Real-Time Particle Isosurface Extraction

Ilya D. Rosenberg*
Ken Birdwell†

Abstract

Particle-based methods are commonly used for simulation of fluid, gelatinous, and gooey substances. Recently, there has been great interest in using these methods in interactive applications such as surgical simulation, surface modeling, and video games. While modern computers are easily capable of simulating thousands of particles in real time, in many cases, a surface must be generated over the particles in order to realistically render the output of such a simulation. This surface extraction step is often the bottleneck in such applications due to the high computational cost and/or large memory requirements of common surface extraction algorithms. We present a new approach for fast, high quality polygonization of isosurfaces that can be used to render surfaces in real-time over thousands of particles in an unbounded spatial domain using a small amount of working memory, and compare it to existing algorithms. Furthermore, we extend our approach to generate polygon faces in back-to-front rendering order for transparent surfaces. Finally, we demonstrate the effectiveness of this new technique with several interactive scenarios showing complex interaction between fluid entities and dynamic objects in a virtual environment.

1 Introduction

In computer graphics, particles are a common and convenient representation for modeling physical entities, especially those that can undergo drastic changes in shape and topology. By adding inter-particle forces, a particle system can be made to act like fluid, solid, and anything in between, or it can be procedurally animated to achieve a specific shape or motion. Furthermore, in interactive applications, particles can easily handle interaction with users and world geometry. In order to create compelling simulations of most phenomena, a large number of particles is necessary. To give the user the illusion that the simulated phenomena are composed of an inordinate number of particles, as they are in nature, it is often desirable to coat the particles with a smooth surface so that the individual particles are not visible. A common technique for generating such a surface is to define an implicit function over the particles in space, and to render the surface wherever that function is equal to a predetermined threshold value. Such a surface is commonly referred to as an implicit surface or isosurface.

Blinn [Bli82] first proposed the use of implicit surfaces as a model for ray-tracing electron density maps over molecular structures and suggested the use of implicit surfaces as a general model for three-dimensional shapes. Reeves [Ree83] proposed the use of particle systems as a technique for modeling and animating fuzzy objects and rendering particles simply by additively blending (splatting) them into a buffer. Sims [Sim90] expanded upon Reeves' work by using particles for modeling other natural phenomena such as waterfalls, fire, and tornadoes. In more recent work, taking advantage of recent hardware, Adams *et al.* [ALD06] proposed a technique where on the order of 100,000 particles are rendered as transparent sprites on the GPU. Because the number of particles is so large, the individual particles become less visible, creating the illusion of a smooth surface. However, the approach suffers from the same problem as earlier work since the illusion is broken at the edges of the surface where individual particles are still visible.

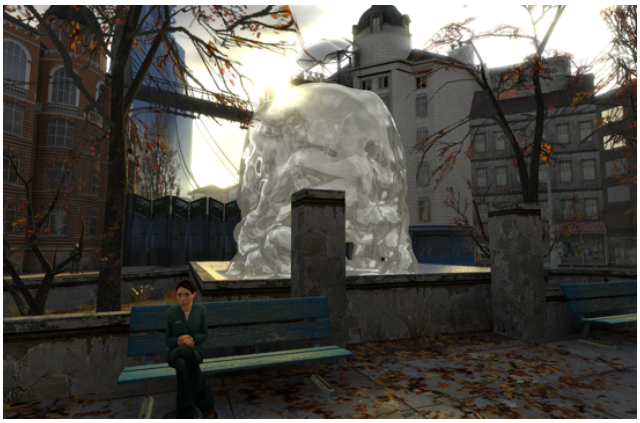

Figure 1: *A fountain rendered in real-time using our technique in a fully interactive video game environment.*

Szeliski and Tonnesen [ST92] and Witkin and Heckbert [WH94] proposed an alternate approach which uses a second class of oriented particles, known as surfels, to track implicit surfaces. More recently, Müller *et al.* [MKN*04] used a large number of surfels to track texture and surface detail over slowly deforming models. The drawback of these techniques is that for a small number of physically simulated particles (phyxels), several orders of magnitude more surfels must be simulated to cover the surface. These surfels must physically interact with both the phyxels and each other. Furthermore, the physical simulation requires frame coherence, otherwise surfels lose track of the surface. Finally, neither the ray-tracing, particle splatting nor surfel tracking approaches produce polygons as output. Thus, they can not fully take advantage of the pipeline used in the majority of interactive graphics applications where geometry is generated on the CPU, and the GPU is used for polygon based texturing, shading and rasterization.

For these reasons, a technique that can efficiently polygonize the isosurface, or in other words, generate a triangle mesh is preferable. Although there are techniques such as marching tetrahedra proposed by Bloomenthal [Blo88] and marching triangles proposed by Hilton and Illingworth [HI97] which can polygonize surfaces, marching cubes proposed by Lorensen and Cline [LC87] is by far the most commonly used approach because is straightforward to implement, does not require coherence between frames, and is arguably the fastest way to extract surfaces from a 3D volume of scalar data.

In the marching cubes approach, a volume of space is divided into individual cubes. Isosurface field values and normals are calculated at the corners of the cubes, and an eight bit lookup is computed based on whether the field values are larger or smaller than the isosurface threshold. The lookup is used to index a static data struc-

*email: ilya@cs.nyu.edu
†email: kenb@valvesoftware.com

I3D 2008, Redwood City, California, February 15–17, 2008.
ACM 978-1-59593-983-8/08/0002

ture which indicates how vertices on the edges of the cube are to be connected to form a set of triangles. The vertices are computed by linearly interpolating the positions and normals at the corners based on the field values. The main drawback of marching cubes is that it requires a regular 3D volume of scalar field values as input. This is not a problem in applications such as medical imaging, where the data is naturally captured as a 3D volume. However, in interactive applications where the input is a set of procedurally generated particles, there is no volume of scalar data a-priori, and it is preferable to sample the field values defined by the particles judiciously by traversing, calculating isosurface values for, and polygonizing only the the cubes that contain segments of the isosurface.

To address this problem, Wyvill *et al.* [WMW86b] proposed a surface-following (continuation) approach which, starting at a point which contains a surface, continued by traversing neighboring cells which contain surface in a depth-first or breadth-first manner. Additionally, the paper noted that it is preferable to force particle fields to have a limited radius of influence, and suggested using a voxel grid to look up particles that have influence at particular locations in the volume. It mentioned that this lookup structure becomes more accurate as the voxel size shrinks to the size of the marching cube grid, but that the memory footprint of the structure and the cost of lookup/insertion increases as the grid size becomes smaller. It also noted that because cube corners can be shared by as many as eight cubes, storing these values in a 3D cache grid and re-using these values is much more efficient than re-calculating them from scratch each time they are needed. However, it did not suggest any way to find and eliminate unneeded cached values during the course of rendering a single frame. Finally, it proposed using a hash function that wraps each coordinate to keep it within the the dimensions of the 3D cache to allow rendering in an arbitrarily large unbounded volume.

Triquet *et al.* [TMC01] detailed several important considerations for extending these techniques for fast isosurface polygonization of particles. These included an improved isosurface field function, finding start points (seeds) from which to traverse the surface by evaluating field values along a fixed direction starting at the center of each particle, and reusing vertex calculations at edges that are shared between cubes in addition to reusing corner calculations. This paper also gave a cursory description of the problem of looking up particles that contribute to a given isosurface field value, and came to the same conclusion as [WMW86b], that the voxel size for the lookup data structure must be coarser than the marching cubes grid, yet failed to provide an analysis of the optimal grid size. Furthermore, the paper noted that using a hash map for making isosurface extraction unbounded slows the algorithm, yet failed to provide an alternative.

Teschner *et al.* [THM*03] proposed an alternate hash function along with analysis of its performance, and additionally provided an analysis of the cost of inserting/looking up elements in a volumetric grid and how it varies with grid size. Although they used tetrahedra as a primitive rather than spheres, their findings showed that the optimal performance of a grid based lookup occurs when the grid is approximately the same size as a tetrahedron's edge length. In our experiments with the same sort of lookup, we found a similar relationship for particles, where the optimal performance occurred when the size of grid cells was approximately equal the radii of the field of influence of the particles. Unfortunately, at this grid size, this type of lookup cache is very imprecise, yielding approximately 6.5 times more particles than the number that actually contribute to the field value at a given point (See Section 6 for more details).

In this paper, we present a novel real-time particle isosurface extraction technique that overcomes the deficiencies of these approaches. Our techique consists of the following major contributions:

1. A spatial decomposition algorithm which divides the volume to be rendered into blocks while avoiding seams or inconsistencies in the surface between adjacent blocks. This naturally limits the upper bound of memory usage, eliminates the need for hashing, allows rendering of particles in an unbounded volume, and enables multi-threaded rendering on multi-core computers.
2. An algorithm for extracting the isosurface within a block which we refer to as *Marching Slices* that avoids excessive growth of cached data by polygonizing the isosurface in a slice by slice fashion. Our algorithm guarantees a single visit to each cube intersecting the isosurface without keeping global information on all visited cubes by discarding slices of cached data that will not be reused in the future. Moreover, because our algorithm renders in slices, it improves the locality of memory accesses, and can be easily extended to output triangles in a back-to-front order when rendering transparent surfaces.
3. A fast and exact particle lookup technique which speeds up isosurface field value calculations by finding all the particles within a fixed influence radius that contribute to a sample point without finding any particles outside of that radius. In contrast, alternative lookup techniques return many particles that are outside of the influence radius, thereby wasting time in field value calculations.

Although blocking in order to subdivide large problems into smaller ones, processing of volumes one slice at a time, and lookup of particle influences by projecting them onto a 2D surface has been done before in other contexts, we believe we are the first to combine these techniques into a unified approach which solves many, if not all of the practical real-world problems that one skilled in the art may encounter when using isosurface-extraction of particle data sets in interactive applications. The rest of the paper details how these three components work in concert to enable memory efficient, spatially unbounded real-time rendering. Furthermore, we present a detailed comparison between this new technique and other particle-based isosurface extraction approaches over multiple scenarios, and further demonstrate its performance in a complex interactive game environment. Due to space limitations, we do not address issues relating to particle simulation or animation in this paper other than mentioning that we use a separate set of data structures (which operate at a much coarser level than the data structures used for rendering) for inter-particle collision detection, and a commercially available physics engine for collisions between particles and the world.

2 Algorithm Overview

The input to our algorithm is a list of particles containing (x,y,z) coordinates for position and the output is a list of triangles represented by a vertex and index buffer. The algorithm requires that the implicit field produced by each particle is radially monotonic, continuous and has a limited radius of influence which we call the cutoff radius R_c. We employ the following function described by Triquet [TMC01] because it can be computed efficiently with a few multiplication and addition operations:

$$f(r) = \left\{ \begin{array}{ll} (r/\sqrt{2}R_c)^4 - (r/\sqrt{2}R_c)^2 + 0.25 & \text{if } r < R_c \\ 0 & otherwise \end{array} \right\}$$

In addition to R_c and a list of particles, our algorithm takes the size of a cube S, the threshold value where surfaces are generated T, and possibly other user-defined data. Without user-defined data, a field calculation routine calculates field values and normals which are

then interpolated by a vertex calculation routine to output vertex positions and normals. The user-defined data can be used along with custom user-defined isosurface field and vertex calculation routines to generate vertices with additional data such as colors, or texture coordinates.

Once all the input data is specified, the algorithm proceeds by subdividing the render volume into a list of blocks. Proceeding one block at a time, it builds a particle lookup cache for accelerating field value calculations, and then finds the seed cubes at which polygonization will begin. It then proceeds to use the marching slices algorithm to polygonize the cubes inside each block slice by slice, using field and vertex calculation routines to generate vertices and writing the output into a user-specified vertex and index buffer.

3 Block Subdivision

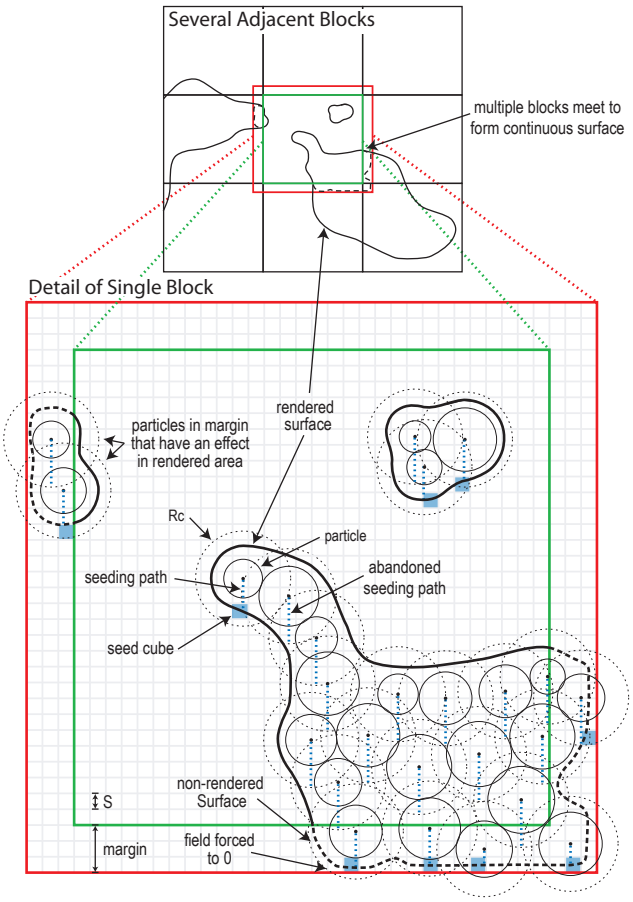

Figure 2: *2D illustration of block subdivision. The green box contains the polygonized surface and red box outlines the margin.*

We divide the volume containing particles into disjoint blocks, which can be processed independently, calculating the surface sequentially using only the memory footprint for a single block, while producing a seamless final result (Figure 2). Starting from the global set of particles, we build a list of blocks, each of which keeps its own list of particles. Each block is only responsible for rendering the cubes inside it, but to maintain consistency with adjacent blocks, it must consider all particles beyond its boundary that may have a field contribution in the interior of the block. Thus, we additionally insert all particles within a distance of R_c from the block into the list.

We expand the original dimensions of the block by a margin of cubes to enclose a distance of R_c, the radius of influence of a particle. Marching slices operates over the expanded volume, but will not output mesh triangles for cubes in the margin. This guarantees extraction of the isosurface of all particles including those located in the margin. Because the marching slices algorithm (Section 4) relies on the assumption that the isosurface is closed in order to guarantee a traversal of the entire surface, we force field values to disappear at sample points on the outer margin boundary by instrumenting our lookup cache in such a way that it does not return particles for samples on the boundary (Section 6).

The overhead of our block subdivision approach is the cost of traversing the cubes that are in the margin. For a typical $100 \times 100 \times 100$ cube block, with a margin that is 5 cubes wide, 25% of the cubes in the full $110 \times 110 \times 110$ volume are margin cubes. Since no vertices are generated for margin cubes, and full field calculations are not necessary (only the sign of the field is needed), the cost of traversing a cube in the margin is approximately half that of a normal cube. Thus, in the case that particles are randomly distributed within the volume, the extra overhead of blocking is approximately 12% of total work. This overhead is small compared to the cost of hashing, the alternate approach for rendering unbounded volumes, as hashing must be done both for corner retrieval and for particle lookups (Section 6). Furthermore, block subdivision decreases memory usage by limiting the number of corner values and particles kept in cache. It is also useful for applications such as surface modeling, where only a small subset of particles move between frames. Here, the blocks with no moving particles can be redrawn without polygonization simply by re-outputting old vertex and index buffers. By not rendering invisible blocks, block subdivision can be used for visibility culling, and can also be used for controlling level of detail by varying S between blocks (although care needs to be taken to avoid seams). Finally because each block is rendered independently of other blocks, it is straightforward to have each block rendered in a separate thread on multi-core hardware.

4 Marching Slices

The intuition for the marching slices algorithm is that memory usage would be greatly reduced if it were possible to polygonize isosurfaces one slice at a time, caching only the data necessary for the current slice. However, this poses two serious challenges. The first is that because the field values are not known a-priori, it is not possible to simply march back-to-front through the volume. Instead, it is necessary to march along connected cubes containing the isosurface starting at a seed cube in a fashion where all the connected cubes in the current slice are traversed before moving on to the next slice, with the ability to march through the volume arbitrarily in both forward and reverse order depending on how the cubes are connected. The second challenge is that in order to realize a reduction in memory usage, we need to discard cached data that will not be reused and ensure that there are no redundant visits to completed cubes.

To keep memory usage to a minimum, we use a sparse set of data structures over the volume (Figure 3). We treat each block as a vertical array of N_z *slabs* of size $N_x \times N_y \times 1$ cubes. We refer to the vertices of a cube as *corners*, reserving the term *vertex* for a vertex that is output by the algorithm for rendering. The slabs are numbered from 0 to N_z where z increases upwards. Each slab is bounded above and below by a two-dimensional slice, with adjacent slabs sharing slices. The slices are numbered from 0 to $N_z + 1$. For each of the N_z slabs, we store a linked list, called the *todo list* containing integer (x, y) tuples which represent cubes in the slab that may need to be rendered. For any slab which is being poly-

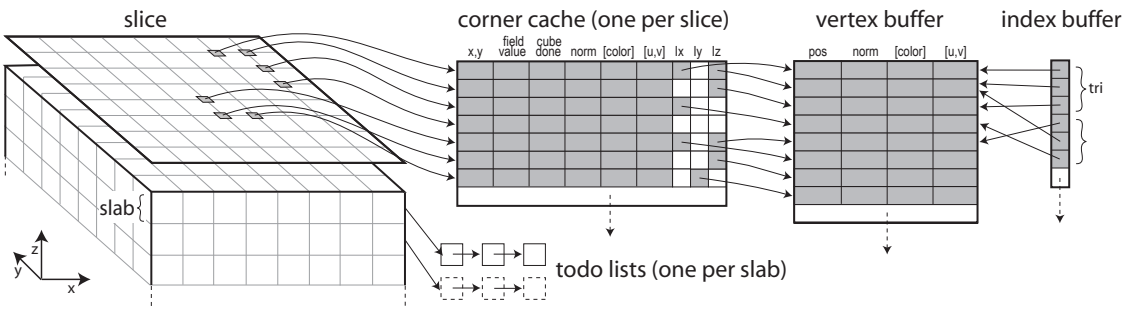

Figure 3: *Data structures used in the Marching Slices algorithm.*

gonized, we allocate two *slice caches* for the slice above and the slice below the slab to store data that may be reused by an adjacent slab. A slice cache consists of a 2D *corner index array* of size $N_x+1 \times N_y+1$, and a dynamically resizing vector of corner values called a *corner cache*. Each element of the corner index array consists of an index which can be used to retrieve a computed corner from the corner cache (or 0 if the corner has not been computed), and two boolean "done" flags to indicate whether the cubes above and below the slice at that location have been rendered.

For corners that have already been visited, the corner cache array stores an (x,y) tuple corresponding to the location of the corner (which is later used for clearing the corner index array), the calculated floating point field value and normal, an optional user-defined structure for data such as colors and texture coordinates and three vertex indices (I_x, I_y, I_z). Each non-zero vertex index refers to an entry in a vertex buffer for the three possible vertices located between the corner at location (x,y,z) and the three other neighboring corners at $(x+1,y,z)$, $(x,y+1,z)$ and $(x,y,z+1)$. Each entry in the vertex buffer stores an (x,y,z) position, normal and optional information such as color, and texture coordinates. Finally, we keep an index buffer which references vertices in the vertex buffer and defines the set of triangles to be drawn in the generated mesh.

Because the marching slices algorithm only traverses cubes that contain the isosurface, one or more starting points which we refer to as *seed cubes*, or simply *seeds*, must be found for each disconnected surface. To ensure that the isosurface surrounding each particle is found, seed cubes are generated for all particles in a block. This is done by snapping to the corner closest to the particle position, evaluating the field value, and then stepping down in the negative z direction one corner at a time until a corner with a field value below the threshold is found. An entry is then pushed into the todo list of the slab that contains the transition in field values with the (x,y) position of the cube that the seed particle was in. The steps are taken in the negative z direction because we prefer to render slices from bottom to top and would like to find seed points as close to the bottom as possible. If seeding traverses farther than R_c down from a particle center without finding an isosurface, the search is terminated, as such a particle must have another particle below it which will find the surrounding surface. If the seeding algorithm steps all the way down to slice 0, a seed cube will be found because field values at the boundary of a block are forced to 0 so that all surfaces are closed, as discussed in Section 3.

Once seeding completes, the marching slices algorithm begins to polygonize the isosurface (Algorithm 1). The algorithm repeatedly polygonizes the bottom-most slab containing an entry in its todo list, until all todo lists are empty. Before entering a slab, the algorithm ensures that the slice caches and corner caches have been allocated for the slices above and below the slab. It then proceeds to polygonize the slab using a variant of surface-following marching cubes that visits only the connected cubes in the current slab. The algorithm repeatedly pops a cube from the slab's todo list and makes sure it hasn't been polygonized by checking the done fields in the slice above and below. It then marks the cube as visited and polygonizes the cube. If polygonization indicates that unvisited neighboring cubes contain the isosurface, they are added to the todo list of their respective slabs. This procedure repeats until the todo list of the current slab is emptied.

Algorithm 1 The Marching Slices Algorithm

$slabs[].todo\_list \leftarrow findAllSeedCubes()$
while one or more slabs has a non-empty todo list **do**
  $z \leftarrow getZOfLowestSlabWithTodoItems()$
  $slab_{current} \leftarrow slabs[z]$
  $slab_{below} \leftarrow slabs[z-1]$
  $slab_{above} \leftarrow slabs[z+1]$
  $slice_{below} \leftarrow slices[z]$
  $slice_{above} \leftarrow slices[z+1]$
  $ensureAllocated(slice_{below})$
  $ensureAllocated(slice_{above})$
  while $!slab_{current}.todo\_list.empty()$ **do**
    $(x,y) \leftarrow slab_{current}.todo\_list.pop()$
    if $slice_{below}[x,y].done_{above}$ *or* $slice_{above}[x,y].done_{below}$ **then**
      continue
    $slice_{below}[x,y].done_{above} \leftarrow$ **true**
    $slice_{above}[x,y].done_{below} \leftarrow$ **true**
    $corner_0 \leftarrow lookupOrEval(slice_{below}[x,y])$
    $corner_1 \leftarrow lookupOrEval(slice_{below}[x+1,y])$
    $corner_2 \leftarrow lookupOrEval(slice_{below}[x,y+1])$
    $corner_3 \leftarrow lookupOrEval(slice_{below}[x+1,y+1])$
    $corner_4 \leftarrow lookupOrEval(slice_{above}[x,y])$
    $corner_5 \leftarrow lookupOrEval(slice_{above}[x+1,y])$
    $corner_6 \leftarrow lookupOrEval(slice_{above}[x,y+1])$
    $corner_7 \leftarrow lookupOrEval(slice_{above}[x+1,y+1])$
    if cube is not in margin **then**
      $polygonize(corner_{0-7})$
    $(do_{above}, do_{below}, do_{left}, do_{right}, do_{front}, do_{back}) \leftarrow cubesToDo(corner_{0-7})$
    if do_{above} *and* $!slice_{above}[x,y].done_{above}$ **then**
      $slab_{above}.todo\_list.push(x,y)$
    if do_{below} *and* $!slice_{below}[x,y].done_{below}$ **then**
      $slab_{below}.todo\_list.push(x,y)$
    if do_{left} *and* $!slice_{below}[x-1,y].done_{above}$ **then**
      $slab_{current}.todo\_list.push(x-1,y)$
    if do_{right} *and* $!slice_{below}[x+1,y].done_{above}$ **then**
      $slab_{current}.todo\_list.push(x+1,y)$
    if do_{front} *and* $!slice_{below}[x,y-1].done_{above}$ **then**
      $slab_{current}.todo\_list.push(x,y-1)$
    if do_{back} *and* $!slice_{below}[x,y+1].done_{above}$ **then**
      $slab_{current}.todo\_list.push(x,y+1)$
  end while
  if $slab_{below}.todo\_list.empty()$ **then**
    $deallocate(slice_{below})$
  if $slab_{above}.todo\_list.empty()$ **then**
    $deallocate(slice_{above})$
end while

Once the todo list in a slab is emptied, a check is performed to see if the cached data for the top or bottom slice can be cleared. The top slice's cache can be cleared when the todo list in the slab above is empty, and similarly, the bottom slice's cache can be cleared when the todo list in the slab below is empty. All cleared slices' corner index arrays are reset by clearing nonzero indices and done flags using the (x, y) entries in their respective corner caches and returned to a common memory pool. The algorithm completes once there are no more todo list items for any slab in the block, at which point the index and vertex buffers can be flushed out to a video card.

To prove that each cube is visited only once, we need only show that the slice caches (storing the cube-done flags) above and below a slab are only deallocated once the visited cubes in a slab can't possibly be re-visited. A slice cache is allocated when corner values are needed, that is, when a slab above or below is traversed. The done flags in the slice cache prevent revisitation of processed cubes in the current slab. They also prevent the addition of cubes that have already been processed to the todo list of an adjacent slab. A slice is deallocated only when the slabs above and below have empty todo lists, at which point, all the connected cubes in both slabs must have been traversed. When this condition is met, there is no danger in deallocating a slice because it no longer contains cube-done values and cached data that can be reused by unvisited cubes.

We can estimate how many slices need to be kept in memory at the same time. We know that if the seeding algorithm were able to find all the local minimum points in the isosurface and seed there, we could traverse the slabs monotonically from bottom to top using only two cache slices to render the entire block. However, because in reality the seeding algorithm does not always find all the local minima (Figure 4), marching slices sometimes needs to step downwards to render connected cubes in slabs below that were missed. Each time the step direction reverses, an additional slice is needed to store the abandoned frontier until the algorithm returns to that slice. From experimental analysis, the typical number of slices used to render a block is three, while situations where more than three slices are necessary are rare and short-lived.

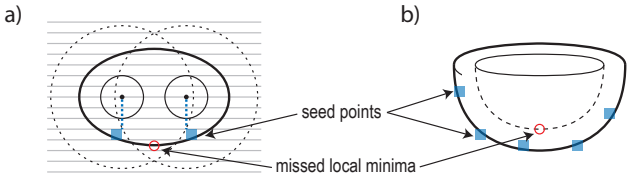

Figure 4: *Cases where seeding fails to find local minima a) at the bottom of a surface, seeding is sometimes off by a single slab b) inside of an upright bowl shape there may be no seeds.*

Compared to the equivalent surface-following marching cubes algorithm which would have kept all $N_z + 1$ slices and associated corner values in memory, our approach uses much less working memory. It is also faster than surface-following due to an increase in the locality of memory accesses. In fact, the amount of cache memory used by our approach with a $110 \times 110 \times 110$ grid is below 1MB (See Results). Additionally, while working on a slice, memory accesses become 2D rather than 3D, and various positional calculations can be reused further improving performance. Our approach also runs much faster than hash map based approaches by avoiding the cost of hash map lookups and uses significantly less memory because the hash-based approaches keeps all corner values in memory, while ours keeps corner values only for the currently active slices (See Results).

5 Transparency and Culling

In some situations, it is advantageous to generate isosurfaces in a front-to-back or back-to-front order. For example, in order to properly render transparent surfaces on most graphics hardware, it is necessary to draw triangles in a back-to-front order. When drawing opaque surfaces, it is usually faster to draw in a front-to-back order to take advantage of per-fragment early-z-culling available on most graphics hardware. Because our algorithm generates surface in a slice-by-slice fashion, either can be achieved nearly for free with a few simple modifications. The first step is to rotate the rendering volume perpendicularly to the viewer so that the sweep happens in the desired direction, and to rotate the particles in the opposite direction so they remain in the same place. Secondly, the block subdivision routine must be modified to render blocks in the desired order according to distance from the viewer. Within a block, because of the structure of the marching slices algorithm, slabs are polygonized in an approximately unidirectional order. The order is only broken when the algorithm steps back to do work on a slab "below" the current slab, which happens when seeding fails to find the bottommost portion of surface enclosing a particle. Because this is rare and the polygons that are incorrectly sorted rarely occlude each other, in practice this is good enough for use in interactive applications without any visible artifacts (See Results). In cases when absolutely perfect back-to-front sorting is required, instead of a global index buffer, separate index buffers can be allocated per slab and flushed out in a back-to-front order for each rendered block. Within an individual cube, we can guarantee that polygons are output back-to-front by preprocessing the static lookup tables which are used by the polygonization routine. Here, we can also ensure that polygons are output with consistent winding to facilitate back-face culling. Because both of these optimizations modify static lookup tables, there is no run-time penalty for their use.

In cases where it is necessary to keep the rendering volume aligned with a fixed basis vector rather than rotating it with respect to the user, front-to-back or back-to-front rendering can still be achieved. In these cases, the algorithm must be modified to perform a sweep along the axis that is most closely aligned to the user. In this approach, if the viewer has a wide field of view, they may be able to look edgewise through a slice, in this case, the cubes within each slice must be cached and individually sorted by distance to viewer before they are rendered.

6 Particle Lookup Cache

Field value calculations are in the inner loop of both the marching cubes and marching slices algorithm and are generally the place where these algorithms spend the majority of computing time. This makes them the major target for optimization, and is the reason why we cache and reuse these values at sample points.

Because the field of influence of the particles is limited to a fixed cutoff radius, it is possible to optimize the field calculations by iterating only over the particles within R_c from a given sample point using a spatial data structure which we refer to as a *particle lookup cache*. Observing that lookups are only performed at field corners, we relax the requirements for the particle cache by only requiring it to know which particles influence the field at a corner, and not at any arbitrary point in space. At the same time, in order to avoid wasting time in the field calculation routine, we require that the particle cache give an exact solution, without returning any particles that are outside of R_c. A simple way to implement such a particle cache would be to build a linked list at each corner referencing all the particles that are within R_c of the corner. However, the insertions spanning a sphere with radius R_c, would be computationally expensive and consume a considerable amount of memory. We can

greatly improve upon this approach by observing that the linked list traversal during field calculation can serve double duty by simultaneously performing a one dimensional search through 3-space. This means that instead of marking a spherical 3D region, we only need to mark a 2D region for each particle in the shape of a disk (Figure 5).

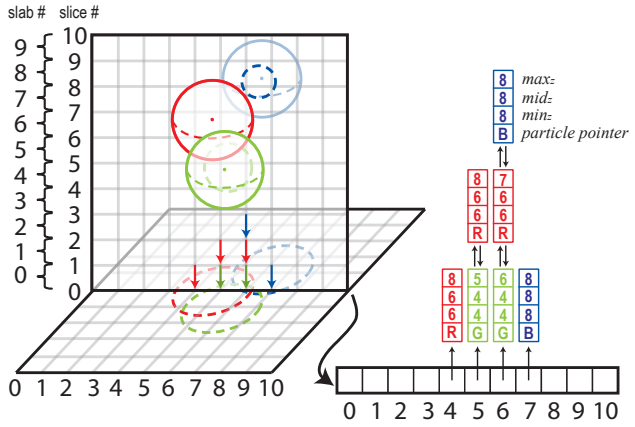

Figure 5: *Cutaway diagram of insertion into particle cache.*

Our approach, which we call a *2D projection 1D lookup* operates as follows: We allocate a single 2D array of dimension $Nx+1$ by $Ny+1$ called the projection slice. Each element in the array is simply a pointer to a doubly linked list. A list pointed to by element (x,y) in the projection slice will contain one entry for each of the particles with influence on the corners in column (x,y). Each element in the list contains the integral values min_z and max_z specifying the range of slices in which the particle has influence, an integer mid_z specifying the slab containing the center of the particle, and a pointer to the particle. The lists are built by quick-sorting all particles from smallest to largest z coordinate, and then, for each particle, inserting a list element into each (x,y) column where it has influence. In order to force the field values to 0 along the boundaries of a block as described in section 3, we avoid inserting elements into columns that are on a boundary, and clamp min_z and max_z to values between 1 and N_z. Thus, there is no additional cost for enforcing this constraint during field value calculation. For particles with a field of influence smaller than R_c, it is straightforward to modify the insertion step to shrink the radius in which the particle will be found and ensure an exact lookup.

Whenever a lookup is performed, the current corner location (x,y) is indexed in the projection slice to access a list containing all particles that have influence in that column. Next, to find the starting point, the list is searched for the bottommost entry with $mid_z \geq z - \lceil R_c/S \rceil$. The list is then traversed upwards to the topmost entry with $mid_z \leq z + \lceil R_c/S \rceil$. This traversal effectively finds every particle within a cylinder of radius R_c and height $2R_c$ around the lookup point. To cull this list down to the sphere of radius R_c, quick integer comparisons are performed for every element to test that $min_z \leq z$ and $max_z \geq z$. If so, the element has influence at the lookup point, and is handed to the field calculation routine. Since we are likely to do lookups above and below recent lookups, to speed up finding of the starting point, the pointer at location (x,y) in the projection slice is updated with the first element found by each particle lookup in that column.

Because the cylindrical volume covered by the list traversal is efficiently culled down to a spherical volume with two integral tests, the 1D search is only slightly less efficient than a simple linked list traversal, yet allows for a significant reduction in the time and memory cost of particle insertion. Furthermore, due to the nearly unidirectional traversal of the marching slices approach, the doubly linked list at a column is usually traversed only once while looking for starting points. Thus, the work done to find starting points is small and proportional to the number of element inserted into the particle cache.

To estimate the memory consumption of our approach, we note that approximately $\pi R_c^2/S^2$ elements are inserted for each particle. With a cutoff radius $R_c = 3$ and cube size $S = 0.8$, approximately 45 elements are inserted per particle. If each of these is 16 bytes in size, the cache consumes 720 bytes per particle, with a cache of 1,000 particles consuming a reasonable 720KB of data.

For situations where many more than 1,000 particles may be inserted into a block, we can reduce memory usage by keeping only the necessary z range of particles in cache. This can be done by removing particles that fall outside the useful z range or by keeping multiple projection slices, each one storing only the elements on a single slab.

Compared to the well known class of approaches used by Wyvill *et al.* [WMW86b, WMW86a], Müller *et al.* [MCG03, MST04a, MST*04b] and Triquet [TMC01], which use either a point insertion and a 3D lookup or a 3D insertion and point lookup, our approach is the only one that does an exact lookup, without returning any particles outside of R_c. This is because these approaches use a separate grid for particle lookups with cube size R_c (much coarser than the marching cubes/slices grid) in order to keep the number of cubes traversed during insertion or lookup down to a reasonable $3 \times 3 \times 3$ volume. Thus, these approaches return 6.5 times more particles than our approach by looking in a $(3R_c)^3$ volume, rather than a $(4/3)\pi R_c^3$ volume, which they must then cull by performing floating point distance calculations for each particle. With these approaches, if the lookup grid resolution is increased to reduce the number of false positives, the insertion/lookup costs quickly grow and dominate overall isosurface extraction costs, especially when each cube access requires a hash map lookup as suggested by previous approaches for unbounded rendering.

In our benchmarks, the memory usage of our approach was never significantly higher than with previous approaches because we build and destroy our particle cache on a per block basis, and can be further reduced by removing unneeded particles from the cache (as explained previously). In terms of time, although our approach does a 2D projection for each particle, it performed significantly faster than previous approaches in all of our benchmark tests because the number of lookups done per block is typically much greater than the number of insertions, and because our lookups are fast and don't return false positives.

7 Results

In order to accurately measure the performance of our technique and compare it to alternative approaches, we created simple particle simulations of a fountain, an amorphous blob, and a particle explosion (Figure 6). We use these simple scenarios as benchmarks to compare memory usage and rendering speed across different rendering approaches on a 3.2Ghz Intel Core 2 Duo processor with 4MB of L2 cache (only one core was used for computation). The approaches we compared are:

1. Our new approach: Marching slices with block subdivision and a 2D projection 1D lookup particle cache.
2. Surface-following marching cubes with blocking algorithm to allow for unbounded rendering, and a point insert, 3D lookup cache.

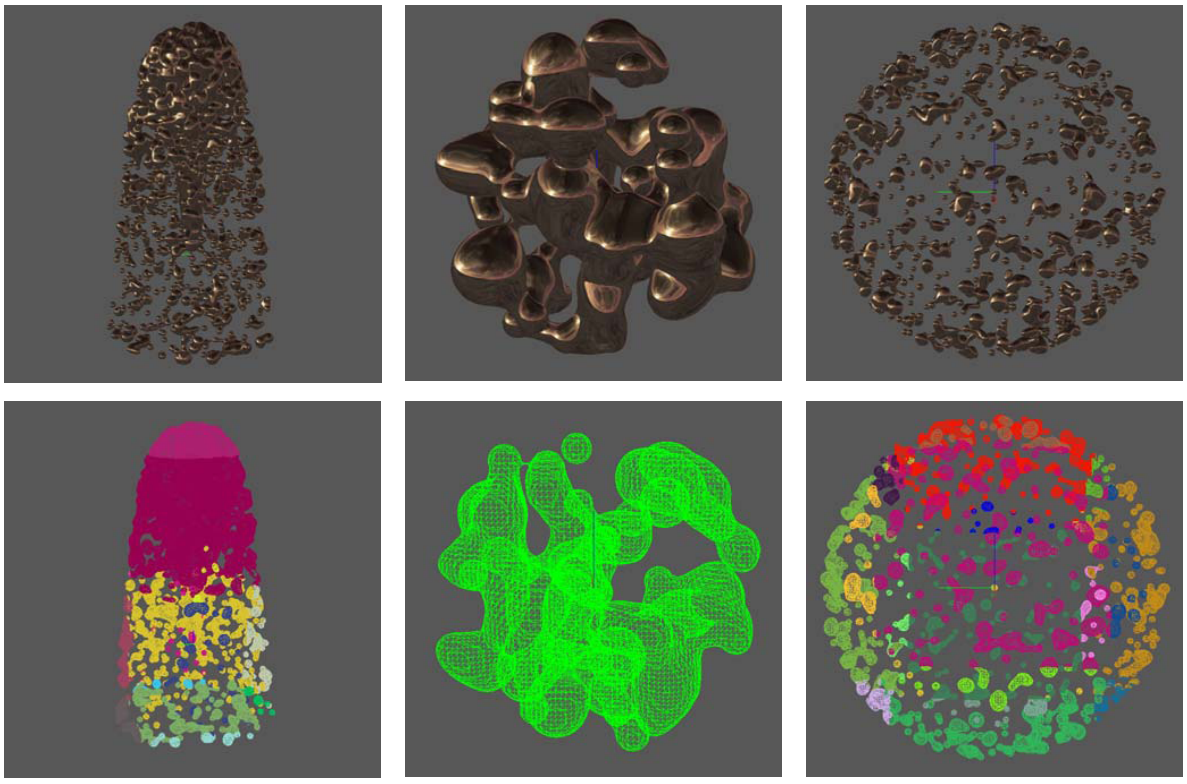

Figure 6: *Screenshots of simulations used for performance comparison. Top row from left to right: fountain, blob, and explosion simulations. Bottom row: wireframe views of simulation with blocks identified by different colors.*

3. Surface-following marching cubes using a hash map to store corners, and a point insert, 3D lookup cache also using a hash map for particle lookup.

All of these approaches were instrumented to measure frame rates, memory usage, and cubes and triangles rendered per second (Figure 7). We measured memory usage separately for the corner caches, which we refer to as *Render Mem* and for the particle lookup caches which we refer to as *Cache Mem*. These fields only measure active memory which is both written to and read from during algorithm execution, and do not measure the memory used for the vertex and index buffers, since these are statically allocated, and in our benchmarks, are the same size (960KB) for all three approaches.

We first compare Approaches 2 and 3 to evaluate the costs and benefits of using the block subdivision component solely versus the hashing approach. We see that over all the test scenarios, frame rates, cubes/sec and tris/second are approximately doubled when using block subdivision instead of hashing. However, memory usage is higher, mainly due to the large 3D arrays used for lookups of field values and nearby particles.

By replacing surface-following marching cubes with marching slices, and using a 2D insertion 1D lookup particle cache along with the block subdivision approach, we realize a significant performance improvement over the other approaches, with our algorithm performing 1.5 to 2.5 times faster than marching cubes with block subdivision and 3.5 to 6.9 times faster than the hashing approach across our benchmarks. We also significantly improve upon the biggest drawback of surface-following marching cubes, reducing memory usage by a factor of 8 to 43, and improving upon the memory usage of the hashing approach by a factor of 2 to 15 across our benchmarks. However, we notice that our particle cache memory footprint, while only a fraction of overall memory usage, is still 2 to 5 times larger than that of the hashing approach. This is a reasonable tradeoff given the increase in overall performance, and can be improved greatly in cases where large quantities of particles are rendered by using the approaches mentioned in section 6 for keeping particles cached only over the active slices.

Because of the small memory footprint of our approach, it can reside fully in the main memory of any modern computer along with other components of an interactive application (which in games include AI, physics, networking and world rendering). This is critical in interactive applications where virtual memory paging is unacceptable because it causes intermittent stalls, degrades performance and frustrates users. Furthermore, because the memory footprint is so small, the algorithm can effectively take advantage of processor level 2 cache, avoiding cache misses which can stall the pipeline of a modern processor for as much as 500 cycles. Thus, we get a similar benefit as in out-of-core algorithms, but at one level higher on the memory hierarchy! This characteristic is especially beneficial on multi-core hardware which shares cache memory between multiple processes and hardware with small amounts of working memory such as hand held devices and cell processors.

We demonstrate the suitability of this algorithm in complex, fully interactive virtual environments. For our first experiment (Video 1), we constructed a fountain simulated with $1,300$ particles and placed it in an outdoor video-game environment with virtual characters. The player can interact with the simulation by throwing objects and explosives into the fountain. The demo uses back-to-front sorted rendering of polygons, and a refractive and reflective water shader for transparency, and runs interactively at a frame rates between 50fps at 60fps. Our second experiment (Video 2) uses our algorithm to animate an amorphous particle monster. The monster is composed of 250 particles which are animated with several

Fountain (3000 particles, Rc = 3.0, T = 0.2, S = 1.0)

Approach	Cubes / Sec	Tris / Sec	Render Mem	Cache Mem	FPS
1	1,636,312	3,171,223	0.517	0.894	46.6
2	947,294	1,836,490	7.474	8.020	27.2
3	462,378	898,280	4.756	0.368	13.1

Fountain (3000 particles, Rc = 3.0, T = 0.2, S = 0.75)

Approach	Cubes / Sec	Tris / Sec	Render Mem	Cache Mem	FPS
1	1,859,083	3,657,317	0.741	1.548	29.1
2	1,033,835	2,033,716	10.513	8.022	16.2
3	477,330	939,837	8.611	0.364	7.4

Fountain (3000 particles, Rc = 3.0, T = 0.2, S = 0.5)

Approach	Cubes / Sec	Tris / Sec	Render Mem	Cache Mem	FPS
1	1,706,667	3,388,940	0.749	0.408	11.7
2	968,568	1,924,154	8.827	8.002	6.5
3	451,350	899,031	2.051	0.346	3.0

Blob (100 particles, Rc = 3.0, T = 0.2, S = 0.3)

Approach	Cubes / Sec	Tris / Sec	Render Mem	Cache Mem	FPS
1	2,145,530	4,288,108	0.246	0.137	77.7
2	1,228,188	2,458,939	8.655	8.000	49.3
3	559,283	1,118,880	3.179	0.022	22.5

Explosion (1000 particles, Rc = 3.0, T = 0.2, S = 0.6)

Approach	Cubes / Sec	Tris / Sec	Render Mem	Cache Mem	FPS
1	1,967,465	3,811,094	0.437	0.099	33.2
2	790,634	1,533,241	8.495	8.001	13.3
3	298,943	586,777	8.211	0.052	4.8

Figure 7: *Performance comparison (memory is measured in MB). R_c is the cutoff radius, T is the threshold (a threshold of 0.2 corresponds to an isosurface of radius 1 for a single particle), S is the size of the cube grid.*

procedural behaviors such as emerging from a sewer, growing tentacles, forming a ball, coating a wall, and chasing players. The tentacles, when grown, are cylindrically textured by interpolating per particle basis vectors and length coordinates onto the surface using a custom field calculation routine. These are then used to generate u, v texture coordinates into a preloaded texture by a custom vertex calculation routine. We also dynamically control the radius of a tentacle along its length by modulating the strength of field functions of the tentacle particles. The monster is scripted to actively move around the scene, interact with objects, and chase the player. Furthermore, it can be blown to bits with explosive, and will automatically reassemble into its original shape. This demo runs interactively at 50-60fps depending on the number of physics objects the monster is colliding with and at 60-70fps during replay. Demo videos for both of these experiments along with various screenshots are included as supplementary materials.

8 Conclusion and Future Work

In this paper, we have presented a new technique for real-time particle isosurface extraction. This approach consists of three novel components: a block subdivision algorithm which divides the render volume into seamless blocks, marching slices which renders isosurface in a slice-by-slice manner, and a 2D insert 1D lookup particle cache which enables fast and accurate lookups for field contributing particles. Using our technique, we demonstrated significant memory reductions and speed improvements over existing approaches for unbounded particle isosurface extraction. We also employed this approach in a video game environment, showing how our technique allows for rendering of dynamic particle-based entities with quality approaching that of offline techniques and has the potential to bring novel experiences to users of interactive applications.

Although we presented this algorithm running in a single thread, we have experimented with parallelized algorithms running on a quad core machine. We found that there is potential for a nearly linear speedup by parallelizing at the block-level for particle distributions that span more than one block, where incidentally, performance improvements are most important. Given that the latest generation of computing and game platforms have multiple cores, this is an important direction for future exploration. Furthermore, because this algorithm operates with 2D data structures, it should also be amenable to implementation on programmable graphics hardware, which is optimized for 2D texture lookups, and where we can expect tremendous performance gains, and many new and exciting applications.

Figure 8: *Screenshot of blob monster.*

Acknowledgments

We acknowledge Fang Cheng, Philip Davidson, Yotam Gingold and Jeep Barnett for their help with diagrams and editing, Lars Jensvold for help with video editing, and everyone involved at New York University and Valve Software for their help and support. We also thank the reviewers for their insightful comments and suggestions.

References

ADAMS B., LENAERTS T., DUTRÉ P.: Particle splatting: Interactive rendering of particle-based simulation data. *Technical Report CW 453, Katholieke Universiteit Leuven* (2006).

BLINN J. F.: A generalization of algebraic surface drawing. *Computer Graphics and Interactive Techniques 1*, 3 (1982), 235–256.

BLOOMENTHAL J.: Polygonization of implicit surfaces. *Computer Aided Geometric Design 5*, 4 (1988), 341–355.

HILTON A., ILLINGWORTH J.: Marching triangles: Delaunay implicit surface triangulation. *Technical Report CVSSP 01, University of Surrey* (1997).

LORENSEN W. E., CLINE H. E.: Marching cubes: A high resolution 3D surface construction algorithm. *Computer Graphics and Interactive Techniques 21*, 4 (1987), 163–169.

MÜLLER M., CHARYPAR D., GROSS M.: Particle-based fluid simulation for interactive applications. *Symposium on Computer Animation* (2003), 154–159.

MÜLLER M., KEISER R., NEALEN A., PAULY M., GROSS M., ALEXA M.: Point based animation of elastic, plastic and melting objects. *Symposium on Computer Animation* (2004), 141–151.

MÜLLER M., SCHIRM S., TESCHNER M.: Interactive blood sim-

ulation for virtual surgery based on smoothed particle hydrodynamics. *Technology and Health Care 12*, 1 (2004), 25–31.

MÜLLER M., SCHIRM S., TESCHNER M., HEIDELBERGER B., GROSS M. H.: Interaction of fluids with deformable solids. *Computer Animation and Virtual Worlds 15*, 34 (2004), 159–171.

REEVES W. T.: Particle systems – A technique for modeling a class of fuzzy objects. *Computer Graphics and Interactive Techniques 17* (1983), 359–376.

SIMS K.: Particle animation and rendering using data parallel computation. *Computer Graphics and Interactive Techniques 24* (1990), 405–413.

SZELISKI R., TONNESEN D.: Surface modeling with oriented particle systems. *Computer Graphics and Interactive Techniques 26*, 2 (1992), 185–194.

TESCHNER M., HEIDELBERGER B., MÜLLER M., POMERANTES D., GROSS M. H.: Optimized spatial hashing for collision detection of deformable objects. *Vision, Modeling, and Visualization* (2003), 47–54.

TRIQUET F., MESEURE P., CHAILLOU C.: Fast polygonization of implicit surfaces. *WSCG (Plzen, Czech Republic) 2* (2001), 283–290.

WITKIN A. P., HECKBERT P. S.: Using particles to sample and control implicit surfaces. *Computer Graphics and Interactive Techniques 28* (1994), 269–277.

WYVILL B., MCPHEETERS C., WYVILL G.: Animating soft objects. *The Visual Computer 2*, 4 (1986), 235–242.

WYVILL G., MCPHEETERS C., WYVILL B.: Data structure for soft objects. *The Visual Computer 2*, 4 (1986), 227–234.