)floor(p->p.z/cubesize);

    // find bounding box of sphere of radius r around p->p
    int x0 = (int)floor((p->p.x-r)/cubesize);
    int y0 = (int)floor((p->p.y-r)/cubesize);
    int z0 = (int)floor((p->p.z-r)/cubesize);
    int x1 = (int)floor((p->p.x+r)/cubesize);
    int y1 = (int)floor((p->p.y+r)/cubesize);
    int z1 = (int)floor((p->p.z+r)/cubesize);

    int size = x1-x0;
    if (y1-y0>size) size = y1-y0;
    if (z1-z0>size) size = z1-z0;
    size++; // largest dimension of parallelepiped of cubes
```

```
if (size==1) {                              // *1. sphere fits in 1 cube
    particle *q = *bucketi(x0&mask, y0&mask, z0&mask);
    for (; q; q=q->next)
        (*proc)(p, q);
    nqsingle++;
    ncrange++;
}
else if (size > num()) {                    // *2. sphere wraps around!
    // visit every particle
    // (we don't use the usual sphere scan conversion method, because
    // wraparound could cause that to visit some bucket repeatedly)
    int i;
    particle *q;
    for (i=0; i<nbucket(); i++)
        for (q=bucket[i]; q; q=q->next)
            (*proc)(p, q);
    nqwrap++;
    ncrange += nbucket();
}
else {                                      // *3. sphere intersects >1 cube
    // visit all cubes within bounding box
    float r2 = r*r;
    int x, y, z;
    float dx, dy, dz, r2_x2_y2;
    particle *q;
    for (x=x0; x<=x1; x++) {
        // find vector (dx,dy,dz) to point of cube (x,y,z) nearest to p
        dx = x<xc ? (x+1)*cubesize-p->p.x :
             x>xc ? x*cubesize-p->p.x : 0;
        for (y=y0; y<=y1; y++) {
            dy = y<yc ? (y+1)*cubesize-p->p.y :
                 y>yc ? y*cubesize-p->p.y : 0;
            r2_x2_y2 = r2-dx*dx-dy*dy;
            if (r2_x2_y2 < 0) continue;
            for (z=z0; z<=z1; z++) {
                dz = z<zc ? (z+1)*cubesize-p->p.z :
                     z>zc ? z*cubesize-p->p.z : 0;
                // skip cubes that don't intersect sphere of radius r
                if (dz*dz < r2_x2_y2) {
                    q = *bucketi(x&mask, y&mask, z&mask);
                    if (!q) ncempty++;
                    for (; q; q=q->next)
                        (*proc)(p, q);
                    ncrange++;
```

```
                    }
                }
            }
        }
    }
    if (debug>3)
        printf("visited %d*%d*%d=%d buckets, r=%g (%.2f cubes)\n",
            x1-x0+1, y1-y0+1, z1-z0+1, (x1-x0+1)*(y1-y0+1)*(z1-z0+1),
            r, r/cubesize);
}

void pds::visit_stats_init() {
    nqsingle = 0;
    nqwrap = 0;
    ncrange = 0;
    ncempty = 0;
}

void pds::visit_stats_get(int &nqsingle, int &nqwrap, int &ncrange, int &ncempty) {
    nqsingle = pds::nqsingle;
    nqwrap = pds::nqwrap;
    ncrange = pds::ncrange;
    ncempty = pds::ncempty;
}

#define CLEN 17

void pds::ds_stats() {          // print data structure statistics
    if (!debug) return;

    particle *p;
    int i, c, cmax = 0, nbne = 0, count[CLEN+1];
    for (c=0; c<=CLEN; c++)
        count[c] = 0;
    for (i=0; i<nbucket(); i++) {
        for (c=0, p=bucket[i]; p; p=p->next, c++);
                                // find length of bucket's list
        if (c>0) nbne++;        // nbne: Number of Buckets, Non-Empty
        if (c>cmax) cmax = c;
        count[c<CLEN ? c : CLEN]++;
    }
    printf("pds: %d particles, %.0f%% moved, list length: avg=%.1f, longest=%d\n",
        npart,
        nmove_attempt ? 100.*nmove_actual/nmove_attempt : 0,
        nbne ? (float)npart/nbne : 0,
```

```
        cmax);
    nmove_attempt = 0;
    nmove_actual = 0;

    if (debug>3) {              // print bucket list length histogram
        int cm = cmax<CLEN ? cmax : CLEN;
        printf(" len: ");
        for (c=0; c<=cm; c++)
            printf("%4d", c);
        printf("%s\nfreq: ", cmax>cm ? "+" : "");
        for (c=0; c<=cm; c++)
            printf("%4d", count[c]);
        printf("\n");
    }
}
```

5 Conclusions

The optimizations described here allowed us to increase the maximum number of particles simulated from 500 to 10,000, and made the latter case run 160 times faster than our initial algorithm, and sped up the theoretical cost from $O(n^2)$ to $O(n)$.

6 Acknowledgements

Andy Witkin developed the constraint method and played a vital role in the development of the rules for particle repulsion, birth, and death. This report was written at the encouragement of John Hart. This work was supported by ARPA contract F19628-93-C-0171 and NSF Young Investigator award CCR-9357763.

References

[1] M. P. Allen and D. J. Tildesley. *Computer Simulation of Liquids*. Clarendon Press, Oxford, 1987.

[2] Frank J. Bossen and Paul S. Heckbert. A pliant method for anisotropic mesh generation. In *5th Intl. Meshing Roundtable*, pages 63–74, Oct. 1996. http://www.cs.cmu.edu/∼ph.

[3] J. H. Conway and N. J. A. Sloane. *Sphere Packings, Lattices and Groups*. Springer-Verlag, 1988.

[4] Hallard T. Croft, Kenneth J. Falconer, and Richard K. Guy. *Unsolved Problems in Geometry*. Springer-Verlag, 1991.

[5] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. Computational Physics*, 73:325–348, 1987.

[6] R. W. Hockney and J. W. Eastwood. *Computer Simulation Using Particles*. McGraw-Hill, New York, 1981.

[7] Ken Knowlton. Computer-aided definition, manipulation and depiction of objects composed of spheres. *Computer Graphics*, 15:352–375, December 1981.

[8] Boris D. Lubachevsky. How to simulate billiards and similar systems. *J. of Computational Physics*, 94:255–283, 1991.

[9] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: an Introduction*. Springer-Verlag, 1985.

[10] Kenji Shimada. *Physically-Based Mesh Generation: Automated Triangulation of Surfaces and Volumes via Bubble Packing*. PhD thesis, ME Dept., MIT, 1993.

[11] Kenji Shimada and David C. Gossard. Bubble mesh: Automated triangular meshing of non-manifold geometry by sphere packing. In *Third Symp. on Solid Modeling and Appls.*, pages 409–419, May 1995. http://www.trl.ibm.co.jp/projects/s7340/meshing/meshingE.htm.

[12] Andrew Witkin and David Baraff. An introduction to physically based modeling. Notes for course at Carnegie Mellon University and at SIGGRAPH, 1997. http://www.cs.cmu.edu/afs/cs/user/baraff/www/15-863/index.html.

[13] Andrew P. Witkin and Paul S. Heckbert. Using particles to sample and control implicit surfaces. In *SIGGRAPH 94 Proceedings*, pages 269–277, July 1994. http://www.cs.cmu.edu/∼ph.